

Scalable Generation of Random Graphs

Dissertation

zur Erlangung des Doktorgrades

der Naturwissenschaften

vorgelegt beim Fachbereich 12

der Johann Wolfgang Goethe Universität

in Frankfurt am Main

von

Manuel Penschuck

aus Frankfurt

Frankfurt (2020)

(D 30)

vom Fachbereich 12. der
Johann Wolfgang Goethe – Universität als Dissertation angenommen.

Dekan: Prof. Dr. L. Hedrich

Gutachter: Prof. Dr. U. Meyer, Prof. Dr. P. Sanders, Prof. Dr. G. Schnitger

Datum der Disputation: 16. April 2021

Acknowledgements

My name may stand alone on the cover of this thesis, yet there were many people —too many to acknowledge them here individually— who supported me along the way.

First and foremost, Ulrich Meyer, thank you for all your help and guidance as my advisor. You were the one who suggested the field of random graphs in the first place, and continuously provided knowledge, exposure, means and freedom, enabling me to learn (of) new topics and to tackle the problems that fascinate me.

I want to thank all my coauthors, especially Hung, for the countless discussions and opportunities to learn and discover. I enjoyed our shared struggle through the many small set-backs and the —then even more satisfying— moments of humble victories.

The experience would not have been the same without everyone of the “third floor”; especially Bert, David, Hannes, Mario, as well as Alex and Volker — thank you for the great and exciting years.

I am grateful to my extended family and friends — their open ears, encouragements and distractions helped me keep on track (more or less); though, I must say, I could have done with less frequent inquiries about my progress. Special gratitude is due to my parents for their immense support on my not-so-direct path leading to computer science.

Last but, most certainly, not least, Mercedes, thank you for always being at my side. I cannot begin to overstate the impact you made on me (and also this thesis).

Manuel Penschuck
4. December 2020

Deutsche Zusammenfassung

Netzwerkmodelle spielen in verschiedenen Wissenschaftsdisziplinen eine wichtige Rolle und dienen unter anderem der Beschreibung realistischer Graphen [34, 261, 14]. Sie werden häufig als Zufallsgraphen formuliert [59] und stellen somit Wahrscheinlichkeitsverteilungen über Graphen dar. Meist ist die Verteilung dabei parametrisiert und ergibt sich implizit, etwa über eine randomisierten Konstruktionsvorschrift. Ein früher Vertreter ist das $\mathcal{G}(n, p)$ Modell [148], welches über allen ungerichteten Graphen mit n Knoten definiert ist und jede Kante unabhängig mit Wahrscheinlichkeit p erzeugt.

Ein aus $\mathcal{G}(n, p)$ gezogener Graph hat jedoch kaum strukturelle Ähnlichkeiten zu Graphen, die zumeist in Anwendungen beobachtet werden. Daher werden populäre Modelle so gestaltet, dass sie mit hinreichend hoher Wahrscheinlichkeit gewünschte topologische Eigenschaften erzeugen. Beispielsweise ist es ein gängiges Ziel die nur unscharf definierte Klasse der sogenannten *komplexen Netzwerke* nachzubilden, der etwa viele soziale Netze zugeordnet werden. Unter anderem verfügen diese Graphen in der Regel über eine Gradverteilung mit schweren Rändern (*heavy-tailed*), einen kleinen Durchmesser, eine dominierende Zusammenhangskomponente, sowie über überdurchschnittlich dichte Teilbereiche, sogenannte *Communities* [34].

Die Einsatzmöglichkeiten von Netzwerkmodellen gehen dabei weit über das ursprüngliche Ziel, beobachtete Effekte zu erklären, hinaus. Ein gängiger Anwendungsfall besteht darin, Daten systematisch zu produzieren. Solche Daten ermöglichen oder unterstützen experimentelle Untersuchungen, etwa zur empirischen Verifikation theoretischer Vorhersagen oder zur allgemeinen Bewertung von Algorithmen und Datenstrukturen.

Hierbei ergeben sich insbesondere für große Probleminstanzen Vorteile gegenüber beobachteten Netzen. So sind massive Eingaben, die auf echten Daten beruhen, oft nicht in ausreichender Menge verfügbar, nur aufwendig zu beschaffen und zu verwalten, unterliegen rechtlichen Beschränkungen, oder sind von unklarer Qualität.

In der vorliegende Arbeit betrachten wir daher algorithmische Aspekte der Generierung massiver Zufallsgraphen anhand von etablierten Netzwerkmodellen. Zu diesem Zweck entwickeln wir praktisch sowie analytisch effiziente Generatoren. Unsere Algorithmen sind dabei jeweils auf ein geeignetes Maschinenmodell hin optimiert. Hierzu entwerfen wir etwa klassische sequentielle Generatoren für Registermaschinen, Algorithmen für das *External Memory Model*¹ (*EMM*, siehe unten), und parallele Ansätze für verteilte oder Shared-Memory Maschinen.

Struktur

Im Folgenden führen wir zunächst das *EMM* ein und geben dann eine Übersicht über die in dieser Arbeit präsentierten Ergebnisse. Komplexitätsschranken beziehen sich dabei häufig auf die Knoten- und Kantenanzahl des generierten Graphen, die wir mit n beziehungsweise m bezeichnen. Die Diskussion folgt der Gliederung der Arbeit, deren Hauptteil (Kapitel 3 bis 8) sich mit drei Arten von Netzwerkmodellen befasst.

[Übersichtsartikel zu Netzwerkmodellen:](#)

[📖 Kapitel 2](#)

[\(Zufalls-\)Graphen:](#)

[📖 Abschnitt 1.2.2 ff.](#)

[Diskussion \$\mathcal{G}\(n, p\)\$:](#)

[📖 Abschnitt 1.2.4.1 ff.](#)

[Komplexe Netzwerke:](#)

[📖 Abschnitte 1.2 und 2.2](#)

[Maschinenmodelle:](#)

[📖 Abschnitt 1.3](#)

¹Aus Konsistenzgründen bezeichnen wir Modelle mit den englischen Namen des Hauptteils.

External Memory Model

Das *External Memory Model* (*EMM*) nach Aggarwal und Vitter [7] ist ein theoretisches Maschinenmodell. Es abstrahiert Speicherhierarchien moderner Computer und begünstigt Algorithmen mit hoher Datenlokalität.

Das *EMM* definiert zwei Speichertypen: einen schnellen internen Hauptspeicher, der M Datenworte umfasst, sowie einen langsamen externen Speicher unbeschränkter Größe. Die Ein- und Ausgabe befinden sich im externen Speicher, wobei Daten nur im internen Speicher bearbeitet werden können. Informationen werden in Blöcken von B Worten zwischen den Speichertypen bewegt; wir bezeichnen einen solchen Blocktransfer als einen I/O. Das Entwurfsziel eines I/O-effizienten Algorithmus ist es, ein Problem mit möglichst wenig I/Os zu lösen. Dafür werden häufig die folgenden algorithmischen Primitive verwendet:

$$\text{scan}(n) = \Theta\left(\frac{n}{B}\right) \text{ I/Os}$$

- Das Lesen und Schreiben von n Werten eines zusammenhängenden Bereichs wird als *scannen* bezeichnet und benötigt $\text{scan}(n) = \Theta(n/B)$ I/Os.

$$\text{sort}(n) = \Theta\left(\frac{n}{B} \log_{M/B}\left(\frac{n}{B}\right)\right) \text{ I/Os}$$


- Das vergleichsbasierte Sortieren eines zusammenhängenden Speicherbereichs mit N Werten benötigt $\text{sort}(n) = \Theta(n/B \log_{M/B} n/B)$ I/Os.

Time Forward Processing:
 [Abschnitt 5.2.2](#)

- Prioritätswarteschlangen können n Operationen in $\text{sort}(n)$ I/Os ausführen und bilden die Basis des sogenannten *Time Forward Processing*, einer Entwurfsmethode mit der sich unstrukturierte Zugriffe oft vermeiden lassen.

Für praktische Werte von n , B und M gilt $\text{scan}(n) \lesssim \text{sort}(n) \lll n$. Für viele intuitiv nicht-triviale Probleme stellt sort eine untere Komplexitätsschranke dar.

Preferential Attachment (Kapitel 3)

Übersicht Pref. Attach.:
 [Abschnitt 2.4.2](#)

Preferential Attachment beschreibt eine von Pólya-Urnen [117] bekannte positive Rückkopplung, die umgangssprachlich auch als „der Teufel [*erleichtert sich*] immer auf den größten Haufen“ bekannt ist. Es handelt sich somit um einen Prozess, bei dem das Ziehen einer zufälligen Stichprobe zu einem Zeitpunkt die Wahrscheinlichkeit erhöht, dieselbe Stichprobe später erneut zu ziehen.

Preferential Attachment liegt einer Reihe von Netzwerkmodellen zugrunde, allen voran dem weitverbreiteten *BA* Modell von Barabási und Albert [32]. Es formuliert einen einfachen stochastischen Prozess, der die Entstehung sogenannter skalenfreier Netzwerke [14] erklärt. Hierbei handelt es sich nach gebräuchlicher Lesart um komplexe Netzwerke, deren Gradverteilung einem Potenzgesetz (Powerlaw-Verteilung) folgt. Die Autoren legen jedoch in ihrem Modell den Fokus auf die Entstehung von Powerlaw-Gradverteilungen in wachsenden Netzwerken und klammern darüber hinausreichende Eigenschaften komplexer Netzwerke (z.B. Communitystrukturen) aus.

Skalenfreie Netze:
 [Abschnitt 1.2.4.2](#)

Das *BA* Modell erzeugt Graphen mittels des folgenden iterativen Prozesses. Wir beginnen mit einem beliebigen Graphen auf n_0 Knoten (häufig dem Kreis C_{n_0} oder der Clique K_{n_0}). Dann fügen wir schrittweise t neue Knoten hinzu, wobei jeder direkt mittels $d < n_0$ zufällig gewählter Nachbarn mit dem vorhandenen Graphen verbunden wird.

Preferential Attachment schreibt nun vor, dass die Nachbarn mit einer Wahrscheinlichkeit proportional zu ihrem gegenwärtigen Grad gezogen werden.

Algorithmisch ist das dynamische und gewichtete Ziehen der Nachbarn von besonderem Interesse. Batagelj und Brandes [35] beschreiben den einfachen Generator, im Folgenden BB-BA genannt, der Graphen in Linearzeit konstruiert. Der Algorithmus erzeugt ein Array von Kanten und nutzt aus, dass jeder Knoten u mit Grad $\deg(u)$ genau $\deg(u)$ mal in diesem Array vorkommt. Somit lässt sich das gewichtete Ziehen auf das Lesen einer uniform gezogenen Position reduzieren.

Während BB-BA auf Registermaschinen mit Einheitskosten optimal agiert, sagt das *EMM* aufgrund der unstrukturierten Zugriffe des Algorithmus stark suboptimales Verhalten voraus. Dies lässt sich experimentell untermauern. So beobachten wir, dass der Durchsatz des Generators um mehrere Größenordnungen einbricht, sobald die Graphgröße den verfügbaren Arbeitsspeicher um nur 2% überschreitet. Wir präsentieren daher die zwei ersten I/O-effizienten BA Generatoren TFP-BA und MP-BA.

- **TFP-BA** ist ein einfacher und leicht erweiterbarer sequentieller Generator, der als Übersetzung von BB-BA in das *EMM* verstanden werden kann. TFP-BA vermeidet die für BB-BA nötigen zufälligen Zugriffe, indem zunächst alle Leseoperationen ausgewürfelt und sortiert, jedoch *nicht* ausgeführt werden. In der Hauptphase wird nun die Kantenliste, ähnlich wie in BB-BA, von Anfang bis zum Ende sequentiell geschrieben. Hierbei werden etwaige Leseanfragen von späteren Positionen nachgeschlagen und mittels I/O-effizienter Prioritätswarteschlange (*Time Forward Processing*) beantwortet.

Wir zeigen, dass TFP-BA $\mathcal{O}(\text{scan}(m_0) + \text{sort}(m))$ I/Os benötigt, wobei m_0 der Kantenanzahl der Eingabe und m der Anzahl erzeugter Kanten entspricht. In der Praxis ist TFP-BA für Ausgaben, deren Größe den Hauptspeicher überschreitet, um Größenordnungen schneller als eine optimierte BB-BA Implementierung.

- **MP-BA** ist ein hoch-optimierter Generator, der sich aus einem parallel-externen Teil und einer GPU-beschleunigten Komponente zusammensetzt. Konzeptionell implementiert MP-BA das für *Preferential Attachment* benötigte dynamische und gewichtete Ziehen mittels eines Binärbaums T , der partiell im externen Speicher gehalten wird. Jedes Blatt in T ist eindeutig einem Knoten u des erzeugten Graphen zugeordnet und mit dessen Grad $\deg(u)$ beschriftet. Innere Knoten speichern die Summe der Blattgewichte ihres linken Teilbaums. Der Baum ist dynamisch, da neue Blätter eingefügt und Knotengewichte erhöht werden.

Preferential Attachment benötigt zwei logische Schritte: Zunächst muss, erstens, ein geeignet gewichteter Nachbar v zufällig aus T gezogen werden um mit diesem eine Kante zu bilden. Danach müssen, zweitens, die Gewichte aller inneren Knoten erhöht werden, in deren linken Teilbaum sich v befindet. MP-BA fasst beide Schritte zu einer Operation zusammen und bearbeitet sie in einer einzigen Traversierung des Baums von der Wurzel zu dem entsprechenden Blatt hin.

Konkret ist jede Operation mit einer natürlichen Zufallszahl x annotiert, wo-

Experimentelle Ergebnisse:
📖 [Abschnitt 3.5](#)

TFP-BA:
📖 [Abschnitt 3.2](#)

Time Forward Processing:
📖 [Abschnitt 5.2.2](#)

MP-BA:
📖 [Abschnitt 3.3](#)

Entscheidungsbaum:
📖 [Abschnitt 3.3.1](#)

bei x kleiner als das Gesamtgewicht des Baums T ist. Wir betrachten nun den Arbeitsschritt an einem inneren Knoten mit Beschriftung y wobei y als Entscheidungsgrenze fungiert: Wenn x kleiner als y ist, wird die Operationen rekursiv an den linken Teilbaum T_L weitergegeben. In Antizipation der Bildung einer Kanten mit einem Knoten in T_L wird außerdem das Gewicht y des linken Teilbaums direkt inkrementiert. Andernfalls wird y von x abgezogen und die so angepasste Operation an den rechten Teilbaum weitergegeben.

Diese Operationen werden asynchron ausgeführt, können unterbrochen und gepuffert werden. Es muss hierbei lediglich sichergestellt werden, dass alle Operationen, die einen Knoten erreichen, in ihrer relativen Ankunftsreihenfolge bearbeitet werden. Durch die Möglichkeit des Pufferns können die Operationen I/O-effizient gestaltet werden. Ferner sind disjunkte Teilbäume unabhängig von einander. Wir entkoppeln daher den Baum in geeigneter Tiefe und bearbeiten alle dort gewurzelten Teilbäume parallel.

I/O-Effizienz:

☞ *Abschnitt 3.3.2*

Paralleler Baum:

☞ *Abschnitt 3.3.4.1*

Durch die hohe Verarbeitungsbandbreite der parallelen Teilbäume kann sich nahe der Wurzel ein Flaschenhals ergeben. Diesen lösen wir mittels eines zweiten Ansatzes zur parallelen Verarbeitung von Anfragen an wurzelnahen Knoten auf. Der zugrunde liegende Algorithmus ist für eine parallele Registermaschine (CREW PRAM) entworfen und wird von uns auf einem Grafik-Rechenbeschleuniger ausgeführt. Hierfür teilen wir die Sequenz von Lese-Update-Operationen in Stapel (Batch) auf und verarbeiten die Operationen innerhalb eines Batch parallel.

PRAM:

☞ *Abschnitt 1.3.3*

Die Parallelverarbeitung muss dabei Abhängigkeiten innerhalb des Batchs erkennen. So können etwa die ersten Operationen die zuvor genannte Entscheidungsgrenze y soweit verschieben, dass eine spätere Anfrage in den linken Teilbaum weitergeleitet werden muss, obwohl dies initial anders wirkte. Bei geeigneter Wahl der Batchgröße tritt dieser Effekt jedoch nur selten auf. Unser PRAM Algorithmus berechnet daher für jede Anfrage einen Sicherheitsabstand zur Entscheidungsgrenze y ; liegt die Anfrage außerhalb dieser Zone, kann die Operation direkt einem Teilbaum zugewiesen werden. Hierdurch kann die Sicherheitszone der verbliebenen Anfragen reduziert werden, sodass der Algorithmus mit hoher Wahrscheinlichkeit innerhalb von konstant vielen Runden terminiert.

PRAM Algorithmus:

☞ *Abschnitt 3.3.4.2*

Wir demonstrieren, dass unsere Implementierung von MP-BA für Graphen, die in den Arbeitsspeicher passen, fast 18 mal schneller als BB-BA ist. Zudem skaliert sie weit darüber hinaus. Verglichen mit einem verteilt parallelen Generator, der auf 48 Zweisockelmaschinen evaluiert wurde, liefert MP-BA kompetitive Ergebnisse.

Implementierung:

☞ *Abschnitt 3.4 ff.*

Simple Graphen mit gegebener Gradsequenz (Kapitel 4 und 5)

Nach der Diskussion eines *Preferential Attachment* Modells ohne Communitystruktur wenden wir uns dem *LFR* Netzwerkmodell [210, 208] zu. Dieses verfügt über explizit erzeugte Gemeinschaften, die ebenfalls ausgegeben werden können. Diese Zusatzinformation macht *LFR* zu einem etablierten Testverfahren, mit dem etwa Algorithmen zur Suche von Communitystrukturen bewertet werden.

LFR:

☞ *Abschnitt 4.3*

Leider skaliert die Geschwindigkeit der *LFR* Referenzimplementierung auf großen Instanzen nur suboptimal. Daher präsentieren wir mit EM-LFR eine schnelle und I/O-effiziente Pipeline, die aus vier neu-entwickelten Algorithmen besteht. EM-LFR folgt dabei der *LFR* Spezifikation [208]. Experimentell demonstrieren wir unter anderem, dass EM-LFR einen Graphen mit 10^{10} Kanten in rund 17 h erzeugen kann, während die Referenzimplementierung in derselben Zeit weniger als 10^8 Kanten erreicht. Somit eignet sich EM-LFR sogar dafür, Eingabeinstanzen für verteilte Algorithmen zu generieren [170].

Im Folgenden legen wir den Fokus auf den komplexesten Bereich der Pipeline. Gegeben sei eine Gradsequenz, d.h. eine Liste von Knotengraden. Gesucht ist eine uniforme Stichprobe aus der Menge aller ungerichteten simplen Graphen (d.h. ohne Mehrfachkanten oder Eigenschleifen), die dieser Gradsequenz folgen. Hierbei handelt es sich um eine gängige Aufgabe in der Netzwerkanalyse, die auch genutzt wird, um existierende Netzwerke zu perturbieren, oder um realistische Nullmodelle zu erzeugen.

Die in Kapitel 4 und 5 diskutierten Ansätze sind auf Markow-Ketten basierende Monte-Carlo Verfahren (MCMC). Obwohl einige in der Praxis vielfach Verwendung finden, sind uns keine allgemeinen und praktisch relevanten analytischen Ergebnisse bekannt, welche die Zeit bis zur Erzeugung einer (nahezu) uniformen Stichprobe beschränken. Wir analysieren daher die Komplexität unserer Ansätze in Abhängigkeit von der Anzahl ausgeführter Markow-Kette Schritte. Es werden folgende Ansätze betrachtet:

1. *EM-HH mit EM-ES*: Hierbei handelt es sich um eine I/O-effiziente Variante des gängigen *Fixed-Degree-Sequence-Model*. *FDSM* überführt zunächst eine gegebene Gradsequenz deterministisch in einen Graph. Um eine uniforme Stichprobe zu erhalten, wird der Graph im Anschluss mittels *Edge Switching (ES)* perturbiert [246]. Den deterministischen Schritt implementieren wird mittels unserer I/O-effizienten Variante EM-HH des Havel-Hakimi-Generators [173, 165]. Hierzu entwickeln wir eine Datenstruktur, die eine sortierte Gradsequenz durch Gruppierung identischer Grade komprimiert. Die Verschmelzung von Graden reduziert sowohl den Speicherverbrauch als auch die Anzahl der Zugriffe auf die Datenstruktur.

Da EM-HH als Teil einer Pipeline konzipiert ist, die Ein- und Ausgabe ohne Zwischenschritt über den externen Speicher verarbeitet, analysieren wir nur die I/Os, die EM-HH intern ausführt.² Wir zeigen, dass EM-HH I/O-optimal arbeitet, und dass der Algorithmus Graphen mit $\mathcal{O}(M^{2\gamma})$ Kanten mit hoher Wahrscheinlichkeit ohne I/Os erzeugen kann, falls es sich bei der Eingabe um eine monotone Sequenz aus einer Powerlaw-Verteilung mit Exponent γ handelt.

Dann übernimmt *Edge Switching* die Randomisierung des erzeugten Graphen. In jedem Schritt werden zwei unterschiedliche Kanten uniform zufällig gewählt und ihre Endpunkte vertauscht. Wenn ein Tausch eine Mehrfachkante oder Eigenschleife erzeugen würde, ist es notwendig diesen ersatzlos zu streichen [83].

²Wenn die Ausgabe für eine andere Anwendung in den externen Speicher geschrieben werden müsste, würde der hierfür notwendige Scan die I/O Komplexität des Algorithmus dominieren.

EM-LFR:

 Kapitel 4

Übersicht zu

Problem und Ansätzen:

 Abschnitt 2.6

Beispiel Nullmodell:

 Abschnitt 1.2.7.1

EM-HH:

 Abschnitt 4.4

EM-ES:

 [Abschnitt 4.5](#)

Unsere I/O-effiziente Variante EM-ES nutzt Stapelverarbeitung, wobei jedes Batch $\Theta(m)$ Tausche ausführt. Durch geeignetes An- und Umordnen der Tausche und Kanten kann eine I/O-Komplexität von $\mathcal{O}(\text{sort}(n + m))$ erreicht werden (im Vergleich zu $\Theta(m)$ I/Os der ursprünglichen Formulierung). Während der Stapelverarbeitung müssen Abhängigkeiten innerhalb des Batch beachtet werden. So können etwa frühe Operationen Kanten erzeugen oder löschen, die von einem späteren Tausch bearbeitet werden. Dies lässt sich durch mehrere Scan- und Sortierphasen sowie dem Einsatz von *Time Forward Processing* lösen.

EM-CM/ES:

 [Abschnitt 4.6](#)

2. **EM-CM/ES:** Das zuvor geschilderte Verfahren nutzt EM-HH, um einen deterministischen simplen Graphen zu erzeugen und diesen dann aufwendig zu randomisieren. EM-CM/ES startet stattdessen mit einem zufälligen nicht-simplen Graphen und überführt diesen in einen simplen Graphen.

Configuration Model:

 [Abschnitt 4.6.1](#)

Die Startinstanz generieren wir mittels einer I/O-effizienten Implementierung des *Configuration Models* [38, 260]. Zudem modifizieren wir EM-ES um nicht-simple Eingaben zu unterstützen. In dieser Variante erlauben wir jeden Kantentausch, der die Anzahl von Mehrfachkanten oder Eigenschleifen nicht erhöht. Um das Entfernen dieser unerwünschten Kanten zu beschleunigen, werden diese bevorzugt als Tauschpartner gewählt.

EM-ES Modifikation:

 [Abschnitt 4.6.2](#)

Nachdem EM-CM/ES einen simplen Graphen erreicht hat, sind im Allgemeinen noch weitere reguläre *ES* Schritte notwendig. Dies ist darin begründet, dass das reine Wegtauschen von nicht-simplen Kanten der Startinstanz mittels der vorgestellten Variante von *ES* nicht zu einer uniformen Stichprobe führt (vgl. [18, 24]). Unsere empirische Untersuchung lässt jedoch den Schluss zu, dass EM-CM/ES dennoch schneller zu einer uniformen Stichprobe konvergiert als die Kombination von EM-HH und EM-ES.

Empirischer Vergleich:

 [Abschnitt 4.10.7](#)

EM-CB:

 [Abschnitt 5.4.1](#)

3. **EM-CB:** Bei *Curveball (CB)* [321] handelt es sich um einen relativ neuen MCMC Prozess, der strukturelle Ähnlichkeiten zu *ES* hat. Jedoch wählt *CB* in jedem Schritt jeweils zwei unterschiedliche Knoten $u \neq v$ zufällig aus und mischt dann deren Nachbarschaften. Hierfür werden zunächst alle Kanten isoliert, die zu gemeinsamen Nachbarn von u und v führen oder zwischen den beiden Knoten verlaufen. Die verbleibenden Nachbarn werden zufällig neu verteilt, ohne die Grade von u und v zu ändern.

Ein *CB* Tausch kann somit potenziell mehr Veränderungen herbeiführen als ein *ES* Schritt. Empirische Analysen unterstützen diese Beobachtung und lassen den Schluss zu, dass die *CB* Markow-Kette zum Mischen weniger Schritte als *ES* benötigt [81]. Hierbei bleibt jedoch zunächst unbeachtet, dass im Allgemeinen *CB* Schritte mehr Arbeit als *ES* Schritte implizieren. Wir entwickeln daher nachfolgend effiziente Algorithmen für *CB* und ergänzen die empirische Untersuchung.

Empirischer Vergleich:

 [Abschnitt 5.5](#)

Neben dem potentiell schnelleren Mischen hat *CB* auch algorithmische Vorteile. So weist *CB* mehr Datenlokalität auf, da alle benötigten Informationen in den Nachbarschaften der beteiligten Knoten zu finden sind. Dies steht im Gegensatz

zu *ES*, das pro Tausch zusätzliche unstrukturierte Existenzanfragen stellen muss um Mehrfachkanten zu verhindern.

Für ungerichtete Graphen muss jedoch beachtet werden, dass die meisten kompatiblen Datenstrukturen jede ungerichtete Kante zweimal speichern – einmal pro Endpunkte beziehungsweise Richtung. Dies negiert die zuvor genannten Lokaltätsvorteile, da getauschte Kanten an diversen Stellen innerhalb der Datenstruktur geändert werden müssen.

Unsere I/O-effiziente Lösung nutzt daher eine dynamische Datenstruktur, die jede ungerichtete Kante nur bei dem als nächstes getauschten Endpunkt speichert. Stapelverarbeitung und *Time Forward Processing* dienen als Grundlagen dieses Ansatzes. Am Beginn jedes Batches werden zunächst alle Knotenpaare, die später getauscht werden, gezogen und in einer Hilfsstruktur indiziert. Die Indizierung erlaubt es, *TFP* Nachrichten zu adressieren und festzustellen, wann genau eine Kante zukünftig wieder benötigt wird.

Wir zeigen, dass EM-CB pro Globaltausch (siehe unten) $\mathcal{O}(\text{sort}(n) + \text{sort}(m))$ I/Os benötigt. Da die Hilfsstrukturen in der Praxis einen Overhead verursachen, beschreiben wir zudem die Variante IM-CB; unsere Experimente zeigen, dass sich IM-CBs einfachere Datenstruktur trotz mehr unstrukturierter Zugriffe auf den Hauptspeicher bei kleinen und mittelgroßen Eingaben auszahlt.

IM-CB:

 [Abschnitt 5.4.2](#)

4. *EM-GCB and EM-PGCB*: Wir erweitern gerichtetes *Global Curveball* (*G-CB*) [81] auf ungerichtete Graphen. Ein ungerichteter Globaltausch ist als zufällige Sequenz von $\lfloor n/2 \rfloor$ einfachen *CB* Schritten definiert, die jeden Knoten höchstens einmal wählen. Wir zeigen, dass dieser neue Markow-Prozess zu einer gleichverteilten stationären Verteilung konvergiert. Unsere empirischen Vergleiche weisen darauf hin, dass *G-CB* schneller als eine entsprechende Anzahl von *CB* Schritten mischt.

G-CB:

 [Abschnitt 5.3.3](#)

G-CB erlaubt es uns, die zuvor beschriebene Hilfsdatenstruktur von EM-CB komplett zu entfernen. Wenn jeder Knoten exakt einmal³ getauscht wird, können wir einen Globaltausch als eine zufällige Permutation der Knoten interpretieren. In der Permutation paarweise-benachbarte Knoten entsprechen dann jeweils einem *CB* Tausch. Unser I/O-effizienter EM-GCB Algorithmus verwaltet diese Permutation nun implizit über eine zufällige, kollisionsfreie und invertierbare Hash-Funktion.


EM-GCB:

 [Abschnitt 5.4.3](#)

Mit EM-PGCB parallelisieren wir EM-GCB. Hierzu unterteilen wir einen Globaltausch in kleine Blöcke und bearbeiten die Tausche innerhalb eines Blocks parallel. Die Blockgröße ist dabei so gewählt, dass Abhängigkeiten in einem Block unwahrscheinlich sind. Seltene Problemfälle werden erkannt und mittels eines Work Stealing Ansatzes bearbeitet. In einer empirischen Analyse erreicht EM-PGCB dieselbe Qualität wie EM-ES und ist bis zu einer Größenordnung schneller.

EM-PGCB:

 [Abschnitt 5.4.4](#)

³Wir nehmen hier vereinfachend eine gerade Knotenanzahl an (allgemein  [Abschnitt 5.4.3](#)).

Graphen mit geometrischer Einbettung (Kapitel 6 bis 8)

Random Hyperbolic Graphs (RHG) [200] sind ein populäres Netzwerkmodell, das viele Eigenschaften komplexer Netzwerke auf natürliche Weise abbildet. Es ordnet jedem Knoten eine zufällige Position auf einer hyperbolischen Scheibe mit Radius R zu. Zumeist werden hierfür Polarkoordinaten genutzt. Dabei werden Positionen durch ihren Abstand r vom Ursprung (Radius) und einer Winkelkomponente θ (Azimut) beschrieben. Beim zufälligen Verteilen der Punkte wächst die radiale Dichtefunktion exponentiell mit dem Radius. Die Mehrheit der Punkte befindet sich hierdurch nahe am Scheibenrand.

Threshold RHG

In der sogenannten *Threshold RHG* [159] Variante werden alle Punktpaare mit Koordinaten $p_i=(r_i, \theta_i)$ und $p_j=(r_j, \theta_j)$ verbunden, deren nachfolgend definierte hyperbolische Distanz $d(p_i, p_j)$ kleiner als R ist:

$$\cosh(d(p_i, p_j)) = \cosh(r_i) \cosh(r_j) - \sinh(r_i) \sinh(r_j) \cos(\theta_i - \theta_j) \quad (1)$$

Die Distanz zweier Punkte hängt somit sowohl von ihrer relativen als auch absoluten Position ab. Die wenigen Punkte nahe des Ursprungs implizieren kleinere Distanzen und haben somit mehr Nachbarn als Punkte am Scheibenrand. Eine genauere Analyse zeigt eine Powerlaw-Gradverteilung, deren Exponent sich über die zuvor genannte radiale Dichtefunktion kontrollieren lässt.

Binomial RHG

Lokale Kohäsion:
 *Abschnitt 1.2.5*

Binomial RHG generalisieren *Threshold RHG*, indem sie eine positive „Temperatur“ T als zusätzlichen Parameter einführen. Die Temperatur beeinflusst die lokale Kohäsion von Knoten (Grüppchenbildung). Jedes Knotenpaar $p_i \neq p_j$ wird dann unabhängig mit Wahrscheinlichkeit $p_T(d(p_i, p_j))$ durch eine Kante verbunden:


$$p_T(d) = \left[\exp\left(\frac{d-R}{2T}\right) + 1 \right]^{-1} \quad (2)$$

Für $T \rightarrow 0$ degeneriert $p_T(d)$ zu einer gespiegelten Einheitssprungfunktion an Stelle R , wodurch *Threshold RHGs* in *Binomial RHGs* enthalten sind.

Im Folgenden stellen wir drei Ansätze zur Generierung von *RHG* vor.

 *Kapitel 6*

- *HYPERGEN* ist ein *Threshold RHG* Generator, der für Rechenbeschleuniger mit kleinem Hauptspeicher konzipiert ist. Um Speicher zu sparen, bricht der Algorithmus mit dem zuvor üblichen Paradigma, zunächst die Koordinaten aller Knoten zu ziehen und diese dann in einer geeigneten geometrischen Datenstruktur zu verwalten. Stattdessen überstreicht *HYPERGEN* die hyperbolische Scheibe mittels einer azithmutalen voranschreitenden Sweep-Line⁴ und erzeugt dabei Punktkoordinaten on-the-fly. Hierzu ziehen wir die Punktmenge mit monoton aufsteigender Winkelkomponente⁵, ohne die endgültige Koordinatenverteilung zu verändern.

HYPERGEN:
 *Abschnitt 6.3*

Der Algorithmus hält nur Knoten im Sweep-Status, die in der Nähe der Sweep-Line Nachbarn finden können, und löscht obsoletere Kandidaten zeitnah. Aufgrund

⁴Tatsächlich muss die Sweep-Line in radialer Richtung gesplittet werden, sodass eigentlich mehrere synchronisierte Instanzen des beschriebenen Algorithmus ausgeführt werden.

⁵Hierbei handelt es sich um eine stark vereinfachte Darstellung (vgl. Abschnitt 6.2).


der azithmutalen 2π -Periodizität der hyperbolischen Kreisscheibe befinden sich am Ende der Ausführung noch spät gezogene Punkte (mit einem Winkel nahe 2π) im Status und müssen potentiell mit Knoten verbunden werden, welche früh gelöscht wurden (mit positivem Winkel nahe 0). Wir starten daher die Sweep-Line und das Ziehen der Punkte neu und verarbeiten die verbliebenen Kandidaten. Pseudozufallsgeneratoren mit identischer Startkonfiguration produzieren dabei im ersten und zweiten Scan konsistente Koordinaten.

Wir zeigen, dass der Speicherverbrauch von HYPERGEN mit hoher Wahrscheinlichkeit durch $\mathcal{O}([n^{1-\alpha}\bar{d}^\alpha + \log n] \log n)$ Worte beschränkt ist, wobei $\alpha > 1/2$ ein Modellparameter und \bar{d} der Durchschnittsgrad der Ausgabe ist. Für realistische Werte von $\bar{d} = o(n/\log^{1/\alpha}(n))$ stellt dies eine erhebliche Verbesserung gegenüber früheren Ansätzen mit einem Speicherbedarf von $\Omega(n)$ Worten dar.

HYPERGEN arbeitet parallel und teilt hierzu die hyperbolische Scheibe in Segmente. Auf jedem Segment wird der oben skizzierte Algorithmus unabhängig ausgeführt. Hierbei müssen einige Vorarbeiten unternommen werden, um überbordende Abhängigkeiten an den Segmentgrenzen zu verhindern.

Parallelisierung:
 [Abschnitt 6.3.2](#)

Eine weitere wichtige Optimierung entfernt alle transzendenten Funktionen (hier \sinh , \cosh und \cos) der häufig zu evaluierenden Gleichung (1) und ersetzt diese durch Werte, die pro Punkt vorberechnet werden. Weiterentwicklungen dieses Ansatzes finden auch in sRHG und HYPERGIRGS (siehe unten) Anwendung.

Vorbereitung:
 [Abschnitt 6.4.1](#)

Unsere Implementierung von HYPERGEN nutzt explizit datenparallele Vektoreinheiten moderner Prozessoren mittels geeigneter *Single-Instruction-Multiple-Data* [129] Befehlssatzerweiterungen [184]. Unter anderem können somit bis zu acht hyperbolische Distanzen gleichzeitig berechnet werden; dies wird erst durch die zuvor beschriebenen Vermeidung transzendenter Funktionen möglich.

Datenparallelismus:
 [Abschnitt 1.3.4](#)

Wir vergleichen HYPERGEN experimentell zu vier anderen state-of-the-art Generatoren und finden, dass HYPERGEN teils 30 mal schneller als die Mitbewerber ist. Auf einem günstigen Standardcomputer produziert HYPERGEN Graphen mit $10^6 \leq m \leq 10^{12}$ Kanten mit einem Durchsatz von knapp 370 Million Kanten pro Sekunde und einem Speicherverbrauch von weniger als 600 MB. Zudem evaluieren wir HYPERGEN auf einem hochgradig parallelen Rechenbeschleuniger.

- *sRHG* kombiniert HYPERGEN mit RHG [139], einem kommunikationsfreien *Threshold RHG* Generator für verteilte Maschinen. HYPERGEN und RHG wurden nahezu zeitgleich, jedoch unabhängig, entwickelt und für unterschiedliche Maschinenmodelle entworfen. Beide basieren auf Pseudorandomisierung in unterschiedlich starker Ausprägung. In einer experimentellen Auswertung demonstrieren wir die Skalierbarkeit von sRHG auf 32 768 Prozesskernen und generieren einen Graphen mit $n = 2^{39}$ Knoten in deutlich weniger als einer Minute.

 [Kapitel 7](#)

sRHG:
 [Abschnitt 7.7.3](#)

- *HYPERGIRGS* und *GIRGS* gehen auf einen zuvor bekannten Algorithmus von Bringmann et al. [70] mit erwarteter linearer Laufzeit zurück. Obwohl einige der im

Best Paper Award · ESA'19
 [Kapitel 8](#)

Vergleich RHG & GIRG:
📖 Abschnitt 8.2.3

Datenstruktur:
📖 Abschnitt 8.3

Beschleunigungstechnik:
📖 Abschnitt 2.3.3.4 ff.

Restrukturierung:
📖 Abschnitt 8.3.3

Folgenden skizzierten Aspekte bereits in [70] beschrieben wurden, bedurfte es erheblicher Anpassungen um den (nach besten Wissen) ersten praktisch effizienten Generator für *Geometric Inhomogenous Random Graphs (GIRG)* zu implementieren.

Mit dem state-of-the-art Generator HYPERGIRGS stellen wir eine Modifikation von GIRGS vor, die *Binomial RHG* exakt erzeugen kann.⁶ In einer ersten Phase, zieht der Algorithmus die Koordinaten aller Knoten und speichert die Punkte in einer Datenstruktur, die als polarer Quad-Tree verstanden werden kann und nachfolgend so bezeichnet wird.

Um nun die Nachbarschaften zu berechnen, werden geeignete Paare von Quad-Tree-Knoten identifiziert. Für jedes dieser Paare werden konzeptionell alle Punkte, die im ersten Quad-Tree-Knoten enthalten sind, mit allen Punkten des zweiten Quad-Tree-Knotens verglichen. Der Prozess wird bei Kanten mit geringer Kantenwahrscheinlichkeit durch eine Kombination von geometrischen Sprüngen und der Verwerfungsmethode beschleunigt. Zudem entwickeln wir eine exakte Heuristik, die in den meisten Fällen eine teure Evaluation von $p_T(d(p_i, p_j))$ vermeidet.

Der Quad-Tree muss den wahlfreien Zugriff auf alle Punkte, die von einem beliebigen Quad-Tree-Knoten (in beliebiger Tiefe) repräsentiert werden, in konstanter Zeit unterstützen. HYPERGIRGS erzielt dies, indem der Quad-Tree mittels einer raumfüllenden Lebesgue-Kurve [268] im Speicher linearisiert wird. Durch die Verwendung sogenannter Morton-Codes [251] lässt sich die Datenstruktur dann effizient konstruieren und adressieren.

GIRGS ist strukturell ähnlich zu HYPERGIRGS. Unsere Implementierung unterstützt bis zu fünf-dimensionale *GIRGs*, wodurch eine Anpassung der bit-parallelen Morton-Code Implementierung nötig wird. Sowohl GIRGS als auch HYPERGIRGS können nach einer Restrukturierung des Algorithmus von Bringmann et al. parallelisiert werden. Als Besonderheit liefern unsere Implementierungen reproduzierbare Ergebnisse in dem Sinne, dass zwei Ausführungen mit identischen Parametern und Startkonfigurationen der Pseudozufallsgeneratoren dieselbe Kantenmenge berechnen (auch wenn die Kantenreihenfolge abweichen kann).

⁶Bringmann et al. stellen bereits den Bezug zwischen *GIRGs* und *RHG* her indem sie eine asymptotische Inklusion zeigen. Wie wir in Abschnitt 8.5.3 jedoch empirisch nachweisen, sind für einen exakten *RHG* Generator weitere Modifikationen notwendig.

Contents

1	Introduction	1
1.1	Motivation	2
1.2	Random Graphs in the Context of Network Analysis	3
1.3	Practical Engineering Challenges	14
1.4	Articles Included in the Present Thesis	21
2	Recent Advances in Scalable Network Generation	27
2.1	Introduction	28
2.2	Graph Properties and Uses of Generators	28
2.3	General Algorithmic Models and Techniques	32
2.4	Basic Models	36
2.5	Random Spatial Graphs	39
2.6	Random Graphs with Prescribed Degree Sequences	45
2.7	Block Models	50
2.8	Graph Replication	53
2.9	Additional Graph Types	58
2.10	Software Packages	63
2.11	Future Challenges	63
3	Preferential Attachment	67
3.1	Introduction	68
3.2	The Sequential TFP-BA Algorithm for EM	70
3.3	The Parallel MP-BA Algorithm for EM	73
3.4	Implementation of MP-BA	79
3.5	Experimental Results	81
3.6	Preferential Attachment beyond BA	84
4	Massive Graphs Following the LFR Benchmark	89
4.1	Introduction	90
4.2	Preliminaries and Notation	92
4.3	The LFR Benchmark	94
4.4	EM-HH: Deterministic Edges from a Degree Sequence	95
4.5	EM-ES: I/O-efficient Edge Switching	100
4.6	EM-CM/ES: Sampling of Graphs from Degree Sequence	105
4.7	EM-CA: Community Assignment	107
4.8	EM-GER/EM-CER: Merging Intra- and Inter-Community Graphs	110
4.9	Implementation	111
4.10	Experimental Results	112
4.11	Outlook and Conclusion	120
	Appendix	122

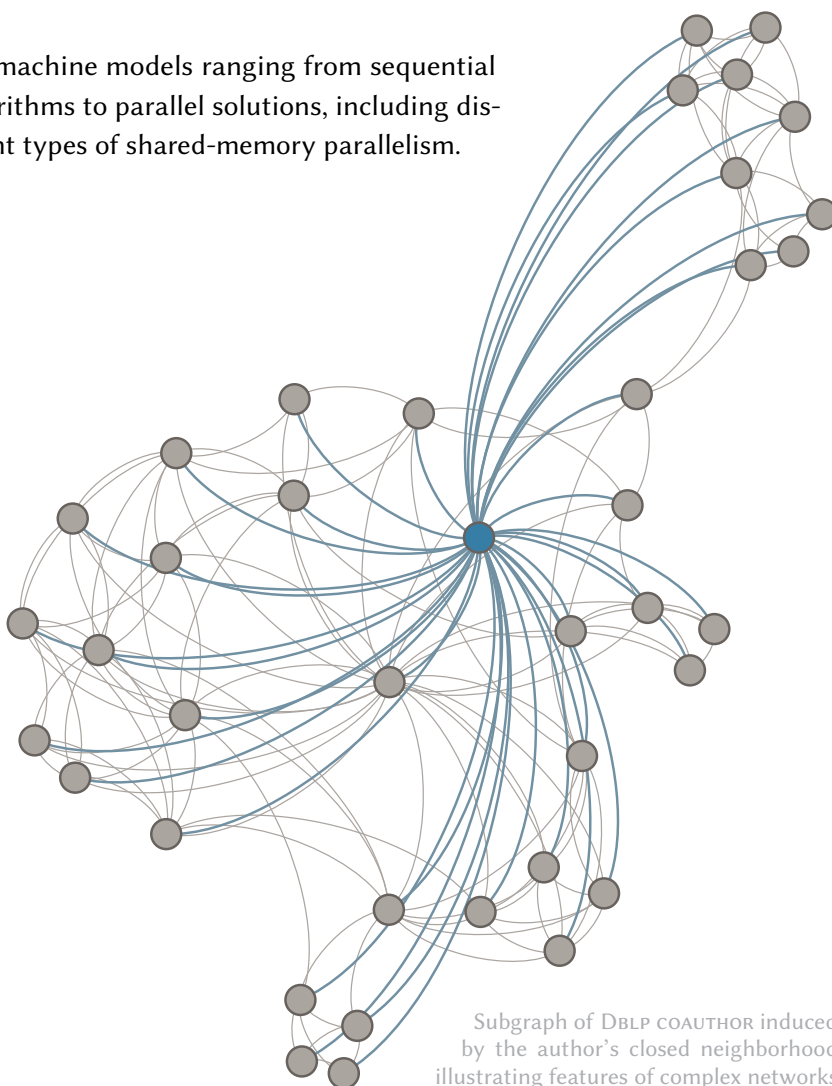
5	Global Curveball	129
5.1	Introduction	130
5.2	Preliminaries and Notation	131
5.3	Randomization Schemes	133
5.4	Novel Curveball Algorithms for Undirected Graphs	135
5.5	Experimental Evaluation	141
5.6	Conclusion and Outlook	144
	Appendix	145
6	Streaming Random Hyperbolic Graphs	159
6.1	Introduction	160
6.2	MemGen: a Fast Algorithm with Linear Memory Usage	164
6.3	HyperGen: Reducing MemGen’s Memory Footprint	169
6.4	Implementation	172
6.5	Experimental Evaluation	173
	Appendix	177
7	Communication-free Graph Generation	183
7.1	Introduction	184
7.2	Preliminaries	185
7.3	Related Work	188
7.4	<i>ER</i> Generator	190
7.5	<i>RGG</i> Generator	193
7.6	<i>RDT</i> Generator	196
7.7	<i>RHG</i> Generators	197
7.8	Experimental Evaluation	208
7.9	Conclusion	215
	Appendix	217
8	Geometric Inhomogeneous and Hyperbolic Random Graphs	221
8.1	Introduction	222
8.2	Models	225
8.3	Sampling Algorithm	226
8.4	Implementation Details	229
8.5	Experimental Evaluation	231
	Appendix	234
9	Summary	243
9.1	Preferential Attachment	244
9.2	Simple Graphs from Prescribed Degree Sequence	245
9.3	Geometrically Embedded Random Graphs	247
9.4	Future Research Opportunities	251

Introduction

Network models play a crucial role in various fields of science and their applications far surpass the original scope of explaining features observed in the real world. A common use case of such random graphs is to provide a versatile and controllable source for synthetic data to be used in experimental campaigns. As such, they can provide valuable insights during the design and evaluation of algorithms and data structures — in particular, in the context of large problem instances. Generating such graphs at scale, however, is a non-trivial task in itself.

The present thesis considers algorithmic aspects of generating massive random graphs. To this end, we develop practically efficient sampling algorithms for a number of established random graph models including preferential attachment networks, graphs with prescribed degree sequences, and models based on geometric embeddings. All models are applicable, but not limited, to social networks.

Our generators target various machine models ranging from sequential schemes over I/O-efficient algorithms to parallel solutions, including distributed computing and different types of shared-memory parallelism.



Subgraph of DBLP COAUTHOR induced by the author's closed neighborhood illustrating features of complex networks

1.1 Motivation

networks and graphs:
📖 Section 1.2.1

Networks are the very fabric that makes societies [34, 261]. As such, humanity is seeking to understand their structures, rules and implications for centuries. The practical importance of networks, however, only sky-rocketed with the advent of the information age. Nowadays, modern computers offer sufficient storage and processing capacity to map out most aspects of human life and the world we inhabit. They are fed by billions of interconnected sensors and computerized personal devices that produce enormous volumes of network data to be exploited.

Computer science provides the means to face this *big data challenge*. However, a formal language capturing the inner structure of the data expected to be processed is required to provide tailor-made solutions. Network models are just that: a mathematical tool to describe and analyze realistic graphs. Research into and applications of these models are deeply intertwined with various fields of science.

random graphs:
📖 Section 1.2.3

Networks are commonly modeled by so-called *random graphs* and, therefore, represent probability distributions over the set of graphs [59]. These distributions are almost always parametrized (e.g., for the graph size or density) and typically follow implicitly from some randomized construction algorithm. Popular models are designed such that we can expect certain topological properties from a randomly drawn instance. A particularly interesting goal is to reproduce the loosely defined class of *complex networks* which, among others, encompasses most social networks.

complex networks:
📖 Section 1.2.4 ff.

By expressing network models as random graphs, we inherit a rich set of tools from combinatorics, stochastics and graph theory. In algorithmics we may, for instance, assume that meaningful inputs are random instances of a *suitable* network model. Then we can derive realistic formal performance predictions using average-case analysis, smoothed complexity, et cetera. In practice, such results tend to be more relevant than worst-case analysis based on pathologic structures that are implausible in applications.

survey of network models:
📖 Chapter 2


Network models also enable or supplement experimental campaigns as a versatile source of synthetic data with controllable independent variables. Synthetic benchmarks are especially useful in the context of large instances where real data is typically unavailable in sufficient size, quantity, or variety. Even if the data exists, procuring and archiving it may be difficult for legal or technical reasons; this threatens the independent reproducibility of results and thus infringes on one of science's cornerstones [273].

1.1.1 Goals

engineering challenges:
📖 Section 1.3

We study algorithmic aspects of the generation of random graphs at scale. While we acknowledge the merits of developing or adapting network models dedicated to fast generation (e.g., [95, 311]), each new model used further fragments the body of empirical studies into domains of incomparable results. Hence, we mostly focus on established graph families for which we develop faithful sampling algorithms that are analytically efficient and practically fast. The considered models include preferential attachment networks, graphs with prescribed degree sequences, and geometrically embedded networks; they capture social networks and are applicable beyond.

Our generators target various machine models ranging from classical sequential schemes over I/O-efficient algorithms to parallel solutions, including distributed computing and different types of shared-memory parallelism.

summary of results:
 [Chapter 9](#)

1.1.2 Outline

The present thesis is organized as follows:

- Section 1.2 discusses features commonly associated with complex networks.
- Section 1.3 introduces *algorithm engineering* as our design methodology. It also summarizes challenges encountered during the development of generators that are efficient in both theory and practice. Most of the discussion pertains to features of modern computers which we capture with advanced models of computation.
- Chapters 2 to 8 present the main part of the present thesis and contain its original work. A short overview is given in Section 1.4.

Each chapter is based on at least one peer-reviewed publication (with the exception of Chapter 2, which is based on a submitted manuscript). The articles' content is reproduced as published and subject to only moderate copy editing ensuring that all chapters adhere to similar formatting, typography, notation, and naming conventions. Remarks and annotations are limited to the pages' outer columns.

Each chapter is prefixed with a title page stating the abstract and co-authors. It also declares the contributions of the present thesis' author.

- Chapter 9 summarizes the results obtained in the main part of the thesis.

1.2 Random Graphs in the Context of Network Analysis

In the following section, we introduce important features of complex networks. The discussion is presented as an explorative and non-exhaustive tutorial focusing on aspects relevant to the generators developed in the main section of the thesis. This hands-on approach is contrasted by Chapter 2 which offers a broader and classical survey.

1.2.1 From Networks to Graphs

We use the term *network* to refer to some process involving connected entities, and use a *graph* as a formal concept to capture “relevant” aspects observed in such a network. A graph has a simple basic structure: it consists of *nodes* (also known as *vertices* or *points*), and *edges* (also known as *arcs* or *links*). Each edge connects at least two –not necessarily different– nodes, and can have a direction, weight, and label.

nodes, vertices, edges

The application or problem at hand determines the set of *relevant* aspects that need to be modeled. It, thereby, guides the process of translating a network into a graph. By design, this translation tends to be lossy since one often chooses to disregard features considered to be unimportant.

a single network can imply multiple application-specific graph models

Assume, for instance, that we want to capture the essence of a scientific collaboration network. We can obtain a graph by identifying authors with nodes and adding an edge for each pair of authors that collaborated on at least one article. Such graphs are referred to as *coauthorship* networks [204]. While this simple model suffices to identify clusters of co-authors and similar structural aspects, it omits the collaborations' contexts.

Thus, we might want to modify the linkage criterion: for each article, we add a single edge between all authors' nodes. Such an edge connects an arbitrary number of nodes and is referred to as a *hyper-edge* and is part of a *hyper-graph*. Since hyper-graphs are non-trivial to process, they are commonly translated into bipartite graphs by identifying each hyper-edge with a newly added vertex. In this concrete example, we extend the set of nodes by adding a vertex for each article and only connect authors and articles. The term *bipartite* describes the property that the set of nodes can be split into two classes (here, authors and articles) such that no edge connects two nodes of the same class.

Case Study 1.1. Throughout the introduction, we regularly present a case study based on a coauthorship network. It relies on empirical data extracted from the DBLP database [298] containing more than 5 million publications in computer science. DBLP was selected for technical reasons as it provides freely accessible and curated real-world data; for the purpose of this introduction it is, however, just one of many networks with similar structural properties. We extracted the coauthorship graph DBLP COAUTHOR, and the aforementioned bipartite modeling DBLP BIPARTITE. For simplicity, we only consider their largest connected components encompassing more than 89% of authors. ◀

1.2.2 Graph Theory

In the following, we formalize the previously rather prose discussion of graphs. Graph theory offers versatile tools to model, analyze and process objects and their relationships. Its first usage in a mathematical proof is commonly attributed to Leonhard Euler [125] who gave a negative resolution to the *Seven Brigdes of Königsberg* problem in the year 1741. Despite this potentially devastating set-back for the city's tourism industry, graphs have become a virtually ubiquitous concept in modern science.

Formally, a graph $G = (V, E)$ is a tuple where $V = [v_i]_{i=1}^n$ represents the set of nodes and $E = [e_i]_{i=1}^m$ the set of edges. If not stated differently, we denote the number of nodes and edges with n and m , respectively. In the interest of brevity, we restrict ourselves to the most common case where any edge e connects exactly two nodes u and v . Then, we say that the two nodes are *adjacent* to each other and *incident* to edge e .

An asymmetric relationship (e.g., person u sends a letter to person v) is modeled by a *directed edge* and denoted as a tuple $e_i = (u, v)$. This implies $E \subseteq V \times V$ and we say G is a *directed graph*, or short, *digraph*. On the other hand, an edge in an *undirected graph* expresses a symmetric relationship, and we have $E \subseteq \{\{u, v\} \mid u, v \in V \wedge u \neq v\}$. Digraphs are strictly more expressive than their undirected counterparts: any undirected edge $\{u, v\}$ can be translated into its two directed pendants (u, v) and (v, u) , while the opposite is not always possible without loss of specificity.

hyper-graph
hyper-edge

bipartite


DBLP COAUTHOR
DBLP BIPARTITE

G, V, E, n, m

adjacent, incident

directed graph (digraph)
undirected graph

Nodes and edges can be augmented with additional information. Such labels are application-specific and may encode distances, costs, capacities, or categorical data (e.g., road types in a street network). The treatment of labels is therefore often tailor-made for specific problems and orthogonal to the graph’s topology. Thus, we focus on general structural aspects of graphs and seldom consider labels.

special graph classes:
 [Section 2.9.2](#)

1.2.3 Random Graphs


A *random graph* is a probability distribution $P: \mathbb{G} \rightarrow [0, 1]$ where \mathbb{G} is the set of all graphs. Virtually all *random graph models*¹ are parameterized and thus form families of probability distributions. The underlying distributions are typically specified implicitly, and often have a finite support defined by some combinatorial constraints.

random graph (model)

As an example, consider the famous $\mathcal{G}(n, p)$ model introduced by E. Gilbert [148] in 1959. In its original formulation, the model’s support consists of all $2^{n(n+1)/2}$ undirected graphs with exactly n nodes. The probability distribution is given indirectly via the following sampling algorithm:


$\mathcal{G}(n, p)$

“Pick one of these graphs by the following random process. For all pairs of points [*nodes*] make random choices, independent of each other, whether or not to join the points of the pair by a line [*edge*]. Let the common probability of joining be p .” [148]

efficient $\mathcal{G}(n, p)$ sampling:
 [Section 2.4.1](#)


In other words, in a random instance of $G(n, p)$ any edge e exists independently with probability p . Observe that $G(n, 1/2)$ hence implies the uniform distribution of all graphs with n nodes. It is therefore a so-called *maximum entropy model* and sometimes even referred to as *the* random graph [34].

Intuitively, the $\mathcal{G}(n, p)$ model should have little structural resemblance to complex networks. Since all edges are chosen independently with identical probabilities, we do not expect the formation of any complex features. In the following, we strive to formalize this intuition by introducing a number of features that are observable in real data and directly related to the original work in Chapters 3 to 8.

broader survey:
 [Section 2.1](#)

1.2.4 Density

Besides the previously considered number n of nodes, the number m of edges is an obvious key figure. The relationship of n and m is of particular interest and leads to the notion of *density*. In algorithmics, one typically focuses on the scaling behavior of m as a function of n . We call a family of graphs *sparse* if $m = \mathcal{O}(n \text{ poly } \log n)$, and *dense* if $m = \Theta(n^2)$. Many observed networks are sparse since the creation or maintenance of an edge requires some form of work or cost.

details on density:
 [Section 2.2.1](#)

sparse
dense

¹In the literature the terms *random graph* and *random graph model* are commonly used interchangeably, and may even refer to a random instance sampled from a model. We adopt the former simplification for the sake of readability.

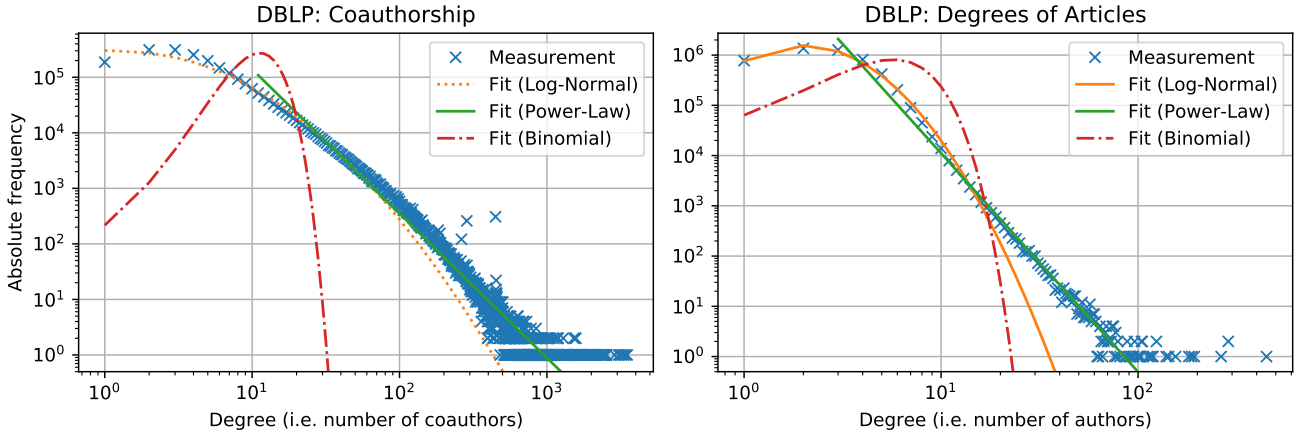


Figure 1.1: Degree frequency of DBLP COAUTHOR (left) and DBLP BIPARTITE (right) compared to common distributions.

In contrast to the global parameter m , we now consider a more fine-grained and local measure. The *degree* $\deg(u)$ of node u is defined as the number of neighbors adjacent to u . In case of digraphs, we distinguish between the out-degree $\deg^{\text{out}}(u)$ defined as the number of edges starting in u , and the in-degree $\deg^{\text{in}}(u)$ counting the number edges ending in u .

Observe that we can interpret the aforementioned relationship of n and m as the average degree $\text{avg deg}(G)$. It is another common way to measure the density of G :

$$\begin{aligned} \text{avg deg}(G) &:= \frac{1}{n} \begin{cases} \sum_{i=1}^n \deg(v_i) & \text{if } G \text{ is undirected} \\ \sum_{i=1}^n (\deg^{\text{in}}(v_i) + \deg^{\text{out}}(v_i)) & \text{otherwise} \end{cases} \\ &= 2m/n \end{aligned}$$

\mathcal{D}_G : *degree sequence*

The degree sequence \mathcal{D}_G of an undirected graph G is defined as $\mathcal{D}_G := [\deg(v_i)]_{i=1}^n$, and analogous extensions are available for digraphs. In the context of random graphs, it is often more appropriate to consider a degree distribution $f: \{0, 1, \dots, n-1\} \rightarrow [0, 1]$ rather than a concrete degree sequence. To this end, the distribution $f(k)$ gives the probability that a randomly selected node u in a randomly chosen graph G has degree k .

degree distribution

1.2.4.1 Gilbert's Graph does not Explain Complex Networks

These simple density measures suffice to demonstrate that $\mathcal{G}(n, p)$ does not accurately model observed networks. To do so, we first study the behavior of $\mathcal{G}(n, p)$ analytically and then compare it to the empirical findings in DBLP graphs.

An undirected instance uniformly sampled from $\mathcal{G}(n, p)$ contains each of its $\binom{n}{2}$ edges independently with probability p . Thus, the total number of edges m follows the binomial distribution $\text{BIND} \left[\binom{n}{2}, p \right]$:

$$m \sim \text{BIND} \left[\binom{n}{2}, p \right] = \text{BIND} [n(n-1)/2, p] \quad (1.1)$$

$$\Rightarrow \mathbb{E}[m] = \binom{n}{2} p = n(n-1)p/2 \quad (1.2)$$

The analysis of the degree distribution is similar and uses the observation that an arbitrary node u is potentially incident to $n-1$ edges that each exist independently with probability p . Thus we have:

$$\text{deg}(u) \sim \text{BIND}[n-1, p] \quad (1.3)$$

$$\mathbb{E}[\text{deg}(u)] = (n-1)p \quad (1.4)$$

$$\text{Var}[\text{deg}(u)] = (1-p)np \quad (1.5)$$

Based on Eq. (1.2), we can replicate the density of any supplied graph with n nodes and m edges by choosing

$$p = \frac{2m}{n(n-1)}. \quad (1.6)$$

Observe that the predicted degrees of a sparse $\mathcal{G}(n, p)$ graph are highly concentrated around their mean: for sparse graphs, Eq. (1.6) implies that $p = \mathcal{O}(\text{poly log}(n)/n)$ is small. Then, Eq. (1.5) bounds the variance in node degrees to $\text{Var}[\text{deg}(u)] = \mathcal{O}(\text{poly log } n)$. We refer to concentrated degree distributions as *homogenous*.

homogenous degrees

Case Study 1.2. The comparison in Figure 1.1 suggests that homogenous degrees are unrealistic for complex networks – certainly for our case study which exhibits highly heterogeneous degrees. Considering the size and density of DBLP COAUTHOR, the $\mathcal{G}(n, p)$ model predicts that most authors have 12 coauthors and suggests that no individual accumulates more than approximately 30 coauthors. Instead, we find a mode of three coauthors and a few researchers who published with more than 3000 colleagues. Both observations are virtually impossible in the $\mathcal{G}(n, p)$ model; the probability to obtain a graph with at least one node with degree above 3000 is less than $10^{-5.900}$.

complex networks have heterogeneous degrees

Some of the high-degree nodes are arguably explainable outliers, e.g., the four articles with the most authors in DBLP BIPARTITE are extraordinary interdisciplinary publications in biology and physics [98, 327, 1, 2]. Still, the overall picture is consistent with most complex networks whose degrees typically show a sub-exponential decay and are referred to as *heavy-tailed*.

heavy-tailed distribution

There are various well known probability distributions featuring heavy-tails. In the context of degree distributions, two prominent contenders are scale-free and log-normal distributed networks. Historically, scale-free networks received most attention at least partially due to the seminal work by Albert and Barabási [32, 14].

While there are competing or informal definitions of the scale-free property (e.g., [14, 218, 352], cf. [74, 96]), most agree on a powerlaw degree distribution with various constraints (e.g., on the powerlaw exponent) or relaxations. Relaxations typically focus on the distributions' tails as the performance of algorithms and dynamics on scale-free networks is largely dominated by the tail's highly connected vertices.

powerlaw:
 [Section 1.2.4.2](#)

However, by doing so one potentially disregards the majority of nodes. This effect can be observed in Figure 1.1 presenting a powerlaw estimation for DBLP COAUTHOR based on a powerlaw fitting method by Alstott et al. [19]. The fit does not accurately describe the frequencies of nodes with a degree of at most 10, which, however, make up 75.5% of the authors. More recently, alternative explanations such as the *log-normal* distribution gained momentum (see e.g., [74] for a discussion).

In this introduction, we focus on powerlaw distributions since they motivate the random graph models considered in the following chapters.

1.2.4.2 Scale-Free Networks and Powerlaw Degree Sequences

In this section, we formalize the notion of scale-free networks. To this end, we define powerlaw distributions, discuss their connections to scale-free networks, and introduce scale-free random graphs models.

Powerlaw Distributions

powerlaw distribution

An integer random variable X follows a powerlaw distribution if the probability $\mathbb{P}[X = k]$ to draw the value k scales proportionally to $k^{-\gamma}$ where $\gamma > 1$ is the so-called *powerlaw exponent*. The powerlaw exponent controls the decay of the distribution's tail and higher values of γ produce smaller tails. Newman [263] derives the expectation value of the maximal degree $\mathbb{E}[\text{maxdeg}]$ and the k -th moment $\langle X^k \rangle$ as follows:²

powerlaw exponent

$$\mathbb{E}[\text{maxdeg}] \propto n^{1/(\gamma-1)} \quad (1.7)$$

$$\langle X^k \rangle = \frac{\gamma - 1}{\gamma - 1 - k} \quad \text{for } k > \gamma - 1 \quad (1.8)$$

Powerlaw Distributions are Scale-Free

scale-free

These basic definitions allow us to motivate the term *scale-free*. A distribution is called scale-free if there exists a $k \geq 1$ such that the k -th moment of the distribution diverges, i.e., it grows beyond any *scale* (see [34, Sec. 4.4] for details). In case of powerlaw distributions, this occurs [263] for all moments with $k > \gamma - 1$.

As a corollary, we find that the average degree (i.e., the first moment) in the limit of $n \rightarrow \infty$ is only finite if $\gamma > 2$. Conversely, the common scenario of $\gamma \leq 3$ implies a divergent variance (second moment). Equation (1.7) gives an intuition for this: in the domain of $\gamma < 3$, the maximal degree grows as $\omega(\sqrt{n})$, translating into a super-constant contribution to the distribution's variance. This skewness in the degree distribution can have significant effects on the performance of algorithms (e.g., due to load balancing issues), and is hence worth exploring using random graphs.

Preferential Attachment Yields Powerlaw Degrees

Barabási and Albert [32] propose a simple stochastic process to explain the emergence of scale-free networks and show that two ingredients, namely growth and selection bias, suffice to yield networks with powerlaw degree distributions.³


²The derivation uses a continuous variant of the distribution and matches in the limit of $n \rightarrow \infty$. This formulation is commonly encountered in the analysis of powerlaw models, e.g., in the early mean-field derivations of the soon to be discussed *BA* model [33].

³Earlier, Price [275] proposed a similar process inspired by Pólya urns [117]. The author applies it to citation networks with a known powerlaw in-degree distribution [276]. The *BA* model is sometimes interpreted as a special case of Price's model, and was selected here due to its wide spread and simplicity.

At its core, their *BA* model relies on *preferential attachment*, a positive feedback loop in dynamic systems where selecting an item at one point in time increases the probability of selecting it again in the future. It is often proverbially summarized as “the rich get richer”. Based on this idea, the authors describe the following random graph:

preferential attachment

“[...] starting with a small number (m_0) of vertices, at every time step we add a new vertex with $d < m_0$ edges that link the new vertex to d different vertices already present in the system. To incorporate preferential attachment, we assume that the probability Π that a new vertex will be connected to vertex i depends on the connectivity [*degree*] k_i of that vertex, so that $\Pi(k_i) = k_i / \sum_j k_j$. After t time steps, the model leads to a random network with $t + m_0$ vertices and $d \cdot t$ edges.” [32] [*Initial degree renamed to d*]

efficient BA generators:
 *Chapter 3*
 and *Section 2.4.2*

The *BA* model is simplistic and rigid since it is specifically designed to explain the emergence of powerlaw degree distributions. Besides the unspecified seed graph of size m_0 , its only parameters are the number t of added nodes as well as their initial degrees d . The parameters control little more than the graph size and its density.

In the following, we highlight some features of observed networks that are not adequately reproduced when using the *BA* model as a graph generator. The first one pertains to the powerlaw distribution since its exponent cannot be controlled and has a fixed expectation value $\mathbb{E}[\gamma] = 3$. Additionally, most *BA* graphs are connected. A connected *seed graph* implies that any graph emitted by the model is connected as well. Even if the seed graph is disconnected, the output will be connected with high probability for $d > 1$ and sufficiently large t . The *BA* model also lacks any notion of clustering, a prominent feature of complex networks. There exist however extensions to increase clustering (e.g., [177, 110]).

clustering:
 *Section 1.2.5*

Observe that many features of the seed graph will carry over to the output if the number of new vertices t is comparably small. Since the model includes growth, it can be used to benchmark dynamic algorithms and data structures by providing a reasonable sequence of node and edge insertions. In Chapter 3, we therefore propose and analyze two generators for the *BA* model that support massive seed graphs exceeding the size of main memory.

1.2.5 Clustering

Complex networks exhibit fluctuations of their density at different scales ranging from small local effects to global structures. These effects are mostly decoupled from the degree distributions discussed in the previous section.

1.2.5.1 Local Clustering

At the smallest level, these fluctuations can be observed via certain induced substructures. These so-called *motifs* [245, 304] often appear with a significantly elevated frequency (see below). While we will quantify such effects using *clustering coefficients* in Section 2.2.1, it suffices for the moment to consider induced closed triangles K_3 .

motif

Case Study 1.3. Consider a researcher v_a with at least two neighbors in the DBLP COAUTHOR network and select two of her coauthors $v_b \neq v_c$. The three nodes form a so-called *2-path*. Then we may intuit from our everyday knowledge about collaborations that it is quite likely that the three authors not only induce a 2-path but actually form a triangle. In fact, 17.4 % of all 2-paths in DBLP COAUTHOR are within a closed triangle.⁴

2-path

The probability of an edge existing between v_b and v_c conditioned on the existence of a common neighbor v_a is therefore $3.4 \cdot 10^5$ times higher than expected from independent edges as in the $\mathcal{G}(n, p)$ model. Note that $\mathcal{G}(n, p)$ is used here for ease of exposition even if it is certainly the wrong reference point due to its vastly different degree distribution which affects the occurrence rate of motifs.

degrees of $\mathcal{G}(n, p)$:
 Section 1.2.4.1

In Section 1.2.7, we construct a better suited null model that reproduces the degree sequence of DBLP COAUTHOR. Then we also investigate the statistical significance of our findings (and hence omit it here). While the improved null model yields quantitatively different results, we arrive at a qualitatively similar conclusion. ◀

In the case study, we observe an elevated frequency of closed triangles, an effect known as *triadic closure* [309]. There are several natural explanations for triadic closure; for instance all three authors may have collaborated on the same article. Alternatively, their small distance within the network may hint at similar research fields, or it may indicate a small social distance (i.e., they all know each other). All these reasons suggest some form of *homophily*, the tendency of individuals to preferentially interact with persons with similar interests, professions, values, et cetera.

triadic closure

homophily

In short, there are processes in the real world that lead to local clustering, which should be reflected by realistic random graph models. We discuss one approach in Section 1.2.6 — namely, models that presume a geometric embedding of the network and that use the geometric distance as a proxy for topological distance.

1.2.5.2 Community Structure

Similarly, clustering can also be observed at larger scales. Consider for instance members of a working group or research project. Then it is likely that this *community* will be structurally observable in our collaboration network as a cluster, i.e., their vertex-induced subgraph will be denser than their outside connections.

community detection

The detection of such communities is an active research field and warrants systematic benchmarks to empirically compare the performance of different detection algorithms. While real data is valuable, it is limited (especially in case of large instances), unstructured, and noisy. Additionally, it is not always clear where exactly communities lie, and whether they are detectable in the network structure at all. Thus, systematic studies exclusively based on observed inputs may be hard to interpret. A mixture of real data and synthetic benchmarks helps to mitigate this issue.

⁴We scale the number of triangles by the number N_2 of 2-paths to ease interpretation. However, observe that N_2 is identical in all simple graphs sharing the same degree sequence. Let $N[v]$ be the neighborhood of v and derive $N_2 = 1/2 \sum_{v \in V} \sum_{u \in N[v]} (\deg(v) - 1) = \sum_{v \in V} \binom{\deg(v)}{2}$. In case of DBLP COAUTHOR we find $N_2 \approx 1.15 \cdot 10^{10}$.

A de-facto standard benchmark for this task is the *LFR* model [210, 208]. It artificially plants a community structure and outputs it as ground truth to quantify the performance of community detection algorithms. In the following, we briefly introduce the sampling scheme for undirected and unweighted networks with non-overlapping communities. Note, however, that more general variants introduced in [208] are structurally similar.

The *LFR* model samples a powerlaw degree sequence and a community size distribution from independent powerlaw distributions. It also assigns each node to a random community. The sampling steps are subject to a number of constraints ensuring that it is feasible to construct a matching graph.

LFR reserves for each node u a constant fraction of u 's neighbors for the so-called *inter-community graph* containing only edges between nodes of different communities. The remaining neighbors are selected from within u 's community.

Based on this network blueprint, the model randomly samples all communities independently from the *Fixed-Degree-Sequence-Model (FDSM)*. It does the same for the large inter-community graph. Since the *FDSM* is unaware of the community structure, it may produce an inter-community graph in which two nodes within the same community are connected; these defects are repaired using a Markov-chain process very similar to the one employed by *FDSM* itself.

1.2.6 Hidden Distances

In the previous section, we suggested the existence of *social distance* as an abstract measure for the separation of two individuals in a presumed *social space*. These notions are rooted in sociology and have been used for random graphs models (e.g., [343, 57]).

Notably, Watts et al. [343] propose the *Watts* model of social networks in which the linking probability between two individuals decreases exponentially with their social distance. To this end, each individual is assigned a “hidden” multi-dimensional *identity vector*. Each dimension can be interpreted as a certain facet of the person, such as geographical position or occupation, and is hierarchically organized as a tree. Then, the social distance between two individuals is defined via the smallest shortest-path distance between both persons in any dimension, i.e., similarity in a single identity trait already yields social vicinity.

The *Watts* model is designed to explain the small-world property of social networks [243]. In an empirical study Milgram and Travers [243, 326] give evidence that the average length of a chain of acquaintances between a randomly selected pair of US citizens is small. Crucially, the experiment also suggests that humans can efficiently navigate the network even if they know little about the other person. Many applications may benefit from understanding and replicating this ability.

Watts et al. approach the issue by experimentally showing that greedy routing based on identity vectors (also known as geographic routing) works well in their model and “captures the essence of the real small-world problem” [343]. Unfortunately, the model seems unsuitable for general complex networks as it is highly specific due to its non-generic social space, and also exhibits a homogenous degree distribution.

further block models:
 [Section 2.7](#)

I/O-efficient LFR:
 [Chapter 4](#)

FDSM:
 [Section 1.2.7.1](#)

social distance
social space

small-world property

geographic routing

1.2.6.1 Geometric Embedding


Watts et al. suggest that their observations apply to a broader class of decentralized search problems. For the general case, the social space is typically modeled as a generic geometric space (i.e., the resulting spatial networks associate each vertex with a point placed in some geometric space). Depending on the model, the probability of connecting two nodes is governed by (i) their relative orientation (e.g., Sections 2.5.1, 2.5.2 and 2.5.4.1), (ii) their absolute position (e.g., Section 2.5.2), or (iii) additional geometric constraints (e.g., Sections 2.5.3 and 2.5.4).

RGG: Euclidean Random Geometric Graphs

Random Geometric Graphs (RGG) are in a sense the most natural variant as they embed networks into a d -dimensional Euclidean space [149, 270]. These types of networks have been extensively studied in the context of physical communication networks (e.g., wireless networks [183, 160]) and the spreading of diseases [149]. However, in general, *RGGs* are inadequate to model complex networks since most relevant parameter domains yield large average path lengths and nearly homogeneous degree distributions.

RHG: Random Hyperbolic Graphs

Random Hyperbolic Graphs (RHG) overcome this issue by replacing Euclidean geometry with Hyperbolic geometry. Despite their simplicity, these models naturally capture and explain many topological features of complex networks (among others, all those discussed earlier; see [136] for a recent survey). *RHG*s have a powerlaw degree distribution [159] with controllable exponent $\gamma > 2$, realistic component sizes [54], local clustering [159], and a small diameter [134]. It is also known that many observed networks imply a hyperbolic geometry and can be embedded with little distortion [56, 15, 50, 303]; then geographical routing works almost optimally [56, 194].

sampling Threshold RHG:
 *Chapters 6 and 8 and Section 7.7.3*

There are two major *RHG* variants, namely the threshold and the binomial types. *Threshold RHGs* link two nodes iff their hyperbolic distance falls below a global threshold. While this variant is easier to analyze and to sample from, *Binomial RHGs* are arguably more realistic as they allow longer edges and shorter non-edges with a small probability. This is achieved with an additional parameter $T \geq 0$ called *temperature*. *Threshold RHG* and *Binomial RHG* coincide for $T \rightarrow 0$ while larger values of T reduce local cohesion.

sampling Binomial RHG:
 *Chapter 8*

1.2.7 Prescribed Degree Sequences as Null Models

A common problem in network analysis is to judge the statistical relevance of an observation. We have already encountered such questions, for instance, we found that 17.4% of all 2-paths in the DBLP COAUTHOR are within a closed triangle which is a high value compared to the $\mathcal{G}(n, p)$ model. The $\mathcal{G}(n, p)$ model, however, is fitted using only the number of nodes n and edges m respectively, thereby neglecting potentially important structural information.

original investigation:
 *Section 1.2.5*

To highlight the importance of selecting an appropriate reference point, consider a graph model consisting of an almost fully connected central component, say $\mathcal{G}(n', P)$ with $P \rightarrow 1$, surrounded by singleton nodes and node pairs only. We can fit this model to match any n and m by choosing appropriate sizes for the three subgraph types. One obtains a graph in which only the central component contains 2-paths. Thus, each 2-path has a vertex-induced triangle with probability $P^3 \approx 1$. Based on this model, the

aforementioned 17.4 % is improbably low. While this graph admittedly seems contrived, there are models specifically designed to have high local cohesion (e.g., [177, 110]) which lead to a similar conclusion.

Thus, we need a way to construct an unbiased null model which still produces similar graphs. A commonly accepted method is to consider all graphs with a similar or identical degree sequence as the network under investigation.

The computational cost of this approach heavily depends on its exact formulation. Two models with linear work sampling algorithms are the *Chung-Lu (CL)* model and the *Configuration Model (CM)*. The *CL* model produces the prescribed degree sequence only in expectation (see Section 2.6.1 for details). The *CM*, on the other hand, exactly matches the prescribed degree sequence but permits self-loops and multi-edges. These parallel edges affect the uniformity of the model [260, p.436] and are inappropriate for certain applications; however, erasing them may lead to significant changes in topology [297].

degree sequence implies null model

Chung-Lu matches degrees only in expectation

Configuration Model permits self-loops and multi-edges

I/O-efficient CM:  Chapter 4

1.2.7.1 Simple Graphs with a Prescribed Degree Sequence

In the following, we focus on simple graphs (i.e., without self-loops or multi-edges) matching a prescribed degree sequence exactly.


The *Fixed-Degree-Sequence-Model* is a common solution to obtain simple graphs from a prescribed degree sequence. It first manifests a biased deterministic graph using the HAVEL-HAKIMI algorithm and then uses an *Edge Switching (ES)* Markov chain process to perturb the graph. In each step, the process selects two edges uniformly at random and exchanges their incident nodes – by doing so the degrees remain constant. If a step were to result in a self-loop or multi-edge, it is rejected without replacement.

Fixed-Degree-Sequence-Model:  Chapter 4

Despite intensive research, it remains an open problem to find practical upper bounds on the Markov chain’s mixing time; i.e., the number of steps required to obtain a uniform sample. In practice, a small multiple of the number of edges typically suffices.

mixing time:  Section 2.6.3

Curveball (CB) is a more recent process but structurally similar to *ES*; instead of exchanging the endpoints of edges, *CB* shuffles the neighborhoods of two nodes. Assuming a sufficiently large average degree, each step of this process (a so-called *trade*) can inflict larger changes to a graph as compared to *ES*, possibly yielding faster mixing times [80, 321, 331]. *Global Curveball (G-CB)* is a variant of *CB* that coalesces $n/2$ trades into a single Markov chain step targeting each node exactly once. This variant has benefits in the implementation and analysis, and additionally seems to converge faster. Still practical and rigorous upper bounds on *CB*’s mixing time are elusive.

Curveball (CB) efficient CB generators:  Chapter 5

Case Study 1.4. In the following, we experimentally support our previous claim that the number of triangles n_{Δ} in DBLP COAUTHOR is significantly elevated. We thus construct a null model without local cohesion and show that it cannot explain our observation.

homophily:  Section 1.2.5

The model is defined as the uniform distribution over all simple graphs with the same degree sequence as DBLP COAUTHOR. In other words, our null-hypothesis is that the observed number of triangles is plausible for graphs with such a degree sequence.

Unfortunately, enumerating all graphs in the support of our null model is infeasible due to computational cost. Hence, we resort to sub-sampling and do so by perturbing

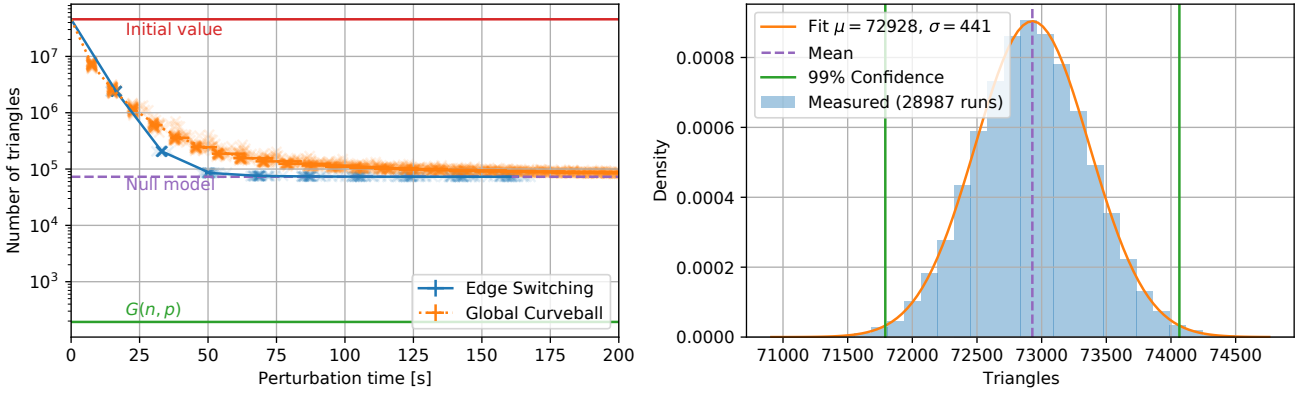


Figure 1.2: Number of closed triangles (n_{Δ}) in DBLP COAUTHOR. **Left:** Progress of *Curveball* and *Edge Switching* as functions of time. For both null models, the line indicating the respective average completely covers the value range of the 99%-confidence interval shown in the right plot. **Right:** Distribution of triangle count after 500 global trades.

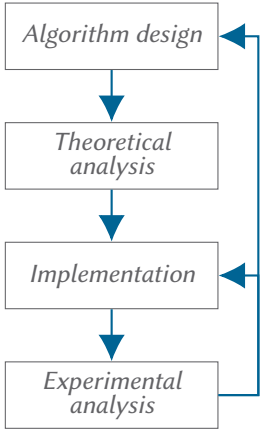


Figure 1.3: Algorithm engineering design circle

DBLP COAUTHOR multiple times. For each run, we compute the number of triangles n_{Δ} after $100m$ *Edge Switching* steps, where m is the number of edges in the graph. For completeness, we also repeat the experiment with *Global Curveball*.

Figure 1.2 presents the measurements. We observe that *G-CB* mixes slower than *ES* which we attribute to the skewed degree distribution with lower average degree. We also find that the null model's average n_{Δ} is (a) 377 times larger than expected from $\mathcal{G}(n, p)$ with matching density and (b) 625 times smaller than in DBLP COAUTHOR.

To judge the statistical significance of our findings, we assume that the values of n_{Δ} measured from our null model follow a normal distribution. Roughly speaking, this can be explained using the central limit theorem since each measurement of n_{Δ} is based on a randomly sampled graph and can be interpreted as a sum of almost independent indicator variables. As indicated in Figure 1.2, our measurements have a mean of $\mu = 72,928$ and a standard deviation of $\sigma = 441$. DBLP COAUTHOR, in contrast, contains more than 45 million closed triangles. Thus, we can strongly reject the null-hypothesis. ◀

1.3 Practical Engineering Challenges

In the previous section, we introduced a number of random graph models and presented the underlying probability distributions in terms of intuitive sampling algorithms. These algorithms are designed to be easy to follow and thereby to communicate the models' central ideas to the reader. Unfortunately, direct implementations of such descriptions seldom yield efficient and scalable generators required to sample massive graphs.

The present thesis bridges this gap for a number of models by engineering generators that are efficient in theory and practice. To this end, we adopt the *algorithm engineering* methodology. As illustrated in Figure 1.3, it proposes an iterative design process alternating between theoretical analysis and empirical studies to identify, understand, and overcome the limitations of a given solution. This process may start with classical algorithm design assuming a *unit-cost Random-Access Machine (unit-cost RAM)* model.

unit-cost RAM

Example 1.5. Consider for instance the *Random Hyperbolic Graph* model that randomly scatters geometric points and connects any two with a sufficiently small distance. A naïve implementation may compute the distance of all pairs which requires at least quadratic work and is therefore prohibitively expensive even for medium-sized graphs. Improved algorithms exploit the geometric structure of the problem and sample instances in time nearly linear in the output size.

Such efficient approaches mark the starting points of the engineering efforts documented in Chapters 6 to 8. Each project optimizes according to different objectives, which results in quite different implementations of the same random graph model. ◀

The *unit-cost RAM*'s greatest strength is also its greatest weakness: depending on the point of view, the model either *abstracts away*, or *fails to capture* the complexity of practical machines. Taking the *unit-cost RAM* as the baseline, real machines imply non-trivial penalties and opportunities for acceleration:

- The model does not appropriately reflect real costs per unit of execution. For instance, the latency of basic arithmetic instructions such as integer addition and division can vary more than two orders of magnitude [130]. For a given machine, these discrepancies typically only contribute constant factors to the total runtime of an algorithm, and are therefore irrelevant for an asymptotic analysis. In practice, however, constants *do* matter. The disagreement is even more severe when factoring in the inhomogeneous cost of data access.
- The model does not include advanced features of modern computers such as parallelism or special hardware-accelerated instructions.

Constants do matter!

cost of data access:
📖 Section 1.3.1

hardware acceleration:
📖 Section 1.3.2 ff.

In the remainder of this section, we highlight a number of relevant aspects of modern computer hardware and discuss related formal models of computation. We augment this discussion with a selection of solutions developed for the present thesis.

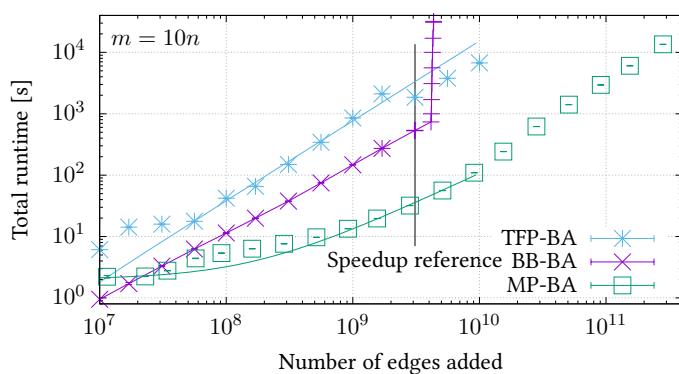
1.3.1 The Cost of Data Transfer

Modern computers are composed of a multitude of memory types. Physical, technological and economic constraints motivate a so-called *memory hierarchy* ranging from the highest level with fast, but small memory areas (registers, caches, et cetera) over main memory down to mass storage (e.g., solid-state drives, rotating disks, or network-attached storage).⁵ When comparing the memory at the upper and lower ends of the hierarchy, we find that their capacities, bandwidths and latencies typically differ in excess of five orders of magnitude. The vast majority of computers used today processes data only in the highest hierarchy levels, and hence needs to move data to and fro.

memory hierarchy: the higher the level, the faster but smaller the memory

⁵We omit additional types such as offline mass storage since these are of little concern in the context of the algorithms considered here.

Figure 1.4: Memory hierarchy affecting BA generators (cf. Section 3.5). BB-BA accesses unstructured memory locations. It becomes orders of magnitudes slower as soon as the data does not fit into main memory and gets swapped out to disk.



The complexity of the memory subsystem is typically abstracted away by an intricate cooperation between the hardware and the operating system; techniques include a uniform virtual address space which may even encompass mass storage. Substantial engineering efforts attempt to mitigate the thereby “hidden” costs of accessing data in lower levels. Unfortunately, most of these heuristics rely on some structure or locality of reference in the access patterns. Section 1.3.1 illustrates this behavior, as the BB-BA algorithm relies on unstructured accesses, which become prohibitively expensive if too low memory levels need to be involved. Two commonly considered types are spatial and temporal locality.

virtual addresses

spacial/data locality

prefetching
block transfers

temporal locality

caches

Spatial locality (also known as *data locality*) refers to the phenomenon that one access to a memory location is likely followed by additional accesses to addresses in the direct vicinity; e.g., when sequentially reading an array from beginning to end. Algorithms exhibiting data locality can be accelerated using explicit or speculative prefetching (to hide the latency of accessing data), and can benefit from reduced overheads by moving and caching blocks of data rather than accessing single data items.

Temporal locality suggests that an accessed address is likely to be accessed again in the near future. Examples include read-modify-write primitives, small and frequently queried lookup tables, or repeated scans through a small vector, e.g., for vector-matrix-multiplication. Temporal locality allows the usage of caches that transparently replicate small parts of lower memory levels to keep them close to the processing units.

While the basic principles of the memory subsystem evolve slowly over time, the exact costs involved depend on the machine at hand and are subject to more frequent changes (e.g., due micro-architecture revisions). The three models of computation introduced in the following are therefore heavily abstracted. They include just sufficient complexity to still allow useful performance predictions that in turn can guide the algorithmic development process.

1.3.1.1 The External Memory Model

External Memory Model

The *External Memory Model (EMM)* by Aggarwal and Vitter [7] is a commonly accepted theoretical framework to reason about the effects of memory hierarchies while processing big data problems.

While the model suggests a scenario with main memory (IM: internal memory) and secondary storage (EM: external memory), it has much broader applications since it promotes design patterns relying on temporal and spatial locality. It thereby often leads to cache-friendly algorithms with a good overall performance – an observation that motivates the strongly related *Cache-Oblivious-Model* [137].

The *EMM* features a two-level memory hierarchy with fast internal memory holding up to M data items, and a slow disk of unbounded size which stores the algorithm's input and output. Since the processor can only operate on data in main memory, it needs to access data in external memory using so-called I/Os. Each I/O transfers a block of B consecutive items between memory levels. Consequentially, the number of I/Os executed by an algorithm is used to measure its performance. Many I/O-efficient algorithms use a number of common primitives:

- Reading or writing n contiguous items on disk requires $\text{scan}(n) = \Theta(n/B)$ I/Os.
- We require $\text{sort}(n) = \Theta((n/B) \cdot \log_{M/B}(n/B))$ I/Os for comparison-based sorting of n items. Roughly speaking, the sorting complexity $\text{sort}(n)$ constitutes a lower bound for most intuitively non-trivial EM tasks [7, 242].
- Pushing and removing n elements from a priority queue is possible with $\text{sort}(n)$ I/Os [23, 22], and may be even cheaper for special cases such as integers of bounded magnitude. This allows *Time Forward Processing* [230], a general technique which translates unstructured I/Os into messages exchanged via a priority queue.

For all realistic values of n , B , and M sorting involves only very few scanning rounds and is in practice several orders of magnitude cheaper than unstructured I/O. While a plethora of software libraries dedicated to the efficient access of memory exists, I/O-efficient implementations in the experimental algorithmics community often employ *STXXL* [102] or *TPIE* [330].

1.3.1.2 Streaming Models

Streaming models form another family of models of computation for big data. They approach inputs exceeding internal memory by imposing a view even more restricted than *EMM*. While there are a number of formulations, they have in common that the input may only be observed in a scanning fashion (i.e., reading from the beginning to end). Some variants of the model consider up to $\Theta(\log n)$ passes acceptable where n is the number of tokens in the input stream. Another crucial constraint pertains to the algorithm's memory consumption that has to be highly sublinear in n .

1.3.1.3 Distributed Multicomputers and Communication Avoidance


Even before the advent of multiprocessors in off-the-shelf hardware, distributed multicomputers were available and remain the foundation of virtually all modern supercomputers. They consist of a (large) number of independent machines that are interconnected to collectively solve a common problem.

M : main memory size

B : block size

$\text{scan}(n) = \Theta(\frac{n}{B})$ I/Os

$\text{sort}(n) = \Theta(\frac{n}{B} \log_{M/B}(\frac{n}{B}))$ I/Os

[Time Forward Processing:](#)
 [Section 5.2.2](#)

streaming models

cf. I/Os in the EMM

communication-free =
communication-agnostic
suitable generators:
📖 Chapter 7

Roughly speaking, there are multiple processing units with local memory that is small compared to the problem size. Accessing non-local information is costly as it involves communication, which may be modeled in various ways (e.g., [329, 207, 99, 17, 101]). Hence, a common design goal is to minimize the number of messages exchanged.

In the present thesis, we only consider the extreme case of so-called *communication-agnostic* (also referred to as *communication-free*) algorithms. Then, each each processing unit is only aware of its rank, the total number of units, and some input configuration. However, it cannot exchange any further information during the execution of the algorithm. Many random graph models can be processed in a communication-agnostic way by recursive decomposition of the problem in combination with pseudorandomness.

Example 1.6. Recall *Random Hyperbolic Graphs (RHG)* where n points are randomly scattered onto a hyperbolic disk S . Assume that the number of nodes is too large to sample, let alone store, all nodes on every distributed machine.

Fortunately, a key property of relevant *RHG* graphs is that most nodes only have a very local neighborhood, i.e., a hyperbolic circle around each node suffices to compute all its links. Observe that many of these subsets overlap due to common edges. In general, there is no balanced mapping of nodes to processors without overlaps. Thus, any two processors with overlapping subsets have to have a consistent view of the underlying region of hyperbolic space.

We achieve this by partitioning the hyperbolic space into k cells (in Chapter 6, we instead use concentric bands). For each cell i , we seed a pseudorandom generator with a value deterministically derived from the cell's index i and, subsequently, use the generator to sample the n_i points contained within the cell. By construction, this process yields consistent results even if executed by multiple independent processing units.

The only information missing is the number n_i of points in cell i . If the cell was considered on its own as in most proofs pertaining to *RHG*, a Poisson process applies. Here, however, the vector $\mathbf{N} = (n_1, \dots, n_k)$ follows a multinomial hypergeometric distribution due to the side condition that exactly n points need to be scattered in total ($\sum_i n_i = n$). Again, all processors obtain consistent values for \mathbf{N} using common seeds for their pseudorandom generators. ◀

sRHG:

📖 Section 7.7.3

HYPERGEN:

📖 Chapter 6

further improvements:

📖 Chapter 7

1.3.2 Advanced Features of Modern Computers

In the (not so) early days of modern computers, the performance of processors grew exponentially and doubled every one to two years. This development was characterized by Moore's "Law" [248] and Dennard Scaling [104].

Moore's Law
Dennard Scaling


Moore observed that the number of components (often simplified as transistors) in an integrated circuit doubles approximately every two years. Dennard et al. noted that the shrinking transistor allowed to reduce the operating voltage and increase the processor's frequency at the same time. Thus, the power density (i.e., consumption per chip area) would approximately stay constant despite rising frequencies and computational power.

Dennard Scaling eventually stopped at the beginning of the 21st century as the small structure sizes ($\lesssim 22$ nm) of integrated circuits reached the point where leak currents and quantum effects significantly contributed to a device's power consumption [58]. While the eventual end of Moore's Law also seems inevitable, integrated circuit fabrication processes still continue to improve at the time of writing the present thesis.

Moore's Law and break down of Dennard Scaling leads to parallel processors

As a consequence of Moore's Law, processor manufacturers can fit an increasing number of functions into processors. Besides functions pertaining to the system design itself (e.g., tasks previously handled by the chipset), many features directly affect the way algorithms are implemented. The nature of such features and extensions available in commodity hardware is strongly related to the decline of Dennard Scaling.

As the performance of sequential execution stagnates, chip designers embrace parallelism in several dimensions (cf. [129]). In the present thesis, we are explicitly concerned with three types of parallelism: distributed multicomputers, shared-memory parallelism, and data parallelism. In the following, we only consider the latter two types since we already identified the cost of data transfer as the main challenge of algorithm design for distributed computing.

distributed computing:
 [Section 1.3.1.3](#)

1.3.3 Shared-Memory Parallelism

Shared-memory parallelism refers to systems in which multiple processors, often on the same chip, operate with a common main memory. Such machines typically suffer from limited scaling capabilities due to physical and technological constraints. A notable exception are massively parallel general purpose graphics processors (GP-GPUs, or GPUs in short), which often include thousands of small processing units able to efficiently sustain even more logical tasks (also known as threads).

An advantage of shared memory is the significantly lower cost of data exchange compared to distributed computers. This gives rise to new algorithmic techniques relying on fine-grained communication using atomic primitives such as compare-and-swap, test-and-set, or fetch-and-add.

shared memory allows fine-grained data exchange

Two major potential bottlenecks in shared-memory parallel programs are *congestion* caused by concurrent memory access, and costs due to *synchronization* with barriers or locks. Thus, algorithms and data structures for modern computers generally avoid global synchronization of parallel processes as best as possible (cf. obstruction-, lock-, and wait-freedom [174]).

congestion
synchronization

The majority of generators proposed in the following chapters take an orthogonal route and formulate the sampling process in a way that exposes independent subproblems. Such subtasks can be carried out by different processors and be computed in parallel without further interaction. This favorable type of orchestration is referred to as *pleasingly parallel* or *embarrassingly parallel*.

pleasingly parallel

Unfortunately, not all problems lend themselves to pleasingly parallel computation, and therefore require more advanced algorithmic treatment. To this end, shared-memory parallelism is frequently captured by the *PRAM* model of computation [187, 295].

PRAM

A *PRAM* consists of P processing units (PUs) with small local registers that operate on a common memory. There exist several subtypes to resolve conflicts arising from concurrent reads and writes to the same memory address. Each PU is aware of its unique rank for *symmetry breaking* (i.e., to find its unique role in an otherwise fully symmetric system). Synchronization issues are typically avoided by assuming that all PUs operate in lock-step.⁶

*symmetry breaking**work**work-optimality*

An established measure of algorithmic performance in the *PRAM* model is *work*. It is defined as the product of the number P of PUs and an algorithm's runtime. Then, an algorithm is said to be *work-optimal* if it requires work linear in the runtime of the best sequential algorithm.

MP-BA:

Chapter 3

Example 1.7. Our parallel generator MP-BA uses a binary tree partially stored in external memory to implement dynamic weighted sampling. Starting at the root, the algorithm pushes a large number of requests down the evolving decision tree. At any point in time, the criterion of each inner node v depends on all requests previously processed at v . We use two types of parallelism to accelerate this process.

First, we observe that, in the relevant range, the expected number of requests at an inner node decreases exponentially with the node's depth. Hence, we cut the computation at a certain level and consider each subtree rooted there as its own independent sub-problem. Thus, the subtrees can be processed *pleasingly parallel*.

To avoid congestion in the upper levels of the tree, nodes near the root are processed using several hundreds of GPU processors each. To this end, we devise a *PRAM* algorithm that collects multiple requests and works on them in parallel. The key insight is that, given an appropriate batch size, we can process most queries without considering their predecessor which are currently processed in parallel. They can therefore be concurrently pushed into the appropriate subtrees. Then, the algorithm recurses on the yet undecided requests and terminates with high probability after a constant number of steps.

prefix sum

Per iteration, we use a constant number of work-optimal parallel *prefix sums* [52] to identify and propagate safe requests. Thus, with high probability our algorithm requires only linear work and is therefore work-optimal. ◀

1.3.4 Data Parallelism

SIMD

Data parallelism refers to a mode of operation where a single function is applied to multiple independent values. It is also known as Single-Instruction-Multiple-Data [129] (SIMD). Data parallelism implies a single control path which is in stark contrast to the parallelism types discussed before where each execution unit is more or less independent and can execute their own section of the program.

⁶While lock-step operation is arguably an unrealistic assumption for most practical multi-processors, it is a good first-order approximation of the single-instruction-multiple-thread paradigm (SIMT) used in modern GPUs (e.g., *warps* [219] or *wavefronts* [5]).

Modern x86 processors feature a number of instruction-set extensions devoted to SIMD including MMX as well as the SSE and AVX families [184]. Most notably, AVX-512 features 512 bit-wide registers and enables simultaneous operations on as little as eight double-precision floats to up to 64 small 8 bit-integers.⁷

AVX-512

Implementation details of the vector units (e.g., reduced base clocks) as well as algorithmic overheads typically cause a speedup slightly below the number of operands; still, vector instructions are quite power-efficient, thereby allowing the processor to sustain more operations within its specified thermal design power [86]. The efficiency of vector instructions also heavily depends on the algorithms they are used in. Suitable algorithms expose sufficient data parallelism and a high degree of data locality to facilitate block transfers to-and-fro the large registers. Another crucial aspect is a uniform control flow among all data words processed in parallel; *path divergence* (i.e., the necessity to apply different instructions to different parts of a register) is possible using so-called *masking*, but often detrimental to performance.

data locality

path divergence

Example 1.8. Our RHG generator HYPERGEN is designed with SIMD in mind, and its implementation uses explicit vector instructions. Among others, they enable us to compute eight hyperbolic distances in double-precision simultaneously. The vectorized section ends when the computed distances have been compared to the threshold distance, i.e., when all node pairs implying an edge have been found. This is a natural point since the different treatment of pairs with and without an edge leads to path divergence. ◀

HYPERGEN:

📖 Chapter 6

1.4 Articles Included in the Present Thesis

While preparing the present thesis, I authored and co-authored eight peer-reviewed conference articles (two of which received best paper awards from the European Symposium on Algorithms), three published or accepted journal articles, two submitted book chapters, and three technical reports. The remainder of this section contains a brief overview of the eight publications in Chapters 3 to 8 selected for the present thesis.

comprehensive summary:

📖 Chapter 9


Chapter 2 surveys the current state of the art of scalable random network generation. It complements the “hands-on” introduction of this thesis with a broader, though less detailed, overview. It is the only included article that is still under submission.

The remaining chapters discuss algorithmic and engineering contributions of practical scalable generators for three random graph families.

Preferential Attachment




Chapter 3, based on [239], introduces two I/O-efficient generators for models based on preferential attachment, and demonstrates the techniques on *Barabási-Albert* graphs. One generator supports massively parallel heterogeneous hardware consisting of CPU processors and general purpose graphics processors. The software stack consists of several interacting algorithms tailor-made for the quite different hardware types.

⁷The not yet available Scalable Vector Extension [318] by ARM supports four times larger registers.

-  *Chapter 3* [239] U. Meyer and M. Penschuck. Generating massive scale-free networks under resource constraints. In M. T. Goodrich and M. Mitzenmacher, editors, *Workshop on Algorithm Engineering and Experiments ALENEX*, pages 39–52. Society for Industrial and App. Math. SIAM, 2016. doi:10.1137/1.9781611974317.4 .

Graphs from Prescribed Degree Sequence

Chapters 4 and 5, based on [82, 167, 168], are concerned with I/O-efficient Markov chain processes for the perturbation of simple graphs. In Chapter 4, we develop EM-LFR, an I/O-efficient sampling pipeline for the *LFR* community detection benchmark, and engineer a parallel implementation able to produce graph instances orders of magnitude larger than the available main memory. EM-LFR consists of several algorithms, out of which the *Edge Switching (ES)* Markov chain is the most challenging building block. In Chapter 5, we consider the novel *Curveball* and *Global Curveball* processes as an alternative of *ES*, and show that their implementations can expose more data locality.

-  *Chapter 4* [167] M. Hamann, U. Meyer, M. Penschuck, H. Tran, and D. Wagner. I/O-efficient generation of massive graphs following the LFR benchmark. *ACM J. of Experimental Algorithmics*, 23, 2018. doi:10.1145/3230743 .
-  *Chapter 4* [168] M. Hamann, U. Meyer, M. Penschuck, and D. Wagner. I/O-efficient generation of massive graphs following the LFR benchmark. In S. P. Fekete and V. Ramachandran, editors, *Workshop on Algorithm Engineering and Experiments ALENEX*, pages 58–72. Society for Industrial and App. Math. SIAM, 2017. doi:10.1137/1.9781611974768.5 .
-  *Chapter 5* [82] C. J. Carstens, M. Hamann, U. Meyer, M. Penschuck, H. Tran, and D. Wagner. Parallel and I/O-efficient randomisation of massive networks using Global Curveball trades. In Y. Azar, H. Bast, and G. Herman, editors, *European Symp. on Algorithms ESA*, volume 112 of *LIPICs*, pages 11:1–11:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.ESA.2018.11 .

Geometrically Embedded Random Graphs

Chapters 6 to 8 present (among others) generators for *Random Hyperbolic Graphs (RHG)* and *Geometric Inhomogenous Random Graphs (GIRG)*. In Chapter 6, we develop a streaming generator for *Threshold RHGs* targeting accelerator hardware with a small dedicated memory. The resulting generator is a sweep-line algorithm which “uncovers” random points in a structured fashion inspired by *order statistics* [100]. We show that, at any point in time and with high probability, only a small fraction of the geometry needs to be stored. The parallel implementation introduces a number of acceleration techniques and remains among the fastest *RHG* generators available for shared-memory parallelism at the time of writing the present thesis.

Refined versions of these techniques carry over to Chapters 7 and 8. For Chapter 7, I contributed the communication-agnostic distributed streaming generator *sRHG* for *Threshold RHGs*. It combines techniques of [271] and [139].

Chapter 8 engineers a previously known sampling algorithm by Bringmann et al. [69] for *GIRG* graphs and adapts it to Binomial *RHG*s. While the original algorithm has an already optimal expected runtime, the chapter discusses significant engineering efforts to obtain a generator that is also fast in practice.

- [271] M. Penschuck. Generating practical random hyperbolic graphs in near-linear time and with sub-linear memory. In C. S. Iliopoulos, S. P. Pissis, S. J. Puglisi, and R. Raman, editors, *Int. Symp. on Experimental Algorithms SEA*, volume 75 of *LIPICs*, pages 26:1–26:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.SEA.2017.26 . 📖 Chapter 6
- [138] D. Funke, S. Lamm, U. Meyer, M. Penschuck, P. Sanders, C. Schulz, D. Strash, and M. v. Looz. Communication-free massively distributed graph generation. *J. Parallel Distributed Comput.*, 131:200–217, 2019. doi:10.1016/j.jpdc.2019.03.011 . 📖 Chapter 7
- [48] T. Bläsius, T. Friedrich, M. Katzmann, U. Meyer, M. Penschuck, and C. Weyand. Efficiently generating geometric inhomogeneous and hyperbolic random graphs. In M. A. Bender, O. Svensson, and G. Herman, editors, *European Symp. on Algorithms ESA*, volume 144 of *LIPICs*, pages 21:1–21:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.ESA.2019.21 . Best Paper · ESA'19 B
📖 Chapter 8

Publications Not Included

While preparing the present thesis, I additionally contributed to the following articles.

- [240] U. Meyer and M. Penschuck. Large-scale graph generation and big data: An overview on recent results. *Bulletin of the EATCS*, 122, 2017. URL: <http://eatcs.org/beatcs/index.php/beatcs/article/view/494> .
- [241] U. Meyer and M. Penschuck. Large-scale graph generation: Recent results of the SPP 1736 – Part II. *it - Information Technology*, 2020 .
- [4] D. Achlioptas, A. Coja-Oghlan, M. Hahn-Klimroth, J. Lee, N. Müller, M. Penschuck, and G. Zhou. The random 2-SAT partition function. *CoRR*, abs/2002.03690, 2020. Accepted for Random Structures And Algorithms. arXiv:2002.03690 .
- [147] O. Gebhard, M. Hahn-Klimroth, O. Parczyk, M. Penschuck, M. Rolvien, J. Scarlett, and N. Tan. Near optimal sparsity-constrained group testing: improved bounds and algorithms. *CoRR*, abs/2004.11860, 2020. URL: <https://arxiv.org/abs/2004.11860>, arXiv:2004.11860 .
- [6] P. Afshani, R. Fagerberg, D. Hammer, R. Jacob, I. Kostitsyna, U. Meyer, M. Penschuck, and N. Sitchinava. Fragile complexity of comparison-based algorithms. In M. A. Bender, O. Svensson, and G. Herman, editors, *European Symp. on Algorithms ESA*, volume 144 of *LIPICs*, pages 2:1–2:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.ESA.2019.2 . Best Paper · ESA'19 A

- [108] P. Dinklage, J. Ellert, J. Fischer, D. Köppl, and M. Penschuck. Bidirectional text compression in external memory. In M. A. Bender, O. Svensson, and G. Herman, editors, *European Symp. on Algorithms ESA*, LIPIcs. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ESA.2019.41 .
- [40] P. Berenbrink, D. Hammer, D. Kaaser, U. Meyer, M. Penschuck, and H. Tran. Simulating population protocols in sub-constant time per interaction. In *European Symp. on Algorithms ESA*, volume 173 of *LIPIcs*, pages 16:1–16:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ESA.2020.16 .

Recent Advances in Scalable Network Generation

joint work with

U. Brandes, M. Hamann, S. Lamm, U. Meyer, I. Safro, P. Sanders, and C. Schulz

Random graph models are frequently used as a controllable and versatile data source for experimental campaigns in various research fields. Generating such datasets at scale is a non-trivial task as it requires design decisions typically spanning multiple areas of expertise. Challenges begin with the identification of relevant features in domain specific networks, continue with the question of how to compile such features into a tractable model, and culminate in algorithmic details arising while implementing the model.

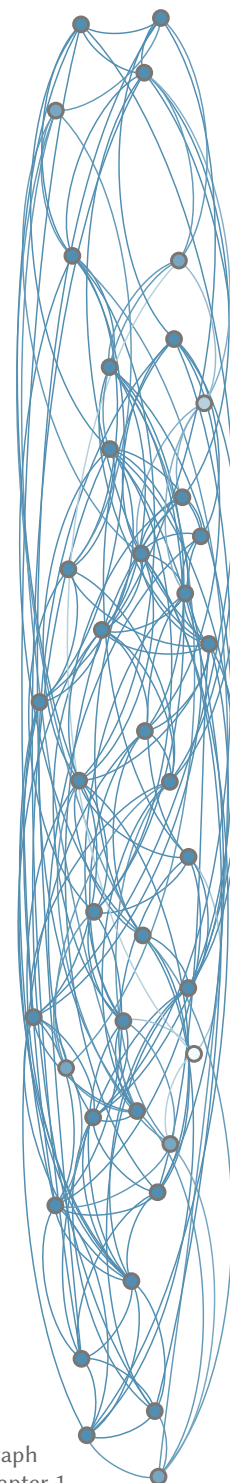
In the present survey, we explore crucial aspects of random graph models with known scalable generators. We begin by briefly introducing network features considered by such models, and then discuss various random graphs alongside with generation algorithms. Our focus lies on modeling techniques and algorithmic primitives that have proven successful in obtaining massive graphs. We consider concepts and graph models for multiple domains (such as social network, infrastructure, ecology, and numerical simulations), and discuss generators for different models of computation (including shared-memory parallelism, massive-parallel GPUs, and distributed systems).

This chapter is based on a yet unpublished manuscript [272].

[272] M. Penschuck, U. Brandes, M. Hamann, S. Lamm, U. Meyer, I. Safro, P. Sanders, and C. Schulz. Recent advances in scalable network generation. *CoRR*, abs/2003.00736, 2020. arXiv:2003.00736 .

My contribution

I coordinated this project and contributed a more than proportional amount of material.



$\mathcal{G}(n, p)$ with $n = 39$ and $p \approx 0.2$ to match the density of the DBLP subgraph as illustrated on the title page of Chapter 1

2.1 Introduction

Generating synthetic networks is one of the most active research areas in network science and general graph algorithms. Theoreticians, domain experts, and algorithm developers need high-quality network data for various purposes such as algorithm engineering, decision-making, and simulations. However, obtaining real-world network data is often accompanied with various obstacles. Some of the reasons for the resulting shortage include classified or proprietary information, legal hindrances, and economic considerations to obtain high-quality data as well as simple lack of future data in evolving networks. For very large graphs processed on supercomputers, transferring the graphs to data centers, storing them indefinitely on disks, and loading or distributing them for experiments can also be quite expensive. In these cases, synthetic networks generated by models are used to substitute the real-world networks.

quality vs complexity

There are two major considerations in the process of designing a synthetic network generator. Firstly, generating data from a high-quality synthetic model requires reproducing important structural properties observed in reference data. Alternatively, one may consider to generate networks with predefined properties but with no original reference (or only much smaller examples). Each case requires an unbiased coverage of the entire set of relevant networks. Depending on the complexity of the structural properties, a generator may need to solve optimization problems of varying complexity. Secondly, for many applications, the generation process must be sufficiently scalable to produce large-scale instances in a reasonable time with respect to the available computational resources. For example, there is little practical value in generating a small network of several tens of nodes for social information network analysis.

*importance of large-scale
synthetic networks*

These considerations suggest a traditional quality/runtime trade-off that must be taken into account when designing or using generators for large-scale networks. In particular, this is important when many large-scale synthetic networks are required for computational experiments; e.g., in order to demonstrate the robustness of an algorithm, to gather sufficient statistical information, or to observe the long-term evolution of a network. While there exist several comprehensive surveys [112, 152, 62] that focus on the quality component of this trade-off, we find that not many pay close attention on the combination. In this survey, we discuss various classes of generation methods. We focus on their scalability, algorithmic, and implementation aspects, as well as on the relevance of different parallel programming models to maintain acceptable performance in the generation of large-scale networks.

2.2 Graph Properties and Uses of Generators

$\mathbb{G}(n)$ and $\mathcal{G}(n)$

The set $\mathbb{G}(n)$ of all simple undirected graphs with n vertices contains $2^{\binom{n}{2}}$ graphs. We define the model $\mathcal{G}(n)$ as the uniform distribution over $\mathbb{G}(n)$. The $\mathcal{G}(n)$ model has proven useful for the probabilistic method in existential combinatorics [59], but it is not at all a plausible assumption for statistics of empirical graphs. Just consider that a graph sampled uniformly at random from $\mathbb{G}(n)$ contains half of the $\binom{n}{2}$ edges in

$[x_i]_{i=a}^b$	Sequence or set $[x_a, x_{a+1}, \dots, x_b]$
$[a..b], [k]$	Sequence or set $[a..b] := [i]_{i=a}^b, [k] := [1..k]$
$\langle X \rangle$	Average, $\langle X \rangle := \frac{1}{n} \sum_{v \in V} X(v)$
$cc(v), cc(G)$	Clustering coefficient $cc(v) := \text{dens}(G[N(v)])$, $cc(G) := \langle cc \rangle$
$\text{dens}(G)$	Density, $\text{dens}(G) := m/\binom{n}{2}$ if $n > 1$ else $\text{dens}(G) = 0$
$\text{dist}(u, v)$	Length (number of edges) of shortest path between u and v
$G = (V, E)$	Graph with nodes $V = [n]$ and edges $E \subseteq \binom{V}{2}$
$G[V']$	Induced subgraph (V', E') if $G=(V, E)$, s.t. $V' \subseteq V$, $E' = \{e \mid e=\{u,v\} \in E \wedge u,v \in V'\}$
$\mathbb{G}(n), \mathbb{G}(n, m)$	Set of graphs with $ V = n$ and $ E = m$, respectively
$\mathcal{G}(n), \mathcal{G}(n, m)$	Uniform distributions over $\mathbb{G}(n)$ and $\mathbb{G}(n, m)$, respectively
n, m	Number n of nodes, number m of edges
$N(v), \text{deg}(v)$	Neighbors $N(v) := \{u : \{u, v\} \in E\}$, degree $\text{deg}(v) := N(v) $
whp.	With high probability: true with probability of at least $1 - 1/n^c$ for $c \geq 1$

Table 2.1: Notation used in this survey

expectation – which is more than typically found in empirical networks (see below). To better match empirical distributions of graphs encountered in different domains of application, models are devised in which certain graph invariants can be controlled for. In this section, we provide background on some of such properties and a number of important model classes. The section concludes with an outline of a few ways in which generators parameterized on graph properties are commonly applied.

2.2.1 Graph Properties

We next recall definitions of a number of graph properties that are commonly used to constrain or skew the distribution of graphs generated from a model. All models referred to below are discussed in more detail in later sections.

Density

We have parameterized the class $\mathbb{G}(n)$ of all graphs by their *order* n , i.e., the number of vertices. The other obvious parameter to control for is m , the number of edges. The overwhelming majority of graphs in $\mathbb{G}(n)$ is dense, i.e., $m = \Theta(n^2)$, but graphs arising in application domains are often *sparse*, say $m = \mathcal{O}(n \log n)$, as, for instance, edges are associated with costs or vertices have limited capacity to be adjacent with others.

For any given combination of n and $0 \leq m \leq \binom{n}{2}$, we obtain a subclass $\mathbb{G}(n, m) \subseteq \mathbb{G}(n)$ of graphs $G = (V, E)$ with $V = [n]$ and $|E| = m$. They all have the same *density* $\text{dens}(G) := m/\binom{n}{2}$, where we define $\text{dens}(G) := 0$ for $n \in \{0, 1\}$. Since the sum of degrees equals twice the number of edges, all graphs in $\mathbb{G}(n, m)$ also have the same *average degree* $\text{avg deg} := \frac{1}{n} \sum_{v \in V} \text{deg}(v) = 2m/n$.

Analogously to $\mathcal{G}(n)$, we denote the uniform distribution on $\mathbb{G}(n, m)$ by $\mathcal{G}(n, m)$ and $\mathcal{G}(n, p)$. While $\mathcal{G}(n, m)$ was the original model of Erdős and Rényi [121], Gilbert [148] intro-

duced the related $\mathcal{G}(n, p)$ model defined on all of $\mathbb{G}(n)$ where each edge is present independently with probability p (see Section 2.4.1). Thus, the number of edges, and as a consequence density and average degree, are fixed only in expectation. Note that $\mathcal{G}(n, p) = \mathcal{G}(n)$ for $p = 1/2$.

clustering coefficient

Random graphs in each of the three models are *balanced*, in the sense that the expected density of any vertex-induced subgraph is constant. To assess differences in local density of a graph $G = (V, E)$, the *clustering coefficient* of a vertex $v \in V$ is defined as $cc(v) := \text{dens}(G[N(v)])$, i.e., the density of the subgraph induced by v 's neighborhood. The average over all vertices is called the clustering coefficient $cc(G) := \frac{1}{n} \sum_{v \in V} cc(v)$ of the graph.

Models that favor local cohesion (i.e., produce substructures with an above-average internal density) include *random geometric graphs* (see Section 2.5). Here, vertices are randomly assigned to positions, e.g., in a Euclidean or hyperbolic space, and edge probabilities are made dependent on the distance in that space.

Heterogeneity in local density is introduced in *planted partition* or *Stochastic Block Model* (see Section 2.7), where vertices are partitioned into subsets, and different edge probabilities are assigned depending on which subsets the endpoints are in.

Degrees

\mathcal{D} and $\deg(v)$

FDSM:

☞ Section 2.6

Similar to density, there may be a tendency for degrees to be distributed unevenly in a graph. The subset $\mathbb{G}(n, m)$ can be restricted further by fixing not only the number of edges but the entire *degree sequence*, i.e., using a sequence $\mathcal{D} := [d_i]_{i=1}^n$ to prescribe the degree $\deg(v) = d_v$ of each node $v \in [n]$. This is done in *Fixed-Degree-Sequence-Model*. As was done for density in the case of $\mathcal{G}(n, p)$, we may also want to sample from all of $\mathbb{G}(n)$ with degrees fixed only in expectation. One such model is introduced in Chung and Lu [93] and uses vertex weights to construct a rank-1 matrix of edge probabilities (see Section 2.6.1). In an even more relaxed setting, the *degree distribution* (i.e., the relative frequencies of degrees) is prescribed by a closed-form function, the shape of which can be controlled via a small number of parameters. The most notable example of this kind are *scale-free* degree distributions, where the share of vertices of degree k is approximately proportional to $k^{-\gamma}$ for some constant $\gamma > 1$. These are obtained, for instance, from *Preferential Attachment*.

scale-free (powerlaw)
degree distribution

Preferential Attachment:

☞ Section 2.4.2

Distances

As densities and degrees are properties that involve counts and frequencies of edges, they can be easily integrated as parameters of random graph models. Prime examples of properties that give rise to more complicated structural dependencies relate to distances.

$\text{dist}(u, v)$

The (*shortest-path or graph-theoretic*) distance, $\text{dist}(s, t)$, between vertices $s, t \in V$ is the length of a shortest (s, t) -path, i.e., the minimum number of edges in any path connecting them. The distance is defined to be infinite, if the two vertices are in different connected components. In the remainder of this section we assume graphs to be connected to avoid the treatment of special cases.

remarks on connectivity:

☞ Section 2.9.3

Analogously to the average degree, the *average distance* (or *characteristic path length*) $\text{avg dist} = \sum_{s,t \in V} \text{dist}(s,t) / \binom{n}{2}$ is used as a criterion to discriminate certain graph structures. When representing, for example, social relations, the resulting graphs typically have bounded average degree, $\text{avg deg} = \mathcal{O}(1)$ and small average distance, $\text{avg dist} = \mathcal{O}(\log n)$. These features, together with a high clustering coefficient, are characteristic of *small-world models*.

small-world phenomenon

In contrast, the random spatial graphs discussed in Section 2.5.1 have large average distance (typically $\mathcal{O}(n^{1/d})$ for d -dimensional Euclidean spaces). It is however possible to control the average distance by considering different aspect ratios of the boxes used for allocating points, or by varying the underlying geometry. More strict than average distance is the *diameter* of a graph, defined as the maximum distance between any two of its vertices. There are multiple ways to define *distance sequences* [76] but, unlike degree sequences, none is commonly used in the definition of random graph models.

Graph Classes

Other graph properties are not expressed via aggregates of lower-level indices at all. For instance, a graph is called *planar*, if it can be drawn in the plane without edge crossings. While, as a consequence of this criterion, they are sparse ($m \leq 3n - 6$), there is no known parametrization in terms of the number of elements, density, degrees, or distances, that could be used for random planar graph models (see Section 2.5.4).

On the other hand, a *split graph* is a graph that has a partition into an induced clique (a complete subgraph) and an induced independent set (an empty subgraph). Split graphs are an idealized version of what is called a *core-periphery structure* [64]. Despite this definition in global terms, split graphs can indeed be characterized using degree sequences only [171].

For each class of graphs, a corresponding random graph model can be defined by the uniform distribution on its elements. Since generators for such models generally depend on specific class characteristics, we consider only random planar graphs as a particularly interesting and important example of a class that does not lend itself easily to modeling.

2.2.2 Use Cases

The most direct and, at least in computer science, most common use of graph generators is in the creation of input instances for computational experiments [234]. Such experiments typically serve to establish response curves recording levels of an outcome (dependent) variable as function of independent variables. Especially in the context of scaling experiments, independent variables are often related to the graph size (e.g., number of nodes, number of edges, average degree), or specific model parameters. Common outcome variables include implementation performance (e.g., running time or solution quality), process characteristics (e.g., spreading of information, resilience against attack), or other graph invariants (e.g., connectivity, graph spectra).

experimental algorithmics

Covering a desired experimental region with benchmark data is often impractical or even unrealistic. To control experimental factors, it is necessary that instances in a computational experiment satisfy certain constraints or show tendencies with respect to certain properties. Empirical data of sufficient coverage and variability may not be available, samples may not be sufficiently representative, or the experimental region may be too vast to store the instances explicitly. In such cases, generative models are a convenient means to fill the void [299].

In the same way that the parameters of $\mathcal{G}(n, p)$ allow to control the order and density of graphs, other models are used to generate instances that vary along a number of dimensions while ensuring a certain distribution, or the presence or absence of certain other properties. In addition to systematic variation of parameters when sampling from a distribution or construction process, there are a number of techniques for graph generation that can be used to augment a given set of benchmark data. These include sampling from models with some parameters learned from the benchmark data [65], perturbation or systematic variation of given instances [63], and scaling these instances by creating larger or smaller graphs with similar structure (see Section 2.8).

computational art

We note that the randomized generation of instances from an explicit or implicit distribution is also a technique in computer graphics and the arts [106]. In algorithmic art it has been suggested that a generated instance should be viewed as representing both itself and the ensemble from which it was sampled [257]. This artistic view is indeed related to computational model-based inference where a focal instance is compared to an ensemble of generated instances with the goal to establish whether the given instance exhibits specific features to a degree that is common or unusual for the sample [312]. A common ensemble is generated from the conditionally uniform model that assigns equal probability to all graphs with the same degree sequence (see Section 2.6). A very recent application is the randomized design of neural network architectures [346].

2.3 General Algorithmic Models and Techniques

2.3.1 Models of Computation

The scalability of a network generation algorithm is connected to the assumed model of computation. In particular, we want algorithms that run in parallel on P identical processors, and require work close to the that of a good sequential generator — at least when the generated datasets are large. In this survey, we discuss the algorithms at an abstract level, and consider whether and how they are implementable on different parallel architectures such as GPUs, shared and distributed memory.

*communication
-efficient and -free*

For distributed memory models, an important aspect are communication costs which easily dominate the overall costs when large datasets are processed. An algorithm is *communication-efficient* if the communication volume per processor is sublinear in the required local work [293]. For graph generators, we can even achieve (essentially) *communication-free* algorithms [139]. See the discussion in Section 2.5.1 for an example how to make a parallel graph generator communication-free.

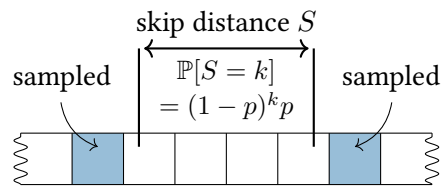


Figure 2.1: Output-sensitive Bernoulli sampling with probability p by skipping S items where $S+1$ follows the geometric distribution $\text{Geom}(p)$.

Another aspect of scalability is that large graphs may not fit into the main memory of the machine. Hence, it makes sense to consider *external memory algorithms* [7] where fast random access is limited to a bounded internal memory of size M , and the external memory is accessed in blocks of size B . An important special case are *streaming algorithms* for graph generation that output edges one at a time without requiring access to the entire graph.

external memory

2.3.2 Random Permutations

The FISHER-YATES SHUFFLE (OR KNUTH SHUFFLE) [195] obtains a random permutation of an array $A[1..n]$ in time $\mathcal{O}(n)$. Conceptually, it places all items into an urn, draws them sequentially without replacement, and returns the order in which they were drawn. The algorithm works in-place, and fixes the value of $A[i]$ in iteration i by swapping $A[i]$ with $A[j]$ where j is chosen uniformly at random from the not yet fixed positions $[i..n]$. Even if FISHER-YATES SHUFFLE seems inherently sequential, the algorithm exposes a sufficiently high degree of independence to be processed in parallel [307].

A random permutation can be computed in parallel by P processors by assigning each element to one of P buckets uniformly at random and then applying the sequential algorithm to each bucket [290]. A similar technique yields an I/O-efficient random permutation algorithm [290].

Going the opposite direction also yields a practically fast and efficient algorithm. Bacher et al. first assign each processor a contiguous section of the input array, shuffle the sub-problems pleasingly parallel and finally merge them randomly [27].

2.3.3 Basic Sampling Techniques

In this section we discuss sampling primitives, e.g., how to generate random numbers with an underlying distribution or how to uniformly take n elements from a universe.

Many of the standard distributions can be sampled in constant (expected) time with well known techniques (e.g., [105, 274]). In this survey we need geometric, binomial, and hypergeometric random deviates [314]. Respective generators are often arithmetically expensive since they require the evaluation of transcendental functions. When many deviates with known parameters have to be computed, this can be greatly accelerated by vectorization taking advantage of SIMD instructions or GPUs [292]. Often, software libraries doing this are already available.

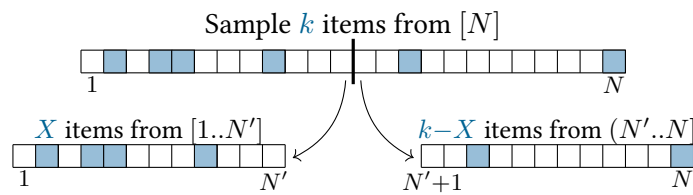


Figure 2.2: Sampling k elements from $[N]$. After splitting $[N]$ and randomly drawing X from the appropriate distribution (see Section 2.3.3.2), we can sample X and $k - X$ items, respectively, from the now independent subranges in parallel.

2.3.3.1 Bernoulli Sampling

*Bernoulli sampling by
geometric jumps*

Sampling each element from $[N]$ with probability p can be done directly in time $\mathcal{O}(N)$ by throwing N coins that show head with probability p . As illustrated in Figure 2.1, it can be made to work in time proportional to the output size by generating distances between sampled elements which have a geometric distribution [127] with parameter $1/p$. Bernoulli sampling is easy to parallelize and vectorize since all samples are independent.

2.3.3.2 Sampling k elements from $[N]$

*sampling without
replacement*

Sampling k elements from $[N]$ without replacement is possible in expected time $\mathcal{O}(k)$ using a hash table or by sampling skip distances. In contrast to Bernoulli sampling, it is however necessary to modify the parameters of each new skip distance [333]. Sampling without replacement can be parallelized in expected time $\mathcal{O}(k/P + \log P)$ using a divide-and-conquer algorithm [292]. This algorithm is based on the observation that when splitting a range of size N into subranges of sizes N' and $N - N'$ respectively (see Figure 2.2), the number of samples to be taken from the left subrange is distributed hypergeometrically with parameters k , N' , and N . This technique can be used to generate $\mathbb{G}(n, m)$ graphs (Section 2.4.1).

sampling with replacement

By using the binomial distribution instead, one can sample with replacement and this also extends to generating sets of geometric objects in Section 2.5.1. Compared to trivial sampling with replacement, the divide-and-conquer approach has the advantage to allow one processor to generate the objects in some well-defined subspace of the overall sampling space.

2.3.3.3 Rejection Sampling

*constant time
rejection sampling*

Rejection sampling is a fundamental technique (also known as *acceptance-rejection method*) to draw from a distribution \mathcal{A} which lacks an (efficient) direct sampling algorithm. It requires a second process \mathcal{B} that is easy to sample from and that “overestimates” \mathcal{A} . We first sample an element x from \mathcal{B} , and only accept x with an appropriate probability. Otherwise, we *reject* x and repeat the process. If the acceptance probability is at least a constant, the expected sampling time is of the same order as the sampling time from \mathcal{B} .

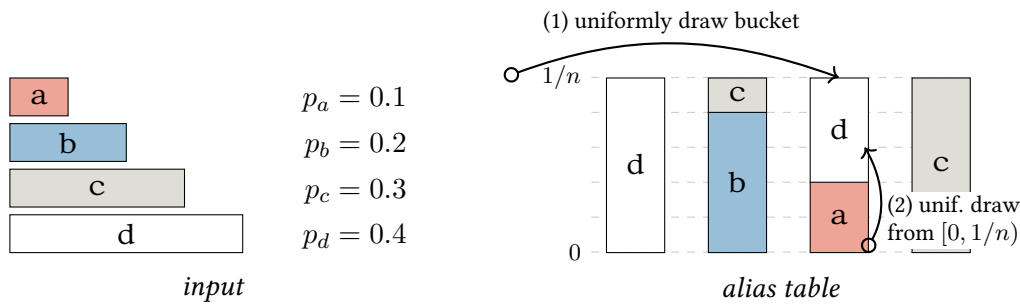


Figure 2.3: Alias table for $n=4$ elements $a, b, c,$ and d with weights $p = (1, 2, 3, 4)/10$. The table consists of n buckets each covering a probability mass of $1/n$. Each bucket contains the partial probability mass of at most two elements. Weighted sampling becomes a two-step process: first uniformly select a bucket, then select the element based on the contained partial weights.

For example, suppose we want to randomly draw from $[n]$ where i should be sampled with probability p_i and where $2 \min_j p_j \geq \max_j p_j$. Then, we can sample uniformly from $[n]$ and accept the sample with probability $p_i / \max_j p_j \geq 1/2$. This is expected to succeed with $\mathcal{O}(1)$ attempts and succeeds whp. with $\mathcal{O}(\log n)$ trials.

2.3.3.4 Weighted Sampling

Consider the case where we want to sample from $[n]$ where each element i appears w_i times. For small integer weights with $W = \sum_{i=1}^n w_i$, an array $A[1..W]$ in which element i has a multiplicity of w_i can be used. Sampling an entry from A uniformly at random yields the required distribution (cf. Section 2.4.2).

In the general case, we want to sample from $[n]$ where i should be sampled with probability p_i . There is a linear time algorithm which computes the data structure depicted in Figure 2.3. This so-called *alias table* allows sampling from a discrete distribution in constant time [339, 338]. The construction can also be parallelized [182]. The table consists of n buckets, each representing a probability of $1/n$. The probability of the elements is assigned to the buckets in such a way that each bucket is assigned the probability mass of at most two elements. conversely, the mass of some elements may be distributed over multiple buckets. Sampling then amounts to uniformly sampling a bucket i and throwing a weighted coin to decide which of the elements assigned to bucket i is to be returned.

Alias table

2.3.4 Sampling from Huge Sets

Sometimes we want to sample from a set that is much larger than the output size or the memory of the machine. For example, we will see several examples where a random graph with n nodes is defined by probabilities for each entry of an adjacency matrix which has size quadratic in n . In Section 2.3.3.1 on Bernoulli sampling we already saw how this works when all the probabilities are the same – we generate skip distances between sampled elements. Similarly, Section 2.3.3.2 explains how to do it for uniform sampling with or without replacement. Moreno et al. [250, 249] extend this approach to

sampling with a few different weights

*sampling with many
different weights*

the case when there is only a moderate number of different probabilities – use one of the above uniform sampling algorithms for each subset of elements with equal probability.

We can further generalize this using rejection sampling (Section 2.3.3.3). We partition the dataset into subsets of elements whose probabilities differ only by a constant factor. In each subset, we perform uniform sampling and use the rejection method to achieve the right sampling probability. The only prerequisite is that we can compute the exact probability for an element produced by uniform sampling. For example, consider the case where we want to perform weighted subset sampling, i.e., a generalization of Bernoulli sampling where element i has an individual probability p_i . In a subset whose probabilities are between $p/2$ and p , we can perform Bernoulli sampling with parameter p and accept element i with probability p_i/p .

Parallelizing these approaches introduces two levels of parallelism – coarse-grained parallelization over the subsets with similar probability and fine-grained parallelization within each set using the methods mentioned in Sections 2.3.3.1 and 2.3.3.2. For the coarse level, some load balancing is needed since the output sizes for each set heavily depend on the heterogeneous sizes and element sizes in each subset.

2.4 Basic Models

2.4.1 Erdős-Renyi’s $\mathcal{G}(n, m)$ and Gilbert’s $\mathcal{G}(n, p)$ models

The closely related $\mathcal{G}(n, m)$ and $\mathcal{G}(n, p)$ models were the first random graph models considered [121, 148]. They come in different variants that can all be understood as uniform sampling from an $n \times n$ adjacency matrix; see also Sections 2.3.3.1 and 2.3.3.2.

- If we sample from the whole matrix, we get directed graphs with self-loops (cf. Section 2.9.1).
- If we exclude the diagonal, we get simple directed graphs.
- If we restrict to the upper triangular part of the matrix, we get undirected graphs.
- If we exclude suitable blocks, we get bipartite graphs etc.

We obtain Gilbert’s $\mathcal{G}(n, p)$ model [148] using Bernoulli sampling with probability p for each edge independently. If we sample m edges without replacement, we get the $\mathcal{G}(n, m)$ model proposed by Erdős and Rényi [121].

*optimal sequential
algorithms*

Batagelj and Brandes [35] present sequential algorithms for these models, and point out several generalizations including a bipartite variant of $\mathcal{G}(n, p)$. For $\mathcal{G}(n, p)$ they propose the Bernoulli sampling approach described in Section 2.3.3.1 applied to the upper triangle of the adjacency matrix. This results in a runtime of $\mathcal{O}(n + m)$. For $\mathcal{G}(n, m)$ they compare two algorithm variants based on hash tables to avoid multi-edges. By now, faster algorithms are available [292, 138] that are more cache-efficient, and also work communication-free in parallel (see also Section 2.3.3.2).

communication-free

for GPUs

Nobari et al. [264] proposed a data parallel generator for the directed and undirected $\mathcal{G}(n, p)$ model. Their generators are designed for graphics processing units (GPUs). Like

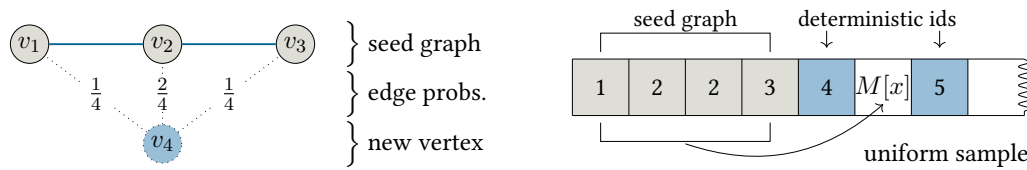


Figure 2.4: *Barabási-Albert* generator ($d = 1$) by Batagelj and Brandes. The seed graph contains nodes $\{v_1, v_2, v_3\}$ and two edges, and we introduce the new node v_4 . Since v_2 has two neighbors it is twice as likely to be chosen as the neighbor for v_4 .

the generators of Batagelj and Brandes [35], their algorithm is based on sampling skip distances but uses precomputations and prefix sums to increase data parallelism.

2.4.2 Preferential Attachment

The preferential attachment model family generates random scale-free networks. Roughly speaking, when a new vertex is added during the network generation process, it is connected to existing vertices that are chosen w. r. t. some of their properties (most often their degrees). There are multiple ways to generate graphs that follow this framework; in the following, we describe the *Barabási-Albert* (BA) model and the *Node Copy* model.

2.4.2.1 Barabási-Albert Model

Barabási and Albert [32] define the BA model. It is perhaps most widely used because of its simplicity and intuitive definition: we start with an arbitrary seed network with nodes $[v_i]_{i=1}^{n_0}$. The remaining nodes $[v_i]_{i=n_0+1}^n$ are added one at a time. They randomly connect to d different neighbors using *preferential attachment*, i.e., the probability to connect v_i to node v_j is proportional to the degree of v_j at that time. The seed graph, n_0 , d , and n are parameters defining the graph family.

Batagelj and Brandes [35] propose BB-BA, a fast and simple BA generator illustrated in Figure 2.4. For simplicity of exposition, we use an empty seed graph ($n_0 = 0$). A generalization only requires a number of straightforward index transformations. The algorithm generates one edge at a time and writes it into an edge array $E[1..2dn]$ where positions $2i - 1$ and $2i$ store the node indices of the endpoints of edge e_i (with $i \geq 1$). Since each new node has $1 \leq d < n_0$ neighbors, we let $E[2i - 1] = \lceil i/d \rceil$.

The central observation is that one gets the correct probability distribution for the other endpoint by uniformly sampling from E , i.e., $E[2dj + k]$ is set to $E[x]$ where x is chosen uniformly at random from $[1..2dj + k]$. Observe that this formulation allows self-loops and multi-edges. The former can be prevented by sampling x from $[1..2dj]$. To avoid multi-edges one can repeatedly sample until d different neighbors have been obtained. Using a hash set of size $\mathcal{O}(d)$ and a sufficiently large seed graph, this results only in an expected constant slowdown.

Meyer and Penschuck [239] propose two I/O-efficient BA generators for the external memory model and discuss generalization to various preferential models. One implements Batagelj's and Brandes' generator using *Time Forward Processing* [230] in

BA: a minimalistic graph model with preferential attachment

Preferential attachment using uniform samples from the edge list

MP-BA in external memory & on GPUs: Chapter 3

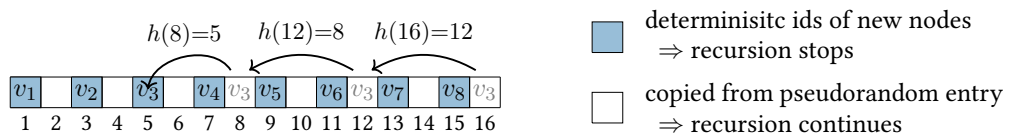


Figure 2.5: *Barabási-Albert* generator by Sanders and Schulz. Computation of the edge list’s 16th value: we compute $h(16) = 12$. Since the 12th value was computed by following $h(12) = 8$, we retrace these steps until we hit an odd position.

$\mathcal{O}(\text{sort}(m))$). The other one is based on weighted sampling using a tailor-made variant of a Buffer Tree [23, 22] and yields the same I/O complexity. The second generator is parallelized by processing subtrees individually. Potential bottlenecks near the root are avoided by processing multiple queries to the same tree node in parallel on a GPU.

communication-free by
pseudorandom hash
functions

Sanders and Schulz [294] propose a communication-free *BA* generator (see Figure 2.5) based on Batagelj’s and Brandes’ algorithm. Recall that the original algorithm computes the position $M[2i]$ by drawing a random index x and subsequently copying $M[2i] \leftarrow M[x]$. The new algorithm, in contrast, obtains a pseudorandom index $x = h(2i)$ using a random hash function h satisfying $h(i) < i$ for $i > 1$. As a result, any processor can at any time of the execution reproduce the value of x , and thereby the value of $M[x]$. If x is odd the value of $M[x] = \lceil x/2d \rceil$; otherwise, we know that $M[x]$ was obtained by copying from $M[h(x)]$ which we can retrace recursively – again without reading from M . This process terminates in expected constant time. As the algorithm never reads from M , all positions can be computed pleasingly parallel.

model generalizations

Networks generated with the *Barabási-Albert* process have a powerlaw degree distribution [32], however, no significant clustering. Several modifications have been proposed to solve this issue. Holme and Kim [177], for instance, add another parameter $0 < P_t < 1$. When adding a new vertex u with d links, the first neighbor is added as in *BA*. For each of the remaining $d-1$ neighbors a weighted independent coin is thrown: with probability $1 - P_t$ preferential attachment is used, otherwise, with probability P_t a triad formation step is carried out. If edge $\{u, v\}$ was added in the previous preferential attachment step, a neighbor w of v is selected randomly, and the edge $\{u, w\}$ is added.

Dorogovtsev and Mendes [110] propose an extreme case which is compatible with most techniques presented before. Their model starts with a triangle as seed graph. Then, rather than drawing nodes, they repeatedly introduce a new node and connect it to the two endpoints of a randomly drawn edge. The resulting graph is always planar.

The parallel generators discussed so far may emit multi-edges.¹The distributed memory generator by Alam et al. [12] (see Section 2.4.2.2) produces simple *BA* networks as a special case of the *Node Copy* model.

¹In practice, the number of multi-edges is typically small if a sufficiently large seed graph is used. Thus, deleting them does not significantly change the degree distribution.

2.4.2.2 Node Copy Model

In the *Node Copy* model [193] links to a node are added by picking a random (other) node in the graph, and copying some links from it. Similar to the BA model, we start with an arbitrary seed network consisting of n_0 nodes and add the remaining $n - n_0$ vertices one at a time. They randomly connect to d neighbors using the following process: pick an existing vertex v_j uniformly at random. Then with probability p , the new node v_i is connected to v_j (direct edge), and with probability $1 - p$, v_i is connected to a random neighbor of v_j (copy edge).

Alam et al. [12] note that the BA model is a special case of the *Node Copy* model with $p = 1/2$. While the opposite is not true, all generators discussed in Section 2.4.2.1 can be adapted to the *Node Copy* model by treating its weighted and unweighted sampling branches individually (see [239] for an in-depth discussion and more generalizations).

A shared-memory parallel algorithm has been proposed by Azadbakht et al. [26]. The authors propose an asynchronous parallel method which avoids the use of low-level synchronization mechanisms. Roughly speaking, the process of generating edges can create unresolved dependencies, i.e., the corresponding data that needs to be provided by the communicating thread to generate the edge locally has not yet been computed. The authors resolve this problem by having a thread executing two programs. The first part checks if open dependencies are resolved and if so generates the final edge. The second routine tries to generate edges and if there is a dependency the thread stores it to work on it later and continues with the next edge.

A GPU-based parallel algorithm for the *Node Copy* model has been presented by [10]. In the algorithm, each thread is responsible for a subset of the vertices. In a two phase approach, direct edges and some of the copy edges are created directly. Due to dependencies, many of the copy edges are put into a waiting queue and created in a second phase, where incomplete edges are resolved and finalized. The algorithm can generate networks from the model with two billion edges in less than 3 seconds. Alam et al. [12] transfer the algorithm into the distributed memory model and show how dependency chains, which are short in practice, are resolved efficiently in parallel. Alternatively if multi-edges are acceptable, [294] can be adapted to multi-GPU scenarios [11] yielding an even more scalable approach.

*algorithmic similarities
between BA and Node
Copy*

*all remaining schemes
retrieve non-local data
and require explicit
dependency resolution*

2.5 Random Spatial Graphs

Entities in real-world networks often have a position in the system that influences their global role in the network as well as their local neighborhood. The spatial network models presented here account for this phenomenon by associating each vertex with a point placed in some geometric space. Depending on the model, the probability of connecting two nodes is governed by (i) their relative orientation (e.g., Sections 2.5.1, 2.5.2 and 2.5.4.1) (ii) their absolute position (e.g., Section 2.5.2), or (iii) additional geometric constraints (e.g., Sections 2.5.3 and 2.5.4).

2.5.1 Random Geometric Graphs

Random Geometric Graphs (RGGs) [149, 270] are a family of random spatial graphs that project networks onto a d -dimensional Euclidean space. These types of networks have been extensively studied as models for communication networks (e.g., ad-hoc wireless networks) and the spreading of diseases [149].

model definition

RGGs are typically constructed by placing n points uniformly at random in a d -dimensional unit cube $[0, 1]^d$. Subsequently any two points x and y with $x \neq y$ are connected by an edge iff their Euclidean distance $d(x, y)$ is within a given threshold radius $r > 0$. The neighborhood of a node x thus consists of all points within the d -dimensional sphere S_x with radius r around x . For nodes near the frontier of the underlying geometry, a significant fraction of S_x can be outside the unit cube, resulting in lower degrees. Hence, some variants of RGGs use tori rather than cubes to implement

connected regime

periodic boundary conditions (e.g., Section 2.5.3). From percolation theory it is known that for constant $\alpha(d)$ the neighborhood radius $r_c \propto [\ln n / (n\alpha(d))]^{1/d}$ is a sharp threshold on the connectivity of a *Random Geometric Graph* with n points [270]. For practical purposes one assumes that n is sufficiently large and r small.

probabilistic variant

Waxman [344] introduces a generalization of RGGs that uses a probabilistic connection function. In particular, two points x , and y are connected with probability $\beta \exp[-d(x, y)/(L\alpha)]$ where L is the maximum distance between two points and $\alpha, \beta \in (0, 1]$ are parameters controlling the edge density. To be more specific, a higher value of β results in a higher edge density and small values of α increase the density of short edges relative to longer ones.

generators for geometric models often partition the space to efficiently find edge candidates

The trivial bound of $\mathcal{O}(n^2)$ work for generating a RGG can be improved under the assumption that the points are distributed uniformly at random. To this end, Holtgrewe et al. [178] partition the unit cube into subcubes with side length r . To find the neighbors of the vertices within a subcube, one only needs to perform distance comparisons between vertices within this cube and the surrounding ones. This allows the generation of RGG in expected time $\mathcal{O}(n + m)$. Holtgrewe et al. [178, 179] propose a distributed memory algorithm for the two-dimensional case based on this partitioning. Using distributed sorting and vertex exchanges between processes, they reassign vertices such that edges can be generated locally. The expected time for the local computation of their generator is $\mathcal{O}([n/P] \log(n/P))$, due to sorting. Perhaps more important for massive-scale systems is that they need to exchange all vertices which yields a communication volume of $\mathcal{O}(n/P)$ per process.

RGGs are partitioned into sub-cubes

A communication-free distributed memory generator for general d was proposed by Funke et al. [139]. Their generator recursively partitions the unit cube into smaller subcubes by computing a set of binomial random deviates. These deviates are sampled consistently across processors similarly to the approach in Section 2.3.3.2. In the end, each processor obtains a local subcube and its associated set of vertices. To generate all adjacent edges for these vertices, each processor redundantly computes the required neighboring subcubes without the need of communication. The expected number of distance comparisons on each processor then is $\mathcal{O}((m + n)/P)$.

Parsonage and Roughan [269] proposed a fast generator for generalized *Random Geometric Graphs* that uses a partitioning approach similar to the one mentioned for plain RGGs. Subcubes of the partitioning are used to derive upper bounds on the connection probability for vertices residing in them. Then the methods from Section 2.3.4 are applied to each subcube, i.e., candidate edges are identified using Bernoulli sampling which are then filtered using rejection sampling.

One can generate more structured and perhaps more realistic geometric graphs by using real-world data to restrict the locations of the points. For example, one can place points randomly along roads to model wireless car-to-car communication. To model other ad-hoc wireless networks one could define a probability distribution of points based on known data on population density. One can also change the definition of edge probabilities in an application-specific way. For example, the *radiation model* [308] has been used to sample trips between locations based on population density. This can be implemented in a scalable way [75], and the result can be viewed as an application-specific graph.

2.5.2 Random Hyperbolic Graphs

Random Hyperbolic Graphs (RHG) [200, 159] are a special case of spatial graphs in which each node has a corresponding point on a two-dimensional² disk D in hyperbolic space. Hyperbolic space allows to embed real-world graphs with low distortions [200]; this can be explained intuitively by the fact that the radius of a hyperbolic disk grows logarithmically with its area — just like the diameters of networks often grows logarithmically with their size. Related to this quirk, the hyperbolic distance $d_H(x, y)$ of points x and y does not only depend on their relative position, but also decreases as the pair moves closer to D 's center. Thus, central nodes tend to have above-average many nodes in their vicinity which often renders them network hubs.

RHG can embed observed graphs well due to their exponential expansion

In RHGs, the point set is scattered randomly onto D where the radial probability density function increases exponentially towards D 's border. The dispersion parameter $\alpha > 1/2$ controls the density of points near the center which becomes maximal as $\alpha \rightarrow 1/2$. In this case the inner $\Theta(\sqrt{n})$ points are expected to form a clique. For $\alpha = 1$, we obtain a uniform distribution onto hyperbolic space, while for $\alpha > 1$ the density near the center decreases. As a result, the central clique shrinks to expected constant size and the giant component dissolves [54].

2.5.2.1 Threshold Model

The simplest RHG variant [159] is the *threshold model*. Here, two points u and v are connected iff their hyperbolic distance $d_H(u, v)$ is below the global threshold value R . With high probability, the resulting graphs have a powerlaw degree distribution with exponent $\gamma = 1 + 2\alpha$, a controllable average degree, a non-vanishing clustering coefficient [159], poly-logarithmic diameter [134], and a giant component [54] for $\alpha < 1$.

²While empirical studies have been conducted in higher dimensions (e.g., [253]), we are not aware of widespread generative models.

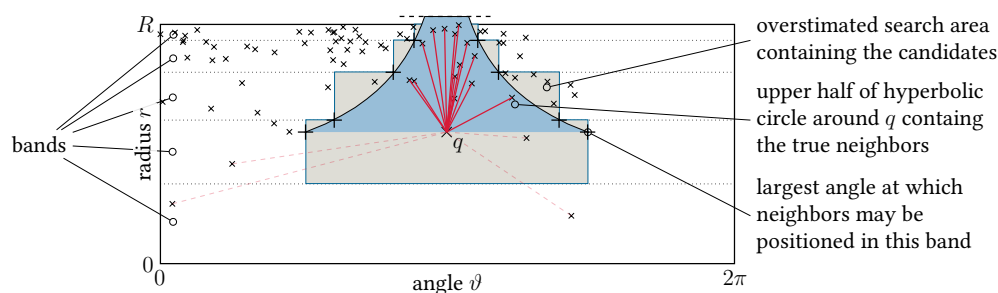



Figure 2.6: NK BAND partitions the Hyperbolic disk (here in polar coordinates) into radial bands. For each point q , it searches neighbors in the q 's own band and all above. To this end, we overestimate the upper half of the hyperbolic circle around q by computing the left and rightmost angles intersections between the hyperbolic circle and the bands.


*RHG*s are partitioned quad-tree-like or into concentric bands

Currently, the most efficient generators for the threshold model share a similar setup originally proposed by v. Looz et al. [224]. The algorithm first partitions the hyperbolic disk D along the radial axis into $\Theta(\log n)$ concentric bands. Then, it searches the neighbors Nu of each point u based on a set C_u of candidate points. As a crucial optimization, C_u contains only points with a higher radius than u itself which are computed as follows: the algorithm overestimates the upper half of the hyperbolic circle Q_u around u containing all of u 's neighbors. For each band b which include a radius identical or large than u 's, one computes the smallest and largest angle intersecting Q_u and band b , and accepts all points in between as candidates. Finally, false-positives in C_u are filtered out by rejecting all points $c \in C_u$ with $d_H(u, c) > R$. Different generators vary in the exact number of bands they use, in the bands' limits as well as how the exact computation of C_u is implemented.

NK BAND by v. Looz et al. [224] initially draws the complete point set. As illustrated in Figure 2.6, each band is effectively an array storing all points contained sorted by their angles. To compute C_u , a binary search for the left- and rightmost points is carried out in each relevant band. The authors demonstrate an empirical runtime of $\mathcal{O}(n \log n + m)$. The parallel implementation is available as part of *NetworkKit*.

HYPERGEN: sweep-line streaming generator:
 Chapter 6

Penschuck [271] proposes the streaming generator HYPERGEN using a small memory footprint with a cache-aware implementation in mind. The generator overlaps the sampling of points with the neighborhood search in a sweep-line algorithm resulting in a memory footprint of $\mathcal{O}([n^{1-\alpha} \bar{d}^\alpha + \log n] \log n)$ whp. or a time complexity of $\mathcal{O}(n \log \log n + m)$ whp..

SRHG: communication-free sweep-line:
 Chapter 7

Funke et al. [139] introduce the communication-free generator RHG as part of *KaGen*. It further sub-divides bands into cells which can be independently recomputed by all processors without communication (see Section 2.3.3). Then, for each point u the set of candidates C_u consists of all points in cells intersecting the hyperbolic circle Q_u around u . The generator requires expected time $\mathcal{O}((n + m)/P + P \log n + n(P\bar{d}/n)^\alpha + n^{\frac{1}{2\alpha}})$, where P is the number of processors. Combining the streaming technique of [271] with the communication-free sampling of [139], Funke et al. [138] propose sRHG capable of generating a graph with 2^{39} nodes and 2^{42} edges on $p = 2^{15}$ cores in less than 1 min.

2.5.2.2 Binomial Model

Similarly to *RGG*, there exists a generalization of *RHG* that replaces the sharp distance threshold of connected nodes by a distance dependent connection probability. The so-called *binomial RHG* features an additional³ temperature parameter $T \geq 0$ controlling the sharpness of the decision threshold. This gives rise to various parameter regimes [16]. For $T = 0$, we obtain the threshold model. For $T > 0$ edges between two points u and v are created with a probability that decreases exponentially with $d_H(u, v)$. In the extreme of $T \rightarrow \infty$, the model degenerates into $\mathcal{G}(n, p)$.

The first generator with sub-quadratic work is *NKQUAD* by v. Looz et al. [223]. It stores the points, which are projected on to a Poincaré disk, in a polar quad-tree. Then, the query of point u samples leaves of the quad-tree by bounding the connection probability to connect to a point in such a leaf from above. All points within a leaf are treated as candidates and randomly selected. The generator has a worst-case runtime of $\mathcal{O}((n^{3/2} + m) \log n)$. The parallel implementation is available as part of *NetworKit*.

Later, v. Looz et al. [336] improve these results using a band-based partition as in *NKBAND*. For long-ranged edges the algorithm exploits that the probability of an edge decreases monotonously as the distance between its endpoints increases. Hence, it uses a combination of geometric jumps and rejection sampling as discussed in Section 2.3.4. The resulting generator has an expected runtime of $\mathcal{O}(n \log^2 n + m)$.

Bläsius et al. [48] propose *HYPERGIRGS*, a generator with expected linear work for $T < 1$ based on the *GIRG* model. While *NKQUAD* and *HYPERGIRGS* are conceptually very similar, *HYPERGIRGS* operates in the native geometry and navigates the quad-tree more efficiently (see Section 2.5.3).

2.5.3 Geometric Inhomogenous Random Graphs

Bringmann et al. [70] propose the *Geometric Inhomogenous Random Graphs (GIRG)* model. It identifies each node with a point on a d -dimensional torus and –similarly to the *Chung-Lu* (see Section 2.6.1) model– assigns each node a non-negative weight. The probability of an edge to be added between two nodes is then a function of the points’ distance and their weights. The authors show that general *RHG*s are asymptotically contained in the ($d=1$)-dimensional *GIRG* model if hyperbolic radii are projected to *GIRG* weights, and angles are mapped onto the 1-dimensional torus.

There exists a sampling algorithm [70] with expected linear work. It decomposes the geometry similarly to a d -dimensional quad-tree and associates intervals of the node weight with the tree layers. To this end, nodes with many potential neighbors (i.e., a high weight) are placed near the tree’s root which makes them candidates to a large subset of the underlying torus. The generator arranges the tree’s leaves in memory using a space-filling curve which allows to efficiently iterate over all points contained in arbitrary subtrees.

³The original proposal [200] by Krioukov et al. already included this parameter, as well as an additional parameter ζ which does not add a degree-of-freedom and is omitted here without loss of generality.

Bläsius et al. [48] engineer the two generators HYPERGIRGS and GIRGS. While the latter samples multi-dimensional *GIRG* graphs, HYPERGIRGS is slightly modified to sample from the exact *RHG* probability distribution and can efficiently generate *RHG* instances with $T < 1$. At time of writing, the parallel implementation of HYPERGIRGS is the fastest sequential generator for the threshold and binomial model.

2.5.4 Random Planar Graphs

Planar graphs can be drawn in the plane such that no edges cross. As they are an intensively studied family of graphs, generators for random planar graphs are very interesting. Some models yield planar graphs by construction (e.g., [110], see Section 2.4.2) or can be tweaked to be planar (cf. Section 2.8.3). Further, a natural question is to ask for a uniform sample from all planar graphs with a given number of nodes n . Such graphs can be sampled in expected quadratic time. If the network size must be realized only up to a constant factor, expected linear time sampling is possible [141]. Although this algorithm has been implemented to generate graphs with 10^5 nodes, it is not very well suited for scaling to much larger graphs since heavy precomputations are needed.

Several models of planar graphs that are easier to generate have been considered [237]. The most scalable of these models consider Delaunay triangulations of random point sets (Section 2.5.4.1).

2.5.4.1 Random Delaunay Triangulations

The *Delaunay triangulation* (DT) of a point set P partitions P 's convex hull into triangles such that no circumcircle of a triangle contains a fourth point in P . This so-called *Delaunay condition* maximizes the minimal angle in each triangle and typically avoids near-degenerate triangles with very sharp angles. There are several generators that generate such a triangulation of a random point set in the unit square (or cube). This is an appealing family of instances since two-dimensional DTs can be generated efficiently. Additionally, the resulting graphs resemble meshes used in numerical computations. Indeed, DT is an important ingredient in generating meshes for numerical simulation (e.g., [305]). For example, for graph partitioning, these graphs are interesting instances since fluctuations in point densities mean that non-trivial partitions are sought that steer through thin areas of the graph.

The widely used sequential two-dimensional generator by Holtgrewe [178, 179] chooses each point independently and uniformly at random. Since the node numbering implies no locality at all, such instances are unexpectedly difficult for parallel algorithms. To expose more locality, the parallel generator by Funke et al. [139] uses the same local point generation strategy discussed in *Random Geometric Graph* (see Section 2.5.1). Moreover, to enable a simple, communication-free, and highly scalable parallelization, this generator uses periodic boundary conditions, i.e., distances are computed as the smallest Euclidean distance to any copy of the point set in x -, y -, (or z -)direction. A periodic boundary condition allows them to avoid long Delaunay edges, thereby reducing the number of recomputations.

Note that this model adaptation is also practically relevant since periodic boundary conditions appear in many scientific simulations. The generator is freely available as part of *KaGen*, and supports a three-dimensional variant of the Delaunay generator.

2.5.4.2 Planar Graphs for Infrastructures

Planar graph generation has always attracted attention of engineers because graphs underlying many types of civil infrastructure are either completely or almost planar. Examples include road networks, power grids, water distribution systems, and natural gas pipelines. Attempts to use random planar graph generators such as Plantri [72], Fullgen [71], and Markov chain based [103] have not ended up with a desired realism even if the generated graphs have been refined to better fit desired properties. Because there is a shortage of high-resolution real data for these networks (except the road networks such as the OpenStreetMap) due to various reasons, such as cost of information collection and confidentiality, generation of domain specific planar graphs is of particular interest.


Because most of these generators are developed for practical purposes, they rely on domain specific properties. For example, road network generators may consider a realistic population density modeled using clustering effects fused with geometric graph edge generation rules and subsequent edge rewiring [162] or a hierarchical city-town-village structure of nodes with domain specific proximity based rules for edges [142]. Power grids are typically spatially-defined and nearly-planar graphs. Despite the fact that many models are claimed to capture the structure of power grids without specific planarity requirement, they either require a planarization postprocessing or model adjustments to generate realistic graphs that can represent a power grid. For example, in [341], the nodes and edges are generated using various random distribution functions within small fixed areas, and proximity constraints, respectively. In [9], the *Chung-Lu* model is initializing the backbone of the graph, and random star-like structures are added to it. Both combinations seem to generate nearly planar graphs. In another domain, a water distribution system is generated by randomly concatenating small grid graphs taking into account domain specific constraints [310].

2.6 Random Graphs with Prescribed Degree Sequences

Sampling graphs with a prescribed degree sequence is a classical problem in theoretical computer science with many practical applications (e.g., as a building block in *LFR*). The task is, given a sequence of degrees $\mathcal{D} = [d_i]_{i=1}^n$, to return a uniform sample from the ensemble of all graphs where each node v_i has degree d_i . The problem is intimately related with the challenge of uniformly perturbing the edges of an existing graph which is frequently used in network analysis for hypothesis testing (e.g., [245, 154]).

The *Configuration Model* yields a linear time algorithm that may produce graphs with self-loops or multi-edges. Applications, however, often require *simple* graphs without those structures. Sampling from such an ensemble is more involved and not all degree sequences can yield a simple graph; we call the ones that do *graphical*.

LFR:
 [Section 2.7.3](#)

Configuration Model:
 [Section 2.6.2](#)
CM emits self-loops and multi-edges

CL approximates degrees:
 Section 2.6.1

*FDSM using CB, ES, and
 HAVEL-HAKIMI*

A fast approximate solution can be obtained using the model by *Chung-Lu* which realizes \mathcal{D} only in expectation. Alternatively, one can generate (potentially) non-simple graphs using the *Configuration Model* and subsequently deal with forbidden edges (see Section 2.6.2). In contrast, the frequently used *Fixed-Degree-Sequence-Model (FDSM)* directly yields simple graphs in a two-step approach. It first generates a highly biased graph in linear time (e.g., using the *HAVEL-HAKIMI* [173, 165] and engineered by [167]), and then perturbs the instance using a sufficiently long sequence of small local updates (e.g., *Edge Switching* (see Section 2.6.3) and *Curveball* (see Section 2.6.4)).

2.6.1 Chung-Lu

Chung and Lu (*CL*) [93] describe random graphs designed to match a prescribed degree sequence \mathcal{D} in expectation. *CL* graphs are parameterized by an n -dimensional non-negative vector $w = [w_i]_{i=1}^n$. In order to be realizable, w 's largest value has to be restricted to $\max_i w_i^2 \leq W$ where $W := \sum_i w_i$. Then, each node v_i is assigned the weight w_i and two nodes v_i and v_j are connected with probability $p_{ij} := w_i w_j / W$. While one typically chooses $w = \mathcal{D}$, the vector may also contain real-valued entries.

Miller and Hagberg [244] give the first efficient generator capable of producing simple *CL* instances. Their sequential algorithm requires a non-increasing weight sequence.⁴ As a result, the probability p_{ij} of an edge between a fixed node v_i and a partner v_j only decreases as j increases, i.e. we have $p_{ij} \geq p_{ik}$ for all $j < k$. Thus, after considering the edge (v_i, v_j) we can obtain the next candidate (v_i, v_k) by sampling a geometric skip distance with p_{ij} and correct the potentially overestimated probability by accepting the candidate only with probability p_{ik}/p_{ij} (cf. Section 2.3.4). The resulting generator has an expected runtime of $\mathcal{O}(n + m)$. The approach can be parallelized using the approach used in *KaGen* [139] – the adjacency matrix is partitioned into slices which can be generated independently. Appropriate load balancing has to take into account that different parts of the matrix incur a highly different amount of work.

Alam et al. [13] also require a sorted weight sequence which they collapse into N_w groups each containing nodes with identical weight. For each group only $\mathcal{O}(1)$ words are stored. Then, they conceptually decompose the adjacency matrix into $\Theta(N_w^2)$ blocks where each block is sampled as a bipartite $\mathcal{G}(n, p)$ graph. The authors show a runtime of $\mathcal{O}(N_w^2 + m)$ and demonstrate a speedup over [244] for practical weight sequences. They also give a parallel variant requiring time $\mathcal{O}((m + N_w^2)/P + P + N_w)$ where P is the number of processors.

Moreno et al. [249] unify both approaches and generalize them into an algorithmic framework suited for static graph models where each edge $\{u, v\}$ independently exists with an (implicitly defined) probability p_{uv} and the number $|\{p_{uv} \mid u, v \in V\}|$ of unique probabilities is small. Then, each group of edges with identical probabilities is treated separately. The authors describe two sampling strategies both yielding an expected sequential time of $\mathcal{O}(n + m + N_w^2)$ where N_w is the number of unique node weights.⁵

⁴This restriction can be lifted by appropriately sorting and relabeling nodes.

⁵Observe that in the common case where w contains integers, the N_w^2 terms in the previous results are

The dependence on N_w^2 can be removed by applying the techniques from Section 2.3.4. Concretely, we can subdivide the set of adjacency matrix entries into a logarithmic number of groups such that the entries within a group have probabilities that differ by at most a constant factor. Sampling within a group can then be done using a combination of Bernoulli sampling and rejection sampling.

Although this may require a small constant factor more calculations than the other approaches, it can be implemented very efficiently because the involved computations (generating skip distances and making acceptance decisions) are very simple and have high data parallelism. Thus, they can use parallelism, vector instructions, GPUs, and existing highly tuned library codes for these tasks.

2.6.2 Configuration Model

The *Configuration Model (CM)* was initially conceived for the theoretical analysis of random graphs [38, 260, 59]. Due to its simplicity, it is now also used to sample from prescribed degree sequences [247], and among others motivated the notion of *modularity*.

In the following, we consider the *Configuration Model* for undirected graphs, and introduce a directed variant in Section 2.9.1. Given a degree sequence $\mathcal{D} = [d_i]_{i=1}^n$ with $\sum_i d_i = 2m$, *CM* generates a graph $G(V, E)$ with $|V| = n$ and $|E| = m$. We first create an urn U which contains exactly d_i balls labeled v_i for each node $v_i \in V$. Then, we draw and remove two balls from U uniformly at random, and add the edge $\{u, v\}$ to E where u and v denote the labels of the two balls respectively. The process terminates when U is empty.

Clearly, *CM* allows for self-loops and multi-edges⁶. While their expected number is small for any graph G with $\max\deg(G) \ll n$, multiple strategies to obtain simple graphs have been considered:

- The *Erased Configuration Model* deletes all self-loops and multi-edges without replacement, and thereby changes the degree sequence non-uniformly. This can result in significant structural changes [296, 332].
- *Rejection sampling* repeatedly samples using the *CM*, and accepts the first simple instance obtained. This method, however, only yields expected polynomial runtimes if the maximum degree $\max\deg(G) = o(\sqrt{\log n})$ is small.
- Recent theoretical results by Arman et al. [24] (building on [143, 236]) improve upon rejection sampling by transforming a multi-graph sampled with *CM* into simple graphs — one defect at a time. The technique is very similar to the one sketched for *INC-REG* (see Section 2.9.4), but uses more switches and treats high

asymptotically dominated by the expected number of edges. Let G be a graph containing N_w different degrees and let $V' \subset V$ be an arbitrary set of nodes, s.t. all nodes in V' have a different degree. The nodes in V' are incident to at least $\sum_{v \in V'} \deg(v)/2 \leq \sum_{i=0}^{N_w-1} i/2 = \Theta(N_w^2)$ edges. Hence, we expect $N_w^2 = \mathcal{O}(m)$ which also holds whp. for $N_w = \Omega(\log n)$ by bounding the realized degrees from below using Chernoff's inequality.

⁶Due to self-loops and multi-edges, not all graphs have the same probability to be generated [260, p.436]. Hence, *CM* is not guaranteed to produce uniform samples.

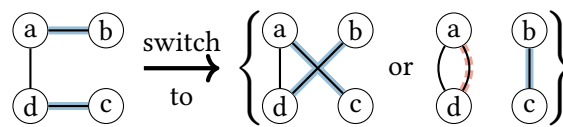


Figure 2.7: Edge Switching in an undirected graph. Two edges (here $\{a, b\}$ and $\{c, d\}$) and one of two possible new topologies are drawn uniformly at random. The swap is rejected, if the new induced subgraph is not simple (right example).

degree nodes separately. Preliminary results suggest that the algorithm can be made practical despite their complexity [18].

jump-start FDSM with CM:
 *Chapter 4*

- *Random rewiring steps* are a common technique to remove forbidden structures (e.g., [208]) in a Markov-style Las Vegas algorithm. Hamann et al. [167] use such pseudorandom edge swaps (cf. Section 2.6.3) at scale to heuristically remove the unwanted edges while preserving the original degrees.

Efficient implementations of *CM* typically exploit that by removing random balls from the urn until it is empty, the model effectively establishes a random permutation of all balls initially contained. In fact, our description of *CM* is very similar to the FISHER-YATES SHUFFLE (see Section 2.3.2). Hence, the following reformulation is equivalent, and leads to scalable generators: store the contents of urn U as a sequence S , shuffle S to obtain a permutation uniformly at random, and then interpret S as a sequence of m node pairs encoding the edge set E .

2.6.3 Edge Switching

*in each step, ES rewires
two random edges*

The *Edge Switching* (also known as re-wiring, swap, or trade) model [279, 151] applies a succession of k small perturbations, so-called edge switches, to a network. While numerous different switching types have been proposed (e.g., [323, 143]), in the interest of brevity we focus on most common variant. In case of an undirected input (see Figure 2.7), two random edges $\{u, v\}$ and $\{x, y\}$ are replaced either by $\{u, y\}$ and $\{x, v\}$, or equiprobably by $\{u, x\}$ and $\{v, y\}$. Clearly, these changes preserve the degrees of all nodes involved. If a swap violates application-specific constraints (e.g., by introducing a self-loop into a simple graph), it is rejected without replacement.

This random process is often modeled as a Markov-chain: the state space corresponds to the ensemble of all legal graphs with the same degree sequence. We connect two states iff their graphs can be transformed into each other using a single swap. For directed graphs and simple undirected graphs, the Markov chains have symmetric transition probabilities, are irreducible and aperiodic; hence, a sufficiently long sampling process converges to a uniform sample [80].⁷ While rigorous upper bounds on mixing times are only known for special graph classes and are impractically high (e.g., [123]), in practice a constant number of swaps per edge often yields sufficiently uniform results [151, 167, 80].

⁷Carstens [80, section 2.2] discusses adaptations for additional graph classes; see also [41].

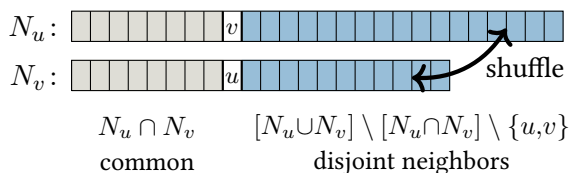


Figure 2.8: A curveball trade. The neighbors of two randomly selected nodes u, v are shuffled. Common neighbors and the edge $\{u, v\}$ must not be traded for the graph to remain simple.

In the following, we only report on the randomization of simple undirected graphs since the directed variant implies fewer constraints and is significantly easier. Swapping of simple undirected graphs requires a data structure that supports the following steps efficiently: (i) gather two random edges uniformly at random, (ii) test whether their replacements already exists (to prevent multi-edges), (iii) update the selected edges. Step (ii) implies that the neighborhoods of up to four nodes have to be considered.

Viger and Latapy [332] implement a graph data structure similar to an adjacency list where each neighborhood $N(u)$ is stored as a hash set; small $N(u)$ are kept in arrays as an optimization. Executing k swaps then requires expected $\mathcal{O}(k)$ work, but causes unstructured memory access resulting in a significant slowdown for large graphs.

Hamann et al. [167] mitigate these unstructured memory accesses with an I/O-efficient implementation executing steps (i) to (iii) in batches. By choosing a batch size of $\mathcal{O}(m)$ swaps, the previously $\mathcal{O}(m)$ unstructured I/Os can be transformed into a constant number of scans over the edge list; the resulting pipeline triggers $\mathcal{O}(\text{sort}(m))$ I/Os whp.. Its implementation is faster than [332] even for instances still fitting into main memory, and scales well for graphs exceeding this threshold.

Bhuiyan et al. [43] propose a distributed approach for P processors. Each processor P_i is assigned approximately m/P edges E_i , and generates swaps by selecting two random edges: one local edge $e_1 \in E_i$ and second (not necessarily local) edge $e_2 \in E$. If edge e_2 is stored on a different computer (i.e., $e_2 \in E_j$ with $i \neq j$), processor P_i sends a message to the remote host P_j which then executes the swap. In general, the algorithm uses additional messages to avoid multi-edges. The implementation performs $1.15 \cdot 10^{11}$ edge swaps on a network with $1 \cdot 10^{10}$ edges in 3 h using $P = 1024$ processors; [167] carry out the same experiment on a single machine ($P = 8$) with a slowdown of 8.3.

2.6.4 Curveball and Global Curveball

Curveball (CB) [321, 80] is structurally similar to *Edge Switching* as it randomizes a graph by executing a sequence of local modifications, so-called *trades*. It is available for directed⁸ and simple undirected graphs. Each trade selects two nodes $u \neq v$ uniformly at random, and shuffles their neighborhoods (see Figure 2.8). To this end, the set of disjoint neighbors $(N(u) \cup N(v)) \setminus (N(u) \cap N(v)) \setminus \{u, v\}$ is identified, and randomly redistributed between the nodes u and v while keeping their degrees unchanged. Compared to the

EM-ES: ES in EM:
 Chapter 4

In each step, CB trades the neighbors of two random nodes

⁸If self-loops are disallowed, directed triangles cannot be reorientated by *Curveball*. Several preprocessing steps have been considered to lift this restriction (e.g., the linear time algorithm [41]).

Edge Switching Markov chain, the basic steps are more complex but at the same time inflict more changes; hence empirically fewer steps are required [81]. Additionally, *CB* trades increase data locality [82].

CB and G-CB in EM:
 Chapter 5

Carstens et al. [81] (extended by [82]) propose *Global Curveball (G-CB)* which further reduces data dependencies. A global trade is a super step in the Markov chain, and consists of $\lfloor n/2 \rfloor$ single trades targeting all nodes of the graph (if n is odd, a random node remains unselected). The underlying Markov chain still converges towards a uniform sample, and in practice shows fast mixing times even for skewed degree sequences. Carstens et al. [82] introduce an I/O-efficient parallel algorithm that exploits the increased regularity of *G-CB*'s trading patterns.

2.7 Block Models

The goal of *community detection* (also known as *graph clustering*) is to identify regions of a graph that are internally densely connected, but only sparsely connected to their outsides. Such communities may be disjoint or overlapping. *Disjoint communities* partition the set of nodes of a graph into disjoint subsets. For *overlapping communities*, each node may belong to more than one community. Numerous measures and algorithms have been proposed to formalize the fuzzy concept of a community and how to detect them (see [133] for a survey).

Many observed graphs (e.g., derived from biological systems or social networks) have a significant community structure [150]. Therefore, it is natural to also consider generators that explicitly generate such structure. For the evaluation of community detection algorithms, synthetic benchmark graphs with planted communities are useful; their outputs' consist not only of the graphs produced but also includes an assignment of nodes to communities.

In the following, we introduce the traditional *Stochastic Block Model* that is also used to theoretically analyze community detection algorithms. Subsequently, we consider the commonly used *LFR* generator, and the more recently proposed *CKB* generator.

2.7.1 Stochastic Block Model

The *Stochastic Block Model (SBM)* [176, 3] (also *Planted Partition Model* or *Inhomogenous Random Graph*) is a versatile framework to model a fine-grained community structure. There exists numerous extensions and generalizations (see [212] for a recent survey). Its base variant has the following model parameter: the number n of nodes, the number k of communities, a community probability distribution $p = (p_1, \dots, p_k)$, and a symmetric matrix $P \in [0, 1]^{k \times k}$.

The *SBM* yields an undirected simple graph G and a *ground truth* community assignment $\chi: V \rightarrow C$. To this end, each node independently selects its community $c_j \in C$ weighted with probability p_j (see Section 2.3.3.4). Subsequently, we introduce an edge between each node pair $\{u, v\}$ independently with probability $P_{\chi(u), \chi(v)}$.

Observe that after assigning nodes to communities, algorithms for *SBM* and *Chung-Lu* models are similar. While in *CL* "community blocks" form endogenously if nodes

share the same weights, these blocks are expressed explicitly in *SBM*. Nevertheless, from an algorithmic point of view they can be dealt with similarly. Thus, Alam et al. [13] and Moreno et al. [249] directly generalize their respective *CL* generators to *SBM*.

2.7.2 R-MAT / Kronecker Graphs

Kronecker Graphs [213] are a family of self-similar graphs that are based on the recursive application of Kronecker products (as defined in Figure 2.9) to the adjacency matrix of an initial seed graph. These graphs obey static graph patterns, such as a powerlaw degree distribution and a small diameter [213]. Additionally, as these networks grow, their density increases. To be more specific, the number of nodes and edges obey a densification powerlaw, i.e., m is proportional to n^α for some $\alpha > 1$. The simplicity of *Kronecker Graphs* allows for the tractable analysis of various of these properties including the graph diameter, clustering coefficient and degree distribution [214, 229]. However, due to the discrete nature of the generation method *staircase effects* can be observed in some of these quantities.

To alleviate this issue, Leskovec et al. [213, 214] introduce *Stochastic Kronecker Graphs*. Instead of using an adjacency matrix, these graphs start with a probability matrix \mathbf{U} as their seed. The seed matrix \mathbf{U} can be obtained from a *deterministic* seed graph by replacing 0-entries with α and 1-entries with β where $0 \leq \alpha \leq \beta \leq 1$ are model parameters. Then, an entry $u_{ij} \in \mathbf{U}$ corresponds to the probability that an edge between the vertices i and j is present. The model then computes \mathbf{U}^k (k^{th} power of Kronecker products), and samples edges using probabilities prescribed in \mathbf{U}^k .

In order to generate *Stochastic Kronecker Graphs* that appear similar to a given graph G , Leskovec et al. [214] introduce a fast and scalable fitting algorithm called *KRONFIT*. The algorithm avoids committing to specific metrics (e.g., shape of degree distribution) by directly matching the adjacency matrix of G and the generated graph. *KRONFIT* achieves a linear running time by exploiting the structure of Kronecker products and using statistical simulation techniques.

A special case of *Stochastic Kronecker Graphs* is the *Recursive Matrix Model (R-MAT)* by Chakrabarti et al. [87]. This model is well known for its usage in the popular Graph 500 benchmark.⁹ A graph with n vertices and m edges is built by sampling each of the m edges independently. To generate a single edge, *R-MAT* partitions the adjacency matrix recursively into four equal-sized quadrants (see Figure 2.10). The quadrants are weighted with probabilities of a , b , c , and d respectively where $a + b + c + d = 1$. The model then randomly selects one of the partitions to recurse on. It continues until it reaches the base case of a 1×1 partition corresponding to the sampled edge.

Some further noise can be added at each step of the recursion to smooth out the degree distribution of the output graph. This can be done by computing a level-dependent uniform random number during the recursion, and slightly modifying the initial probabilities for each quadrant [302].

⁹The benchmark additionally requires the generator to relabel all nodes based on a random permutation of node indices to avoid information leaking that could be exploited by algorithms tailor-made for the benchmark. As relabelling is straight-forward, we omit its discussion here.

$$\otimes: \mathbb{R}^{n \times m} \times \mathbb{R}^{n' \times m'} \rightarrow \mathbb{R}^{(n \cdot n') \times (m \cdot m')},$$

$$(\mathbf{U}, \mathbf{V}) \mapsto \begin{bmatrix} u_{1,1}\mathbf{V} & \cdots & u_{1,m}\mathbf{V} \\ u_{2,1}\mathbf{V} & \cdots & u_{2,m}\mathbf{V} \\ \vdots & \ddots & \vdots \\ u_{n,1}\mathbf{V} & \cdots & u_{n,m}\mathbf{V} \end{bmatrix}$$

Figure 2.9: Definition of the Kronecker product used to generalize R-MAT.

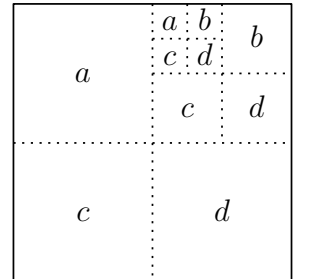


Figure 2.10: R-MAT recursively subdivides the adjacency matrix into quadrants, and each time selects one of them with probabilities a , b , c , and d respectively. The recursion terminates when a sub-matrix of size 1×1 is reached.

Note that this process produces a directed graph that may have multi-edges (which are typically converted into single edges). Furthermore, directed graphs can be made undirected by setting $b = c$, and removing all matrix entries above the main diagonal and copying the lower half (cf. Section 2.9.1).

The time complexity of the initially proposed *R-MAT* generator [87] is $\mathcal{O}(m \log n)$ since recursion has to be repeated for each edge. Furthermore, this process is embarrassingly parallel, i.e., edges can be generated independently of one another. This leads to a distributed memory algorithm with a runtime of $\mathcal{O}(\lceil m/P \rceil \log n)$. Finally, one can easily generalize the resulting algorithm to generate *Stochastic Kronecker Graphs*.

The time for generating an edge can be reduced from logarithmic to constant using bit parallelism [181]. The algorithm precomputes sequences of a logarithmic number of decisions together with their probability. These sequences can be sampled in constant time using the alias table data structure. Thus, generating an edge just amounts to concatenating a constant number of sampled sequences of decisions.

alias table:

☞ Section 2.3.3.4

2.7.3 LFR

LFR plants a known community structure

The *LFR* benchmark has been first introduced for unweighted, undirected graphs [210] and later extended to directed and weighted graphs [208]. The node degrees are drawn from a powerlaw distribution with user-supplied parameters; the community sizes are drawn from an independent powerlaw distribution. The mixing parameter μ determines the fraction of neighbors of every node u that are not part of u 's communities. The remainder of a node's degree, its internal degree, is divided evenly among all communities it belongs to.

Nodes are assigned to communities at random such that each node's internal degree is smaller than the size of the community as otherwise not enough neighbors can be found. This is done using a bipartite version of *Edge Switching* with additional rewiring steps to satisfy these degree constraints. For each community, *LFR* generates a graph with the given degree sequence using the *Fixed-Degree-Sequence-Model*. It analogously samples the global *inter-community* graph with the remaining degrees. Additional *ES* steps are used to rewire edges such that no edges of the global graph are within a community and to rewire edges that are part of multiple communities.

Edge Switching:

☞ Section 2.6.3

FDSM:

☞ Section 2.6

EM-LFR: LFR in EM:

☞ Chapter 4

The authors provide sequential implementations of the different variants of the *LFR* benchmark [210, 208]. Hamann et al. [166, 167] propose an external memory algorithm that uses I/O-efficient *ES* and a streaming implementation of *HAVEL-HAKIMI* to generate graphs. Rewiring steps are also implemented based on *ES*. If the number of communities is smaller than M , the assignment can be solved in a semi-external algorithm in time $\mathcal{O}(\text{scan}(n))$ I/Os, otherwise I/O-efficient sampling is used that needs $\mathcal{O}(\text{sort}(n))$ I/Os. The implementation of this external memory algorithm processes independent communities in parallel and outperforms the original implementation even for graphs still fitting into the internal memory. Additional parallelism can be exposed by replacing *ES* with *Global Curveball*.

2.7.4 CKB

The *CKB* model [95] is based on the *Community-Affiliation Graph Model (AGM)* [348]. *AGM* encodes the assignment of nodes to communities using a bipartite graph. Every community is similar to a $\mathcal{G}(n, p)$ graph with an individual edge probability p . As a result, nodes sharing multiple communities have a higher probability of being connected. An additional ε -community with a small edge probability ε consisting of all nodes is added [347] to allow edges between any pair of nodes. Yang and Leskovec show that this model captures many properties of real-world networks [348].

The main goal of *AGM* is not graph generation but rather fitting this model to a real graph in order to detect communities. Consequently, *AGM* provides no synthetic bipartite node-community graph, but instead uses structures taken from the observed graphs. The *CKB* model mitigates this issue, and gives the means to randomly sample the necessary parametrization for *AGM*. The number of communities a node is part of, as well as community sizes follow powerlaw distributions. The edge probability of a community c_k is given by $p_{c_k} := \alpha/x_{c_k}^\gamma$, where x_{c_k} is the number of nodes in c_k and $0 < \gamma < 1$, $\alpha > 0$ are parameters.

The original implementation of *CKB* uses Apache Spark [350]. It first generates the two degree sequences for nodes and communities on a master node, and then sends them to the other worker nodes. The number of edges for each community is sampled from a binomial distribution. In order to compensate for multi-edges and self-loops, *CKB* calculates their expected number and samples more edges. The generator then uses the *Erased Configuration Model* to sample the assignment of nodes to communities, and to generate the individual community graphs. For both steps, each worker emits edge subset of similar sizes, which are then distributed and merged using the Hadoop Distributed File System (HDFS). On Amazon EC2 with 100 m1.large instances (two virtual cores, 7.5 GB RAM each, which might be a 2010 Intel Nehalem server processor [191]), the authors are able to generate a graph with one billion nodes in less than 2 hours [95].

2.8 Graph Replication

The practical goal of *graph replication* methods is to generate graphs that are similar to one or more reference graphs. We already discussed a simple variant in Section 2.6, where the goal is to sample graphs with matching degree sequences. However, depending on the application, other or additional properties are of importance. Graph replication is typically used for tasks, such as algorithm testing, performance evaluation, benchmarking, and assisting decision makers in various domains including (but not limited to) engineering, software design, epidemiology, and viral marketing.

In such domain dependent tasks, a similarity between the synthetic and original graphs is often not well-defined. Thus the designers of graph generators have to be careful when quantifying the *realism* of a synthetic graph. In some cases, such realism is measured in specific structural properties of paths, loops, special forms of network backbones at various resolutions, and connectivity between node clusters.

Hence traditional general properties (cf. Section 2.2) seem insufficient to generate a useful synthetic network. For example and as discussed in Section 2.8.3, the preserved properties may include the second shortest path length [161], or the graph Laplacian [306].

2.8.1 BTER

The *Block Two-Level Erdős-Rényi (BTER)* model generates graphs approximately following a prescribed degree sequence and clustering coefficient per degree [196].¹⁰ Its input parameters are $[d_i]_{i=1}^{d_{\max}}$ and $[c_i]_{i=1}^{d_{\max}}$ where d_i specifies the number d_i of nodes with degree i and c_i requests that all nodes of degree i to have a clustering coefficient of c_i . Both quantities are only realized in expectation.

BTER generates graphs in two steps. In the first phase, it constructs *homogeneous affinity blocks* by grouping together $d+1$ nodes of degree d in a greedy fashion. The remaining unassigned nodes form so-called *heterogeneous affinity blocks* of mixed degrees (treatment omitted here, refer to [196]). Each homogeneous affinity block is materialized as a $G(d+1, p)$ graph where p is chosen to match the requested clustering coefficient c_d in expectation. Unless an affinity block forms a clique, the prescribed degrees are not met and additional edges need to be added in a second phase. To this end, *BTER* computes for each node $v \in V$ its excess degree e_v as the difference between its requested degree and the degree expected from phase 1. It then supplements the missing edges with a *CL* graph on the degree sequence $\mathcal{D} = [e_v]_v$. Ultimately the resulting graph is constructed by merging the subgraphs from both phases.

Kolda et al. propose a parallel sampling algorithm [196]: each worker independently adds edges by first randomly selecting for which phase the edge is introduced. Then the node pair is drawn from the appropriate distribution. This process almost surely generates duplicate edges. To avoid this bias, more edges are generated to account for the expected losses during de-duplication. Similarly, the number of nodes with degree 1 is scaled by a correction factor $\beta > 1$ to compensate that approximately 36 % of them remain singletons while 28 % receive at least two neighbors [113].

Based on their efficient sequential and parallel implementations of *CL*, Alam et al. [13] also present efficient sequential and parallel implementations of *BTER*. They are able to generate a graph with 131 million nodes and 4.6 billion edges in 210 seconds sequentially and just 0.37 seconds using 1024 processors, i.e., they achieve a speedup of about 572.

GBTER [66] (*Generalized BTER*) generalizes *BTER* by allowing the user to supply the groups of the nodes instead of automatically grouping nodes by degree. This enables the generation of a graph with a prescribed community structure. The user can also supply a density per group, replacing the automatically assigned density to match a certain clustering coefficient.

EGBTER [119] (*Extended GBTER*) adds support for clustering coefficients in addition to the user-supplied groups. *EGBTER* takes a community assignment $\chi: V \rightarrow C$, the groups, the expected within-community degree of each node $[d_i]_{i=1}^n$, the expected

¹⁰The original formulation [301] did not explicitly consider the clustering coefficient.

global degree of each node $[D_i]_{i=1}^n$, with $d_i \leq D_i$, and the within-community clustering coefficient distribution $[c_i]_{i=1}^{d_m}$. For the generation process, the authors essentially run *BTER* for every community separately and then generate an *CL* graph for the remaining degrees. Thus, scalable implementations of *BTER* can be easily adapted for *EGBTER*.

A-BTER [311] (*Adapted BTER*) uses a specially configured version of *BTER* to generate benchmark graphs for community detection similar to the *LFR* benchmark. At the same time, *A-BTER* replicates the degree and clustering coefficient distribution of a given graph similarly to *BTER*. Using a heuristic scaling mechanism, *A-BTER* slightly adapts these distributions to match a prescribed average and maximum degree and clustering coefficient on a possibly larger graph. *A-BTER* takes a mixing parameter that, like in the *LFR* model, denotes the fraction of inter-cluster edges. Using a linear program, *A-BTER* further adjusts the degree and clustering coefficient distributions such that the resulting graph matches the mixing parameter while keeping the adjustments minimal. The assignment to affinity blocks, and the edge generation closely follows the *BTER* model. Using the parallel edge-skipping technique [13], the actual intra- and inter-community edges are generated efficiently in a parallel, distributed implementation. The implementation is based on MPI and OpenMP. On 512 compute nodes where each node has 128 GB RAM and two marvel ThunderX2 ARM processors with 28 cores per processor, *A-BTER* generates a graph with 925 billion edges and 4.6 billion nodes in 76 seconds. While the resulting graphs do not perfectly match the original *LFR* model, they show that community detection algorithms show similar behavior. Further, they show that both the degree and clustering coefficient distributions as well as the mixing parameter are as desired.

2.8.2 Darwini

The *Darwini* [116] generator takes both a degree distribution, and a distribution of clustering coefficients per degree as input. Usually, such distributions would be obtained from an observed network. Using this input, *Darwini* can generate a graph of a different scale that is similar to a real-world network from which the distributions have been used. The authors show that it outperforms other approaches like *BTER* in terms of the reproduced metrics on the Facebook social graph.

Darwini first samples for every node a degree, and then a clustering coefficient from the distribution of that degree. The basic idea of *Darwini* is then to group nodes into buckets, where all nodes in a bucket have the same (or a similar) desired number of triangles in order to achieve their clustering coefficient. For each bucket, a $\mathcal{G}(n, p)$ graph is generated. Bucket sizes and probabilities p are chosen such that the expected number of triangles in each bucket matches the desired number while ensuring that no node in the bucket can exceed its target degree. After this first step, each node has a residual degree left. *Darwini* iteratively adds edges between buckets until it achieves the target degrees. Roughly speaking, it is unlikely that these inter-bucket edges create new triangles. Thus, this step allows to both reach target degrees while keeping the target clustering coefficients. In each iteration, the algorithm attempts two strategies for

creating edges. Firstly, from each node, it attempts to create an edge to a random node, which succeeds if the other node has a non-zero residual degree. Secondly, it shuffles the nodes into small groups and attempts adding edges between nodes within a group. For edges within a group, the algorithm favors edges between nodes of similar degree.

The authors provide an open source implementation of *Darwini*¹¹ in Apache Giraph¹². While the implementation carries out the assignment to buckets on a central compute node, it distributes the creation of edges within buckets while processing each bucket sequentially. Globally uniformly distributed edges are generated in parallel while the random groups are again aggregated to generate the edges within the group. *Darwini* introduces an additional layer of super-communities to generate graphs larger than the main memory of the compute cluster. It is executed individually for each super-community, and then later edges between nodes in super-communities are generated by sampling potential neighbors and testing if they still want new neighbors. The authors generated a scaled up version of the entire Facebook social graph with 3 trillion edges in 7 hours on a compute cluster of 200 machines each with 256 GB RAM and 48 cores.

2.8.3 Multilevel generators

Multilevel algorithms for computational optimization on graphs are well known to be successful on such problems as partitioning [288], linear arrangement [286], force-directed drawing [340], and vertex separator [164]. The main idea behind these algorithms is to create a hierarchy of increasingly coarse representations in which each next-coarser graph is structurally similar but smaller than the current level coarse representation. While constructing a hierarchy, an optimization problem is solved for each coarse representation (from the coarsest level to finest) in a such way that a coarse level solution serves as an approximation for the next-finer level. Here we describe how the multilevel algorithm design pattern can be used for graph generation.

The multilevel¹³ graph generator *MUSKETEER* [161] deems the following two observations important to generate realistic replica: Firstly, various graph properties —while being different in absolute terms— are as important on the coarse levels as on the original finest level. For example, both clustering coefficient and diameter, can be illuminating at the coarse levels, when the finest scale information is aggregated after coarsening. Secondly, applying a single node or edge editing operation (such as adding/removing a node or adding/removing/rewiring an edge) at a coarse level would be equivalent to applying many operations on the whole regions in a graph at the finest level. On the one hand, this will contribute to the realism of a generated graph because the local structure will be preserved if changes are applied at the coarse levels.

Conversely, if the changes are applied only at the fine levels, the global structure of a graph will remain unchanged. The user can therefore control the levels where editing

¹¹<https://issues.apache.org/jira/browse/GIRAPH-1043>

¹²<https://giraph.apache.org>

¹³In the original paper, the authors used the term *multiscale* to emphasize that the generator acts at multiple scales of a complex system coarseness. Here we replace it with the term *multilevel* to avoid possible associations with scalable performance.

operations should be applied thereby controlling the desired deviation from the original graph at both local and global resolutions.

MUSKETEER follows the same coarsening scheme as in multilevel approaches for combinatorial optimization problems, such as partitioning and linear ordering [286, 288]. However, it edits synthetic graph preserving the required properties at each level during the uncoarsening phase instead of optimizing some objective. Throughout all levels, all editing operations are local. For example, in [161], a distribution of the second shortest path length measured at each level during the coarsening is preserved when a new node is added and connected to other nodes. The nearly linear complexity of this approach is comparable to that of other multilevel optimization algorithms. Reported preserved metrics include degrees, assortativity, eccentricity, clustering, betweenness centrality and harmonic mean distance centrality.

The multilevel approach can also generate planar graphs [89] by applying linear time planarity test (for Kuratowski subgraphs [325]) followed by rejecting added edges that violate planarity at each level of coarseness. The graph editing at each level is required to only produce planar graphs.

Indirectly, the multilevel approach attempts to preserve the spectral properties of an original graph either at the user specified, or all levels of coarseness. Conversely, Shine and Kempe propose *SpectralGen* [306] to intentionally produce graphs with a similar (or matched) spectrum of the graph Laplacian. The spectrum of the Laplacian encodes high-level connectivity information about the original graph, and can be controllably preserved within the multilevel frameworks using advanced coarsening schemes [90].

SpectralGen acts as follows: Firstly, it generates a template matrix with a spectrum close to the original graph Laplacian using a randomly sampled orthonormal basis followed by a linear programming based fitting. Since the result is not necessarily a correct graph Laplacian, *SpectralGen* subsequently rounds the matrix to a valid graph Laplacian using linear programming. The approach is currently prohibitive expensive for large-scale graphs. There is no theoretical guarantee that the generator samples uniformly from the set of graphs with approximately correct spectra. The authors, however, experimentally demonstrate a significant variation in the generated graphs while preserving useful metrics including betweenness centrality and path length distribution.

One of the interesting future challenges is combining other generative approaches with the multilevel framework by applying them at different levels of coarseness. For example, one may want to generate a graph with specific global geometry at the coarse levels while addressing local properties, such as clustering coefficient or degree distribution, at the fine levels only.

2.8.4 dK -Graphs

Mahadevan et al. [228] propose dK -graphs, a family of models parametrized by d . The parameter controls the level of details captured from the input graph:

- For $d = 0$, only the average degree is kept.
- For $d = 1$, the degree distribution is reproduced analogously to *FDSM*.
- For $d = 2$, for every pair of degrees the number of connections between nodes of these degrees is preserved.
- For $d > 2$, for every possible induced subgraph with d nodes, the occurrences of a certain d -tuple of degrees is preserved.
- For $d = n$, the input graph is replicated exactly.

FDSM:
 Section 2.6

The authors report that they obtain graphs very similar to an internet topology graph supplied as input already for $d = 3$. Given an input graph, they use a restricted version of *Edge Switching* that only allows switches that preserve the considered distributions. Given just the desired properties and no input graph, the authors consider edge switches that bring them closer to the desired property. They report no running times, but report that the algorithmic complexity increases sharply with increasing values of d . However, for $d = 2$, it should be possible to adapt efficient implementations for *Edge Switching*. For $d > 2$, it seems unlikely that such graphs could be generated efficiently as an increasing number of edge switches would need to be rejected and there are no obvious ways to select them directly.

2.9 Additional Graph Types

Applications often mandate special types of networks: street networks, for instance, are *directed* to account for lanes or one-way-streets, and may express the distance between two points as *edge weights*. In the following, we outline modifications of previously discussed models and generators to cater to such use cases. For the sake of brevity, we focus on general techniques and selected examples we no intention of completeness.

2.9.1 Directed Graphs

Algebraically, a digraph is a binary $n \times n$ adjacency matrix A . For an undirected graph, the matrix A is symmetric with an empty diagonal and, thus, fully determined by either its upper or lower triangle matrix.¹⁴ Therefore, the set of all digraphs with n nodes is exponentially larger than its undirected counterpart. As a direct consequence, random graphs models need to be extended to properly deal with digraphs.

In the simplest case the two edges (u, v) and (v, u) are treated independently (e.g., as in the directed variants of *Erdős-Rényi* and *Gilbert*). Bollobàs et al. propose a directed

¹⁴Many generators, e.g., *ER* (Section 2.4.1), *R-MAT* (Section 2.7.2), or *RHG* (Section 2.5.2), exploit this fact by only sampling from one triangle matrix. The other triangle matrix then follows due to symmetry.

Preferential Attachment. In each iteration, their model randomly selects one of three actions: add an edge with a new node as tail, add an edge with a new node as head, or connect two existing nodes. When selecting a random edge source, node v is chosen with probability linear in $\deg_{\text{out}}(v)$. Analogously, edge heads are selected based on the nodes' in-degrees. Most of the *PA* algorithms for the undirected case carry over by treating edge heads and tails separately. The main observation is that node u with in-degree d_u^{in} appears exactly d_u^{in} times as an edge head, i.e., at odd position in the edge array E (and analogously for out-degrees). Hence, random edge tails are copied from uniformly selected even indices, and edge heads from uniformly selected odd positions.

Boguña et al. [55] propose a directed random graph model influenced by *RHG*. It captures causality relations on cosmological scales, i.e., where the speed-of-light limits what is observable. Nodes have a random position in space and time, and a directed edge (u, v) is added if u can be observed by v .


The notion of a degree sequence $\mathcal{D} = [d_i]_{i=1}^n$ can also be extended to digraphs by replacing each degree by a 2-tuple $\mathcal{D}_{\text{directed}} = [(d_i^{\text{in}}, d_i^{\text{out}})]_{i=1}^n$. Then, the *Configuration Model* uses two independent urns – one for edge heads, one for edge tails. Similarly, the *FDSM* model is available for digraphs. It is the building block for a directed variant of the *LFR* benchmark which otherwise draws and processes the in- and out-degree sequences independently. The necessary modifications to the deterministic *HAVEL-HAKIMI* generator, *Edge Switching*, and even *Curveball* are mostly obvious. Here, one notable exception is that the Markov chains of *ES* and *CB* are not irreducible on the ensemble of simple digraphs because the direction of a directed 3-cycle cannot be reversed [80]. One solution is a linear-time preprocessing step by Berger and Carstens [41].

2.9.2 Weighted Graphs

We model *weighted networks* by augmenting a graph $G = (V, E)$ with weights $w: E \rightarrow \mathbb{R}$, and define the *strength* $s(u)$ of node u as the sum of weights of all edges incident to u . The semantics of *edge weights* w are domain specific; examples range from energy flows in food webs, over capacities in transportation networks, to costs or distances in street maps. For positive integer weights, $w(e)$ can also be interpreted as the multiplicity of edge e – this is especially sensible when modeling capacities along edges [259]. Here, a multiplicity $w(e) = 0$ encodes the absence of edge e .

The arguably simplest way to obtain a weighted graph is to assign random edge weights drawn independently and identically distributed to a random graph. This is a common theme in both theoretical (e.g., shortest path, minimum spanning trees, or first passage percolation) and empirical studies (e.g., *LFR* variant [208]).

The *Weighted Random Graph* model (*WRG*) [145] is a maximum entropy model proposed as the weighted variant of $\mathcal{G}(n, p)$. The distribution $WRG(n, p')$ is defined over $\mathbb{G}(n)$ and assigns each potential edge e the multiplicity $w(e)$ drawn from the geometric distribution $\text{Geom}(p')$. Since any edge e exists with $\mathbb{P}[w(e) > 0] = 1 - p'$, the *WRG* model is topologically equivalent to $\mathcal{G}(n, p)$ with $p = 1 - p'$. The generators discussed in Section 2.4.1 thus carry over with little modifications.

TFP-BA and MP-BA support such graphs:
 Chapter 3

Preferential Attachment:
 Section 2.4.2

Configuration Model:
 *Section 2.6.2*

Stochastic Block Model:
 *Section 2.7.1*

In real networks, however, the assumption of an independence between topology and edge weights does not hold. Serrano and Boguñá, for instance, observe non-trivial correlations between a node's degree and strength [300]. The authors then analyze the *Configuration Model* where the number of balls for each node follows a powerlaw degree sequence with small exponent $|\gamma| \ll 3$. They derive the distribution of degrees and strength in the limit of $n \rightarrow \infty$, show that both follow different powerlaw distributions, and demonstrate non-trivial correlations. Instances of this model can be sampled by counting¹⁵ the multiplicities emitted by a standard *CM* generator. The *Stochastic Block Model* can be extended in the same spirit (e.g., [205]).

Britton et al. [73] propose another weighted *CM* variant. It features a second family of distributions assigning each ball a weight in addition to the number of balls for each node. It then only pairs balls of equal weights; if the number of balls with a given weight is odd, a random ball is dropped. This model can be efficiently implemented, by treating each weight class as an independent urn.

In spatial graphs, a distance function induced by the underlying geometry is a natural choice for edge weights. For example, one could augment *RGs* with edge weights that give the Euclidean distance of the connected points.

Other, application-specific solutions have been studied. Hyun-Joo et al. [192], for instance, consider a finance network where each node corresponds to a market player. The agents are augmented with latent variables indicating their performance. Then weights are computed based on latent variables of their endpoints.

2.9.3 Connected Graphs

random regular graphs:
 *Section 2.9.4*

Few of the previously discussed models guarantee that the generated graphs are connected. Preferential attachment models are a note-worthy exception as they stay connected if the provided seed graph is. Additional well known models with efficient generators include *Random Delaunay Triangulation* and the Watts-Strogatz model [342]. Another source of very sparse and connected graphs are random regular graphs.

There are three general techniques to obtain a connected graph from an existing model. In the following we consider $\mathcal{G}(n, p)$ graphs. They undergo phase transitions as p is increased [122]. In the extreme of $np > (1 + \varepsilon) \ln n$, the graphs are connected with high probability. In such cases *rejection sampling* leads to efficient generators. Here, one generates graphs G_1, G_2, \dots and returns the first G_i that is connected.

Yet, $\mathcal{G}(n, p)$ almost surely yields isolated nodes for $np < (1 - \varepsilon) \ln n$. This renders rejection sampling an unsuitable choice in this parameter region. Another approach is to identify the largest connected component C of a randomly drawn network and to remove all nodes and edges not contained (or incident with) C . This selection process can introduce biases and must not preserve properties of the original model. Additionally, the number n' of nodes of the resulting graph is a random variable.

In order to be efficient, n' should be sufficiently close to the number n of nodes emitted by the original random graph model. This is the case for models featuring a large

¹⁵Also known as *word counting*. It can be efficiently computed in most practical machine models.

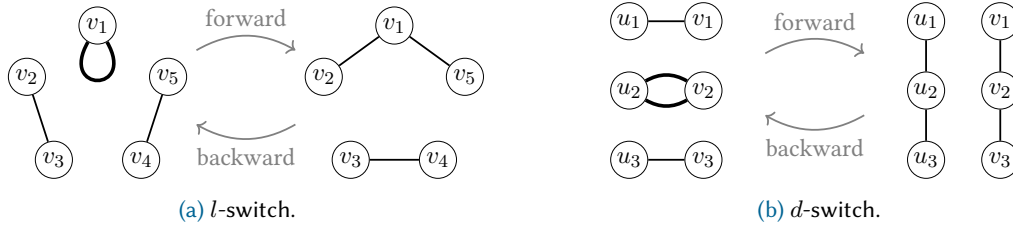


Figure 2.11: The l -switch removes a single loop; d -switch remove a double edge. The remaining nodes are selected such that no new loop or double edge is created.

giant component.¹⁶ While there exist general characterizations for the presence of a giant component (e.g., based on a graph's degree distribution [122]), many random models have more precise predictions. *Threshold RHG*, for instance, has a giant component of size $\Omega(n^{1-2\alpha})$ whp. rendering filtering for small values of α efficient. The same is true for $\mathcal{G}(n, p)$ if np is sufficiently close to $\ln n$.

Inversely to filtering, adding edges can lead to connectivity. A simple oblivious method is to add a random spanning tree (e.g., [285]). Alternatively, one can selectively introduce bridges to iteratively merge disjoint connected components.

The edges of a prescribed connected graph can be shuffled using *Edge Switching*. Here, the Markov chain is the normalized, induced subgraph of the original Markov chain discussed in Section 2.6.3, where the state space is reduced to connected graphs. Mihail et al. [151] show that this modified version still eventually leads to a uniform stationary distribution and discuss characterizations of compatible degree sequences. Viger and Latapy [332] further engineer the *ES* for connected graphs.

2.9.4 Regular Graphs

A graph G is said to be r -regular for $r \in \mathbb{N}_{>0}$ if all nodes have degree exactly r [59, 345]. This implies $r < n$ and that $n \cdot r$ is even. In the following, we denote the set of all r -regular graphs with n nodes as $\mathbb{G}^{(r)}(n)$ and the uniform distribution over it as $\mathcal{G}^{(r)}(n)$.

Observe that $\mathbb{G}^{(r)}(n) \subset \mathbb{G}(n, m)$ since all graphs in $\mathbb{G}^{(r)}(n)$ contain $m = n \cdot r/2$ edges. However, regular graphs appear relatively infrequently [38] with a ratio $|\mathbb{G}^{(r)}(n)|/|\mathbb{G}(n, m)| = \Theta[\exp(\frac{1-r^2}{4})]$ in the limit of $n \rightarrow \infty$. As a result, $\mathbb{G}^{(r)}(n)$ exhibits substantially different properties compared to $\mathbb{G}(n, m)$; for instance, a random $G \in \mathcal{G}^{(r)}(n)$ for $r \geq 3$ is almost surely r -connected¹⁷ [59]. In contrast, a uniform sample $G \in \mathcal{G}(n, m)$ is 1-connected whp. only for an average degree of $\Omega(\ln n)$ (cf. Section 2.9.3). This renders $\mathbb{G}^{(r)}(n)$ of particular interest to obtain random connected graphs of small constant degree r . Additionally, almost all $\mathbb{G}^{(r)}(n)$ with $r \geq 3$ are Hamiltonian [284].

To sample from $\mathcal{G}^{(r)}(n)$ all techniques discussed for the *Fixed-Degree-Sequence-Model* (see Section 2.6) carry over since $\mathbb{G}^{(r)}(n)$ can be fully characterized by the degree sequence $\mathcal{D} = [r]_{i=1}^n$. Most prominently, rejection sampling over the *Configuration*

¹⁶For constant $c > 0$, a connect component $C \subseteq V$ in $G = (V, E)$ is a *giant component* if $c|C| > |V|$.

¹⁷A graph is r -connected if it remains connected after the removal of any $r-1$ nodes.

Model (see Section 2.6.2) is efficient for small constant values of r .

McKay and Wormald [236] propose DEG, a more versatile scheme for $r = \mathcal{O}(n^{1/3})$. Rather than rejecting all non-simple graphs emitted by *CM*, the generator also accepts some multi-graphs. The authors show that this selection yields a constant acceptance probability. Then, DEG iteratively removes loops and double edges using the two switching types illustrated in Figure 2.11 – in each iteration one switch is selected uniformly at random to remove exactly one defect. Since these switches cause a bias, the emitted graphs are not uniformly distributed on $\mathbb{G}^{(r)}(n)$ anymore. To counteract this bias, each switch may be rejected with small probability.¹⁸

The generator DEG was improved twice culminating in INC-REG. In a first step, Gao et al. [143] introduce additional switches which do not decrease the number of defects but allow for tighter bounds on the rejection probability. INC-REG [24] additionally allows triple edges in the multi-graph and adds new switches to remove them. The authors further accelerate the switching process by simplifying the way nodes participating in a switch are sampled. As a result, an illegal switch (e.g., one introducing multi-edges) can be selected. In this case the computation is restarted, effectively splitting the rejection step of the DEG into two simpler steps. INC-REG has an expected runtime of $\mathcal{O}(rn + r^4)$ for $r = o(\sqrt{n})$ which is optimal for $r = \mathcal{O}(n^{1/3})$.

2.9.5 Threshold Graphs

Threshold graphs were introduced by Chvátal and Hammer [94]. Mahadev and Peled [227] discuss numerous characterizations and applications of threshold graphs. Just like their superclass of split graphs, they can be defined in terms of their degree sequence. Alternatively, they can be obtained by iteratively adding nodes that are either connected to all previously added nodes (dominating nodes) or none of them (isolated nodes). By randomizing the decision what kind of node to add, we immediately obtain a random graph generation algorithm that is linear in the size of the generated graph. Generating n nodes such that dominating and isolated nodes are chosen with equal probability yields a uniform distribution on the set of unlabeled threshold graphs of n nodes [107]. Due to the simplicity of this algorithm, it can be transferred to a distributed setting without communication. For every node, we consistently flip a coin to determine if it is a dominating or isolated node. If it is dominating, we emit edges for all node ids that are smaller than u . To also generate edges to nodes with larger node ids, we need to repeat the decision for all nodes v with larger node id and emit an edge if v is a dominating node. As the probability of dominating nodes is 0.5, this does not increase the running time in expectation.

¹⁸The bias is because, in general, the number of multi-graphs with ℓ loops (k double edges) does not match the number of multi-graphs with $\ell-1$ loops ($k-1$ double edges). Thus, there are fewer switches that decrease the number of loops (or double edges) than the inverse operation [236]. As a countermeasure, the rejection probability is derived from the ratio of forward- and backward-switches. In fact, it is over-estimated as the exact computation is too expensive. Since each switch is selected uniformly at random and independently of the rejection step, the process eventually yield uniform samples from $\mathbb{G}^{(r)}(n)$ – even if restarts occur more frequently than required.

Table 2.2: List of publicly available implementations sorted by name of the toolkit. Abbrev.: *BA*: Barabási-Albert, *ER*: Erdős-Rényi, *ES*: Edge Switching, *FDSM*: Fixed-Degree-Sequence-Model, *RDT*: Random Delaunay Triangulation, *RGG*: Random Geometric Graph, *RHG*: Random Hyperbolic Graph, *SBM*: Stochastic Block Model, *WS*: Watts-Strogatz, *MMod*: Machine Model, *SEQ*: Sequential, *SHM*: Shared-Memory, *DM*: Distributed Memory, *Py*: Python

Toolkit	Url & Models	Language	MMod
Implementations of Multiple Models			
GraphTool	https://graph-tool.skewed.de · <i>ES, RDT, SBM</i>	C++	SHM
GTGraph	http://www.cse.psu.edu/~kxm85/software/GTgraph · <i>ER, R-MAT</i>	C	SEQ
IGraph	https://igraph.org/ · <i>BA, ER, ES, SBM, WS</i>	C++, Py, R	SEQ
KaGen	https://github.com/sebalamm/KaGen · <i>BA, ER, RDT, RGG, RHG</i>	C++	SHM, DM
NetworkX	https://networkx.github.io/ · <i>BA, Caveman, ER, Holme-Kim, LFR, RGG, SBM, WS</i>	Python	SEQ
NetworKit	https://networkkit.github.io/ · <i>BA, CL, Clustered Random Graphs, ER, FDSM, Pub-Web, RHG, R-MAT</i>	C++, Py	SHM
Snap	https://snap.stanford.edu/snap · <i>BA, CM, Forest Fire, Multiplicative Attribute Graphs, Node Copy, R-MAT</i>	C++	SHM
Implementations of a Single Model			
Darwini	https://issues.apache.org/jira/browse/GIRAPH-1043 · <i>Darwini</i>	Java	DM
FEASTPACK	https://www.sandia.gov/~tgkolda/feastpack/ · <i>BTER</i>	MATLAB	SEQ
GIRGs	https://github.com/chistopher/girgs · <i>GIRGs, RHG</i>	C++	SHM
Graph500	https://graph500.org/ · <i>R-MAT</i>	C	DM
HyperGen	https://github.com/manpen/hypergen · <i>RHG</i>	C++	SM
LFR	https://sites.google.com/site/andrealancichinetti/files ·	C++	SEQ
MUSKETEER	https://github.com/sashagutfraind/musketeer · planar version: https://github.com/isafro/Planar-MUSKETEER	Python	SEQ
R-MAT	https://github.com/lorenzhs/rmat · <i>R-MAT</i>	C++	SHM

2.10 Software Packages

In this section, we aim to give a short overview over publicly available software packages as well as implementations of single models. In principle, we try to avoid historic generators that are not widely used anymore. We focus on tools that either can generate a wide-range of models and for those we report all the models that we covered within this survey, or software that is specialized on a single model. An overview can be found in Table 2.2.

2.11 Future Challenges

Generating graphs remains a widely open field for future research. It is an interesting question to what extent the multitude of algorithms that we sketched in this survey can be improved further or how techniques outlined can be used to derive algorithms for new or other models not discussed here. We believe that there are plenty of open problems.

Parallelism and Hardware Issues

As current parallel machines are able to run billions of threads, scalable graph generation remains an open problem for many models. This becomes even more pronounced for supercomputer systems with millions of processors that often are hierarchically organized (e.g., in islands, racks, nodes, CPU sockets, cores, and threads). These hierarchies and heterogeneity make the implementations highly complicated. One way to tackle this problem may be to design more algorithms that are either communication-free or communication-efficient. Still even sequential algorithms are often hard to get scalable. A quite obvious challenge is to make not-yet-scalable graph generators scalable, either by clever engineering or simplifying the model. For example dK -graphs are an interesting model, but there is little known how difficult it is to generate them. While scalable implementations for $d = 2$ seem possible, $d = 3$ might be an interesting challenge.

Numerical Stability

Many generators (e.g., based on hash functions) use fewer random bits or a smaller pseudorandom state than required to allow the creation of every possible instance. While implications of this issue are well understood for a number of random combinatorial objects (e.g., in case of random permutations [289, 195]), it is an open question if graph properties of interest are unbiased in such cases. We expect that using fewer random bits yields a trade-off between coverage and efficiency – however, this needs thorough investigation from both the theory and practical side. This is particularly consequential for hypothesis testing, where a generator is to create an unbiased ensemble.

Another possible source of bias are numerical instabilities which typically occur during floating point operations. While moderately sized instances can be sampled with default standard library arithmetic/special functions, it gets challenging for huge distributed instances. A simple approach to overcome this issue would be to use arbitrary precision libraries which is often practically infeasible. Alternatives include to explicitly manage the errors (e.g., [46]), or to use efficient and exact sampling methods as demonstrated by [67] for *ER* and *CL*.

Models vs. Applications

There is a gap between research on scalable graph generation algorithms and the applications in which they are used. Often, the domain specific properties a model should reflect are highly confounded, poorly formalized, or not even fully understood. In these cases, the models described here correspond to rather idealized situations in which many details have been stripped away. While they may be sufficiently close for benchmarking of algorithms, they are unlikely to be suitable for statistical modeling [152, 312].

An example of more expressive models are *ERGMs* (*Exponential-Family Random Graph Models*) [226] common in social network research. Given, say, graph statistics s_1, \dots, s_k and an associated vector $\theta = (\theta_i)_{i=1, \dots, k}$ of parameters, a graph $G \in \mathbb{G}(n)$ is assigned probability $P_\theta(G)$ defined as follows:

$$P_{\theta}(G) = \frac{1}{Z(\theta)} \exp\left(\sum_{i=1}^k \theta_i \cdot s_i(G)\right) \quad \text{with} \quad Z(\theta) = \sum_{H \in \mathbb{G}(n)} \exp\left(\sum_{i=1}^k \theta_i \cdot s_i(H)\right)$$

Statistics are chosen based on theories regarding micro-level mechanisms that may explain the formation of a network. They typically range from the number of edges, to counts of various other kinds of small subgraphs. The *Exponential-Family Random Graph Models* with the edge count statistic $s_1(G) = m$ and parameter $\theta_1 = \ln \frac{p}{1-p}$, for instance, is equivalent to $\mathcal{G}(n, p)$.

Because of the potential generality and complexity of distributions, sampling from models such as *Exponential-Family Random Graph Models* is usually done using Markov-Chain-Monte-Carlo algorithms. These are of limited scalability, and can benefit from algorithmic contributions towards more efficient updates of statistics after each step of the Markov chain. Limited scalability is also an issue for parameter estimation, a common task in network modeling. Here, new approaches seem to be necessary to make such models scale to larger graphs (see, e.g., [88, 319]).

Moreover, many application domains suppose inter-dependencies between graph structures and other attributes of nodes and edges, as well as multiple types of edges and the evolution of graphs over time. The identification of recurring principles and algorithmic building blocks may be one of the most important challenges in this area.

Libraries and Portability

It would be highly helpful to have well maintained libraries that are able to work on different models of computation including GPU, HPC, and Big-Data-Tools such as MapReduce [101], Spark [350], or Thrill [44]. In principle, given the same random seeds and parameters of the model, the libraries should be able to guarantee that the graph generated is the same independent on the underlying platform that is used. Then, different researchers will be able to perform experiments on the same graphs on very large machines without the need to transfer and manage huge amounts of data. Moreover, such libraries could include plug and play algorithms like dropping random edges, union, dynamization, et cetera. The obvious way to deal with portability would be (de)serialization of data and writing to disk (which is often a bottleneck). It remains to be answered what data structures are a good fit in this case, or how to partition the data in distributed generators.

Acknowledgements

We would like to thank the organizers and the staff of the Dagstuhl seminar 18241 on “High-Performance Graph Algorithms” during which the initial idea for this survey was formed. We are also grateful to Matthieu Latapy for his valuable comments and pointers. As usual in algorithmics the order of authors is alphabetical with the exception that we moved Manuel Penschuck to the front since he coordinated this project, and contributed a more than proportional amount of material.

Generating Massive Scale-Free Networks under Resource Constraints

joint work with U. Meyer

Random graphs as mathematical models of massive scale-free networks have recently become very popular. While a number of interesting properties of them have been proven, huge instances of such networks actually need to be generated for experimental evaluation and to provide artificial datasets. In this paper, we consider generation methods for random graph models based on linear preferential attachment under limited computational resources and investigate our techniques using the well known Barabási-Albert (BA) graph model. We present the first two I/O-efficient BA generators, MP-BA and TFP-BA, for the *External Memory Model* and then extend MP-BA to massive parallelism based on but not limited to GPGPU. Our simple and easily generalizable sequential TFP-BA outperforms a highly tuned implementation of the sequential linear-time BB-BA algorithm by Batagelj and Brandes by several orders of magnitude once the graph size exceeds the available RAM by only 2 %.

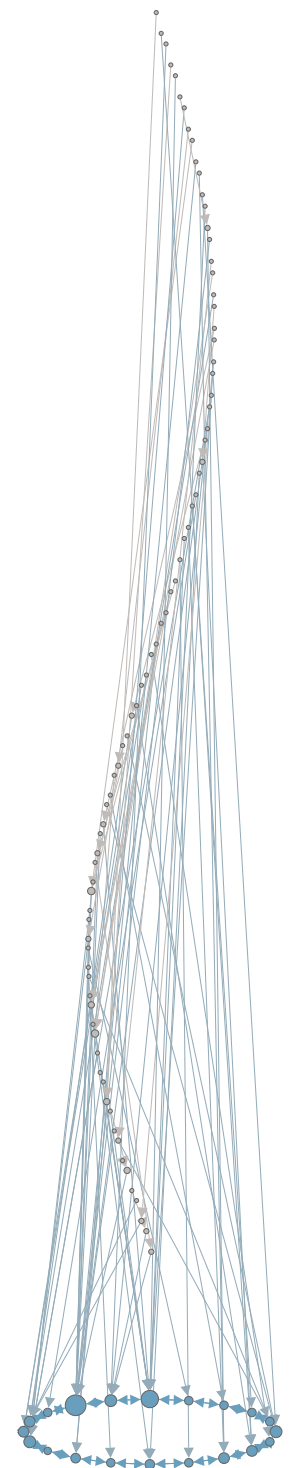
An implementation of MP-BA targeting heterogeneous systems with CPUs and GPUs is 17.6 times faster than BB-BA for instances fitting in main memory and scales well in the EM setting. Both schemes support a number of features in more general preferential attachment models, e.g., seed graphs exceeding main memory, vertices with random initial degrees, the uniform sampling of vertices, directed graphs and edges between two randomly chosen vertices. Compared with previous studies on computer clusters, MP-BA yields competitive results and already poses a viable alternative using only a single machine.

This chapter is based on the peer-reviewed conference article [239]:

- [239] U. Meyer and M. Penschuck. Generating massive scale-free networks under resource constraints. In M. T. Goodrich and M. Mitzenmacher, editors, *Workshop on Algorithm Engineering and Experiments ALENEX*, pages 39–52. Society for Industrial and App. Math. SIAM, 2016. doi:10.1137/1.9781611974317.4 .

My contribution

I am the main author of this paper and its implementation.



Barabási-Albert graph
 $n = 100, |V_0| = 20, d = 1$
 New nodes are added on top.

3.1 Introduction

Search engines, social networking sites, e-commerce platforms and other businesses regularly keep their real-world data in the form of graphs. Similarly, the internet or biological networks can be viewed as graphs. Many of these networks can be mathematically modeled as random graphs to prove certain structural properties.

As a consequence, a multitude of different random graph models have been invented [36]. In order to support the appropriateness of a concrete model experimentally or to supply realistic test data to study algorithms, one needs to be able to generate large random instances according to the model's rules. We introduce two non-approximating primitives to support linear preferential attachment in a parallel setting for graphs that do not fit into main memory. Due to its simplicity and the availability of previous results, we focus on the popular Barabási-Albert (BA) graph model which yields scale-free networks, i.e., graphs with a powerlaw degree distribution [32]. Batagelj and Brandes gave a simple linear time BA generator referred to as BB-BA [35].

With data volumes growing much faster than a typical user's computing infrastructure, however, the sequential algorithm by Batagelj and Brandes becomes challenging to apply. While it might be tempting to use some commercial cloud service with huge main memory, logistic issues will often force users or companies to rather keep their data locally and either stick to their existing hardware or seek for a moderate cost-effective upgrade. As a consequence, significant parallelism is often restricted to the usage of GPGPU and/or the input data will typically not completely fit in the main memory (RAM) of the computer system at hand but has to reside on external storage as hard disks. External-memory (EM) algorithms [242, 335] are especially tuned for this setting. At first glance, BB-BA seems far from being extendible to either GPGPU or the EM setting since it appears to be inherently sequential and suffering from highly unstructured random memory accesses. Nevertheless, we manage to transfer the main idea of BB-BA to the EM setting by applying a dynamically developing hierarchy of structured data access patterns. Together with lazy processing, this data hierarchy facilitates the EM generation of BA graphs in sorting complexity and also reveals sufficient parallelism to yield interesting speedups with GPGPU.

3.1.1 External Memory Model

We consider the commonly accepted *External Memory Model* of computation by Aggarwal and Vitter [7]. It assumes a two-level memory hierarchy with fast internal memory with a capacity to store M data items (e.g. vertices or edges of a graph) and a slow disk of infinite size. In an I/O operation, one block of data, which can store B consecutive items, is transferred between disk and internal memory. The measure of performance of an algorithm is the number of I/Os it performs. The number of I/Os needed to read/write N contiguous items from/to disk is $\text{scan}(N) = \Theta(N/B)$. The number of I/Os required to sort N items is $\text{sort}(N) = \Theta((N/B) \cdot \log_{M/B}(N/B))$. For all realistic values of N , B and M , $\text{scan}(N) < \text{sort}(N) \ll N$. Sorting complexity constitutes a lower bound for most non-trivial EM tasks.

M : main memory size

B : block size

scan

sort

3.1.2 Barabási-Albert Preferential Attachment Model

The BA method builds random graphs sequentially: given an (often small) seed graph $G_0(V_0, E_0)$, new vertices are subsequently connected to $d \leq |V_0|$ existing vertices. Since every vertex initially has a known vertex degree, we call this form of growth *structured vertex insertion* as opposed to random initial degrees which we refer to as *unstructured*. The neighbors are randomly chosen with probabilities proportional to their current degrees. This vertex selection is referred to as *linear preferential attachment*.

For $d > 1$, the neighbors of a new vertex can either be chosen independently, i.e., each with the same probability distribution, or sequentially such that earlier selections influence the probability distributions of later neighbors. Since in the former case, all edges are selected at a point in time when the new vertex still has degree zero, self-loops are only possible in the latter variant. We use the parameter $\sigma = 1$ to indicate sequential selections with possible self-loops, and $\sigma = 0$ for independent sampling.

Note that both processes can connect a new vertex to an old one via two or more parallel edges [61]. Depending on the size of the seed graph, this affects only a small fraction of edges but may still be undesired for certain applications. In this case, common strategies are to either directly reject the edge and to keep on trying to sample a new neighbor until it is unique or to remove all generated parallel edges in a post-processing phase without any replacement. In this paper, we allow the generation of parallel edges.

3.1.3 Review of the BB-BA Algorithm

The linear time sequential algorithm of Batagelj and Brandes iteratively fills m edges into an array $Q[1 \dots 2m]$ where an edge occupies the entries of two consecutive array positions. After the generation of m' edges for the first n' vertices these are stored in the first $2m'$ array positions. If by then a vertex with index $1 \leq v \leq n'$ has accumulated degree d_v , label v appears d_v times in $Q[1 \dots 2m']$. Therefore, when generating the i -th edge for the next vertex with index $n' + 1$, the respective neighbor can be determined as $Q[r]$ where r is a random number uniformly drawn from $\{1, \dots, 2(m' + i - 1)\}$, i.e., $Q[2(m' + i) - 1] := n'$ and $Q[2(m' + i)] := Q[r]$. The beauty of this approach lies in its simplicity and the fact that a standard random number generator for increasing number ranges suffices. Its application in an external memory setting with $m \geq 2M$, however, would produce $\Omega(m)$ I/Os with high probability due to the unstructured dependent accesses to $Q[\cdot]$. As shown in Figure 3.8, this reduces the algorithm's performance by orders of magnitude and renders it inadequate for graphs exceeding main memory.

3.1.4 Related Work

Sequential RAM implementations of the BB-BA algorithm (or variants of it) appear in a number of network analysis frameworks such as *NetworkX* [163] or *NetworKit* [316]. In this context, Atwood et al. present a generator for generalized non-linear preferential attachment based on augmented heaps and treaps [25]. For advanced models of computation there are various generation algorithms that do not strictly follow the Barabási-Albert generation rules and hence only yield some approximation. For

$G_0(V_0, E_0)$ seed graph
 d : insertion degree

(un)structured insertion

linear preferential attachment

σ : sequential selection flag

removal strategies for parallel edges

we allow parallel edges ($\sigma = 1$)

Q : edge list

example, a distributed memory parallel generator of this type was given by Yoo and Henderson [349] and an approximate generator based on MapReduce/Hadoop has been proposed in [221]. An efficient exact distributed memory parallel algorithm has been given by Alam et al [12]. It applies the more general random graph copy model [203] and relies on the efficient parallel resolution of dependency chains which are shown to be short on average. The authors run experiments on a cluster with up to 768 cores. A similar way to identify hidden parallelism in sequential algorithms has recently been used for other graph problems in [307]. Very recently, Sanders et al. proposed an yet unpublished pleasingly parallel algorithm which requires no communication during its main loop. [294] It eliminates random accesses to $Q[\cdot]$ by repeatedly applying a suited hash function until an odd index or a seed edge was found. Due to the structured vertex insertions in BA such values can be trivially computed (or looked up when the seed area is hit). For unstructured vertex insertions and seed graphs exceeding main memory, the algorithm in its current form requires sorting complexity in the EM model and a replication of the vertices' degrees in a distributed scenario. We are not aware of any BA generator results on external memory or GPGPU.

3.1.5 Our Contributions

We present the first two I/O-efficient BA generators, MP-BA and TFP-BA, for the *External Memory Model* of computation. In Section 3.2 we introduce our flexible, easily implementable, sequential TFP-BA algorithm. We then propose the hierarchical MP-BA design which decomposes the task into independent subproblems, which can be processed pleasingly parallel. TFP-BA outperforms a highly tuned implementation of BB-BA by several orders of magnitude once the graph size exceeds the available RAM by only 2%. An implementation of MP-BA targeting heterogeneous systems with CPUs and GPUs is 17.6 times faster than BB-BA for instances fitting in main memory and scales well in the EM setting; it poses a viable alternative compared to the results reported by a previous BA generator using a cluster of computers. In Section 3.6 we generalize both algorithms to a number of different preferential attachment models.

3.2 The Sequential TFP-BA Algorithm for EM

In a first step, we aim to transfer the main idea of BB-BA, namely the simple random number generation for increasing ranges, to the external memory setting. Our sequential TFP-BA algorithm is based on the observation that the generation of queries to random indices can be completely decoupled from the materialization of the actual edges. This allows us to track the dependencies between edges and to move all queries to $Q[x]$ to the unique point in time at which $Q[x]$ is written. Observe that in BA an edge i will only request information from some earlier edge $j < i$. The dependency graph is therefore acyclic and we are free answer queries long before the result is required.

LINK token

QUERY token

In order to perform this lazy evaluation, TFP-BA captures all information (implicitly) used by BB-BA when accessing $Q[\cdot]$ by using two types of tokens, LINK (create edge) and QUERY. Since we consider tokens to be edge-based, we in fact use two different

Time Forward Processing:

 *Section 4.2.3*

BA's dependency graph

Algorithm 1: TFP-BA: base version without optimizations of Theorem 3.1.

Input : $G_0(V_0, E_0)$ seed graph, $N \geq |V_0|$ vertices in final graph, $D \leq |V_0|$ edges per new vertex, $\sigma \in \{0, 1\}$, $\sigma = 1$ for self-loops
Output: Edge list E of BA-PA random graph

```

1 initialize (EM) prio. queue PQ ordering tokens lexicographically with LINK < QUERY-x
2 M ← 1
3 foreach {u, v} ∈ E0 do // Phase 1
4     PQ.ENQUEUE(⟨M, LINK, u, v⟩) // Copy seed graph
5     M ← M + 1
6 for u ← |V0| + 1 . . . N do // Phase 2
7     M' ← M // Generate query tokens
8     for i ← 1 . . . D do
9         R ← rand_uniform({1, . . . , M' + σ(i - 1)})
10        x ← rand_uniform({1, 2})
11        PQ.ENQUEUE(⟨R, QUERY-x, M, u⟩)
12        M ← M + 1
13 while not PQ.EMPTY do // Phase 3
14     T ← PQ.REMOVETOP()
15     if T == ⟨M, LINK, u, v⟩ then
16         LAST_EDGE ← (u, v) // Output edge
17         E ← E ∪ {{u, v}}
18     else if T == ⟨i, QUERY-x, j, u⟩ then
19         PQ.ENQUEUE(⟨j, LINK, u, LAST_EDGE[x]⟩) // Lookup last edge and reinsert
    
```

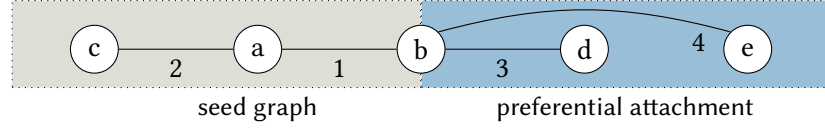
$QUERY-x$ token with $x \in \{1, 2\}$ where $x = 1$ requests the first vertex incident to an edge and $x = 2$ the second one respectively. Most importantly, each token stores an edge index i which encodes the moment when the data related to the token becomes available in $Q[\cdot]$, i.e., the creation of an edge has the same index i as all queries to it.

QUERY-x token

Following the *Time Forward Processing* technique [91, 230], we employ an off-the-shelf EM priority queue PQ which enforces a lexicographic order over the tokens' indices and types with $LINK < QUERY-x$. PQ places all queries to an edge directly after its creation and the algorithm only needs to retain a copy of the last edge created in order to answer such requests. A query token additionally contains all information required to produce the edge it is intended for. Thus, once the query is processed, a link token with an appropriate edge index is used to transport this data "forward in time" to the point where the edge is supposed to be materialized. This delay is necessary in case queries to the new edge exist – which is not revealed until TFP-BA reached this point. As shown in Algorithm 1, TFP-BA consists of three phases:

1. Let $G_0(V_0, E_0)$ be the seed graph with arbitrarily ordered edges $E_0 = \{e_1, \dots, e_{m_0}\}$. For every edge $e_i = \{u, v\} \in E_0$ enqueue the token $\langle i, LINK, u, v \rangle$ into PQ .
2. For each new vertex u and each of its d edges enqueue $\langle i, QUERY-x, j, u \rangle$ into PQ

Figure 3.1: Application of TFP-BA. Similar steps are illustrated as one.



Sorted token sequence after second phase

$$\underbrace{\langle 1, \text{link}, a, b \rangle}_{\text{seed graph}} \leq \underbrace{\langle 1, \text{qry2}, 3, d \rangle}_{\text{random query}} \leq \underbrace{\langle 2, \text{link}, a, c \rangle}_{\text{seed graph}} \leq \underbrace{\langle 3, \text{qry1}, 4, e \rangle}_{\text{random query}}$$

1. Create $\{a, b\}$ as edge 1 | Last edge: (a, b)
 ~~$\langle 1, \text{link}, a, b \rangle$~~ $\langle 1, \text{qry2}, 3, d \rangle \leq \langle 2, \text{link}, a, c \rangle \leq \langle 3, \text{qry1}, 4, e \rangle$
2. Insert token with last edge's 2nd vertex | Last edge: (a, b)
 ~~$\langle 1, \text{qry2}, 3, d \rangle$~~ $\langle 2, \text{link}, a, c \rangle \leq \langle 3, \text{link}, b, d \rangle \leq \langle 3, \text{qry1}, 4, e \rangle$
3. Create $\{a, c\}, \{b, d\}$ as edges 2 and 3 | Last edge: (b, d)
 ~~$\langle 2, \text{link}, a, c \rangle$~~ ~~$\langle 3, \text{link}, b, d \rangle$~~ $\leq \langle 3, \text{qry1}, 4, e \rangle$
4. Insert token with last edge's 1st vertex | Last edge: (b, d)
 ~~$\langle 3, \text{qry1}, 4, e \rangle$~~ $\langle 4, \text{link}, b, e \rangle$
5. Create $\{b, e\}$ as edge 4 | Last edge: (b, e)
 ~~$\langle 4, \text{link}, b, e \rangle$~~ Empty PQ: Done

where the running edge index equals $j = |PQ| + 1$, $x \in \{1, 2\}$ is a random flag, and i a random integer drawn uniformly at random from an increasing range as described for BB-BA.

3. While PQ is not empty repeat: remove PQ 's smallest token and execute one of the following actions based on its type:
 - $\langle i, \text{LINK}, u, v \rangle$: output $\{u, v\}$ as the i -th edge and retain a copy $\text{last_edge} \leftarrow (u, v)$ for later queries.
 - $\langle i, \text{QUERY-}x, j, v \rangle$: query the x -th vertex incident to the i -th edge, i.e., let $u \leftarrow \text{last_edge}[x]$. Enqueue the new token $\langle j, \text{LINK}, u, v \rangle$ into PQ .

Theorem 3.1. Let m_0 be the number of edges in the seed graph and m the number of generated random edges. Then TFP-BA requires $\mathcal{O}(\text{scan}(m_0) + \text{sort}(m))$ I/Os, and has a time complexity of $\mathcal{O}(m_0 + m \log m)$. ◀

Proof. The algorithm is dominated by priority queue operations; all other instructions are asymptotically negligible. Every edge is the result of a constant number of token operations: for seed graph edges one token is inserted/removed. For random edges two tokens are used. Thus, using a suited EM priority queue [291], TFP-BA requires $\mathcal{O}(\text{sort}(\tilde{m}))$ I/Os and time $\mathcal{O}(\tilde{m} \log(\tilde{m}))$ where $\tilde{m} = m_0 + m$.

Since there are no dependencies between seed edges, they can be permuted arbitrarily. Hence, the tokens from the first phase do not have to be sorted using PQ . We rather skip the first phase and generate the link tokens on the fly in the third phase by scanning through the input and merging it with PQ . ◻

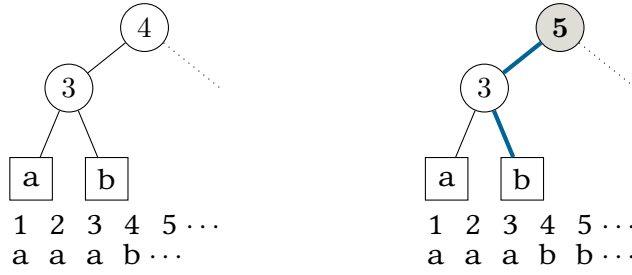


Figure 3.2: Query token $\langle 4, d \rangle$ traverses the decision tree on the left and eventually yields edge $\{b, d\}$. The “sorted edge list” is illustrated below the trees. The figure on the right shows the updated weights taking the new edge into account.

3.3 The Parallel MP-BA Algorithm for EM

While TFP-BA works well in the EM setting, it seems that its main loop can hardly be parallelized due to the unstructured reinsertions and token dependencies. Additionally, it relies on a generic sorting algorithm which does not exploit problem-specific features.

Thus, we need to reorganize the algorithm. As illustrated in Figure 3.2, our MP-BA algorithm maintains a “sorted edge list” in which all occurrences of a vertex are grouped together into a single segment. While the original adjacency information is lost in this setting, the permutation does not influence the probability to choose a given vertex when uniformly sampling an index. Each group of identical vertices can be collapsed into a single vertex annotated with its degrees, i.e., its random mass that needs to be accounted for. This alone reduces the memory consumption from $\Theta(m)$ to $\Theta(n)$ but inhibits constant time sampling. To efficiently query and update the structure, MP-BA recursively splits the sequence into independent subranges. At the lowest level of this hierarchy, a subrange consists of a single vertex index augmented by its weight.

MP-BA interprets edge list Q as urn with two labeled balls per edge

3.3.1 The Decision Tree

In the following, we first describe a sequential internal memory solution in order to present the high-level ideas and then refine it to yield the desired I/O-performance and parallelism. Let $n = 2^D$ for some natural number $D \geq 1$. We maintain a full binary tree T where each inner tree node t keeps a counter W_t for the *weight of its left subtree*. The k -th leaf of T (ordered from left to right) corresponds to the vertex¹ with index k and has a weight representing the degree $\deg_G(k)$ of vertex k in G . This value does not have to be stored since it is already encoded in the decision tree T .

T : search tree storing the urn. The i -th leaf keeps degree of v_i .

W_t : weight of left subtree rooted in t

Now consider the situation where vertices $1, \dots, n'$ with their m' incident edges have been created and we turn to the new vertex with index $n' + 1$. For generating its i -th incident edge let r be a random number uniformly drawn from $\{1, \dots, 2(m' + \sigma(i - 1))\}$ where $\sigma = 1$ if self-loops are to be produced and $\sigma = 0$ otherwise. As illustrated in Figure 3.2 and shown in Algorithm 2, we can identify the corresponding target vertex by traversing T starting from the root in a binary search like fashion: compare r with

¹We use the term *node* only for the decision tree while *vertex* always refers to the generated graph.

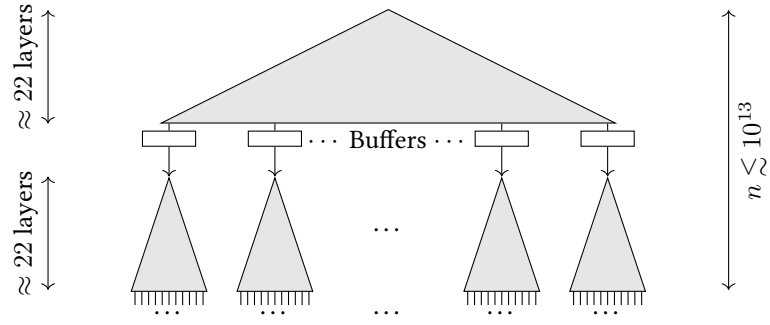


Figure 3.3: Tree structure with buffers. For realistic values of n , M and B , a single buffer layer suffices for graphs with $n \lesssim 10^{13}$ nodes.

anticipate the resulting edge and update the tree's weights accordingly already during search

W_t for the current inner tree node t . If $r \leq W_t$ then proceed to the left tree child and increase W_t by one. If $r > W_t$ reduce r to $r - W_t$ and descend into the right subtree. Eventually, the search will end in some leaf of T with index k and the edge $\{n' + 1, k\}$ is appended to the generator's output. Subsequently, the weight of the $(n' + 1)$ -th leaf has to be incremented by increasing W_t of all its parents where $(n' + 1)$ is in the left subtree. For $\sigma = 0$, we directly initialize T such that the weights of all leaves are d .

3.3.2 I/O-Efficiency

Theorem 3.2. Let n_0 be the number of vertices in the seed graph, n' the number of random vertices, and m' the number of random edges generated. Then MP-BA requires $\mathcal{O}(\text{sort}(n_0 + m'))$ I/Os. \blacktriangleleft

Proof. Assuming that we are provided the degrees of all vertices in the seed graph $\mathcal{O}(n_0)$ updates of the tree structure suffice before the beginning of the random phase which will then cause $\mathcal{O}(m')$ operations. Let $n = n_0 + n'$ be the maximal number of items stored in the tree and $m = n_0 + m'$ the asymptotic number of operations issued. So far, our new approach does not improve upon the I/O-performance of BB-BA, it might even take up to $\Omega(m \log(n))$ I/Os.

However, as illustrated in Figure 3.3, two standard modifications will improve it to $\mathcal{O}(\text{sort}(m))$ I/Os. In order to be able to delay the tree traversal, we again represent a query using a bufferable token $\langle \text{query}, r, u \rangle$. The first modification introduces a FIFO buffer of size B before each inner tree node and only starts processing queries to that node once its buffer is full. In this case, $\mathcal{O}(1)$ I/Os suffice to flush its whole content to the buffers of its two children, which might trigger recursive flushing operations there. This modification alone reduces the I/O complexity to $\mathcal{O}(m/B \cdot \log n)$.

The second standard trick is to increase the outdegree of the inner tree-nodes from two to $c \cdot M/B$ and the buffer sizes from B to $c \cdot M$ for some appropriately chosen constant $0 < c < 1$. Each inner node now keeps a little tree of depth $\mathcal{O}(\log(M/B))$ with $\Theta(M/B)$ counters and with a buffer for $\Theta(B)$ tokens at each of its leaves. In that way, the height of T is greatly reduced without harming the efficiency of the block transfers, yielding the desired $\Theta((m/B) \log_{M/B}(n/B)) = \mathcal{O}(\text{sort}(m))$ I/O-bound. \square

Using Buffer Trees, the same asymptotic bounds can be achieved [23, 22] – however with higher constants. Since only nodes with positive degree can be sampled, singleton vertices in the seed graph can be removed. The number of edges therefore constitutes an upper bound of the number of vertices. We will drop the distinction between initialization and generation phase for the remainder of the chapter and carry out further analyses only in terms of n and m .

3.3.3 Balancing

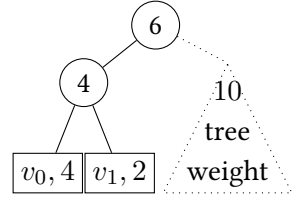
For our first parallel approach (see Section 3.3.4.1), we require that the probability mass in any two subtrees rooted in the same layer is *similar*. So far, this cannot be satisfied since the random numbers’ domain is increased with each new edge.

Hence initially only the first n_0 leaves, stored in the left-most subtrees, are reachable at all. The effect is further amplified due to the skewed degree distribution of the resulting graph in which vertices with lower indices are expected to have higher degrees. This issue is omnipresent in the parallel processing of graphs with powerlaw degree distributions and is commonly approached using techniques such as round robin- or randomized PU assignments [12, 349].

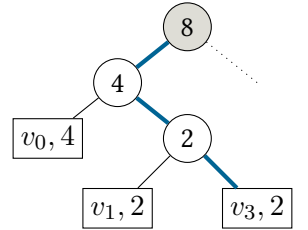
We present a dynamic scheme more natural to our hierarchical design. It introduces an additional token class, namely *vertex tokens*, that carry the index and a suitable initial degree of a vertex. These tokens are used to grow a balanced decision tree rather than starting with a full binary tree with n leaves: given the seed graph $G_0 = (V_0 = \{v_1, \dots, v_{n_0}\}, E_0)$, initialize the decision tree as a single leaf corresponding to vertex v_1 with weight $\deg_{G_0}(v_1)$. Then for $i = 2, \dots, n_0$ generate tokens $\langle \text{VERTEX}, i, \deg_{G_0}(v_i) \rangle$. During the preferential attachment phase, use d tokens of form $\langle \text{QUERY}, j, \{\text{Random Number}\} \rangle$ followed by the vertex token $\langle \text{VERTEX}, j, d \rangle$ for the insertion and connection of vertex j . In case $\sigma = 1$, self-loops $\{i, i\}$ can be immediately output by the generator without a dedicated request and only have to be accounted for by increasing the vertex token’s weight. In this mode of operation, a new edge may depend on an earlier edge of the same vertex. This rare event can be directly resolved in the token generator by retaining the $\mathcal{O}(d)$ random values of all previously generated edges incident to the new vertex.

As illustrated in Figure 3.4, a vertex token traverses the decision tree starting from the root, always selecting a subtree that is not heavier than its direct sibling. If in tree node t the left subtree is chosen, W_t is increased by the token’s weight. When leaf u is eventually reached, u is pushed down one layer such that u and a newly generated leaf corresponding to the vertex token become children of a new common inner node.

The balancing’s worst-case performance during the processing of the seed graph depends on the nearly arbitrary degree distribution of the input. Our task reduces to the greedy approximation of $\text{min-MAKESPANPROCESSSCHEDULING}$ for 2 identical machines which is known to have an approximation factor of $7/6$ if the tokens arrive sorted by degree in a descending order [155]. Let W_l and W_r be the respective weights of the two subtrees directly under T ’s root and δ_{\max} the maximal degree in the seed graph.



(a) Initial state



(b) After inserting v_3 twice

Figure 3.4: Balancing both subtrees by inserting into lighter one.

Then, the worst-case bound is given by $|W_l - W_r| \leq 7/6 \cdot \delta_{\max}$. Experiments with powerlaw seed graphs suggest that observable deviations are much smaller with $|W_l - W_r| < c\tilde{d}$ where \tilde{d} is the seed graph's average degree and typically $c < 2$.

During the random attachment phase, the more structured sequence of d edges followed by a vertex of weight d further improves the situation, rapidly reaching the condition $|W_l - W_r| \leq d$ which henceforth holds as an invariant. As the tree's weight grows, the constant difference becomes less significant and the probability weights in both subtrees can be considered identical. Therefore, the decision whether any token enters the left subtree is sufficiently well modeled as an independent Bernoulli event with $p = 1/2$. Consequently, the situation is similar at lower levels (see Section 3.5.1). This also implies that the height of the tree remains $\mathcal{O}(\log n)$; simulations further yield that only an insignificant amount of leaves can be found in levels deeper than $\lceil \log n \rceil + 1$.

3.3.4 Parallelism

We now extend MP-BA with two non-exclusive parallelization schemes.

3.3.4.1 Tree Decomposition

The algorithm's hierarchical design suggests a tree decomposition as shown in Figure 3.5: given a PRAM² with $p \leq D = \Theta(\log(n))$ processing units (PUs), evenly distribute D/p subsequent layers of T per PU and run the algorithm in a pipelined way. In this setup, each PU executes $\Theta(m \log(n)/p)$ operations, yielding a work-efficient parallelization of MP-BA with time complexity $\Theta(m \log(n)/p + p)$. This simple mapping is well suited for root-near layers and used for this purpose in our GPU implementation.

Unfortunately it is not I/O-efficient, as with high probability the last PU has to access $\Omega(n)$ different memory locations in an unstructured fashion. To resolve this issue, we assign each PU a subtree (rather than a layer as before) enabling I/O-efficient access patterns as discussed in Section 3.3.2. Due to balancing, a node t in depth j receives approximately 2^{-j} of all tokens. In order to perform the same work as the root's PU P_1 , the PU assigned to the subtree rooted in t has to cover 2^j times more layers than P_1 .

We decompose T into $\Theta(\log n)$ smaller subtrees by applying level-wise cuts to obtain segments of heights $1 = d_1 < d_2 < \dots < D$ where $d_{i+1} = d_i 2^{d_i}$. Hence, segment i contains a forest with d_i subtrees of similar size which can be processed pleasingly parallel and in the same I/O-efficient setting as described in Section 3.3.2. The expected workload of all PUs is identical with high probability, yielding the same work and runtime bounds as before. However, any form of parallelism for MP-BA exclusively based on tree decomposition is limited to a speedup of $\mathcal{O}(\log n)$ since $\Omega(m)$ operations need to be performed for the root node.

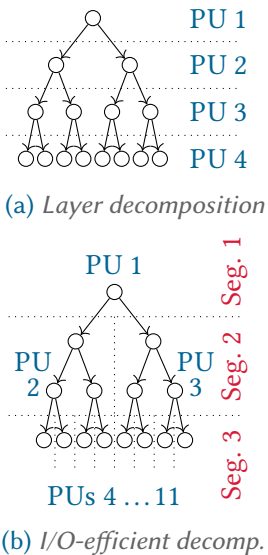


Figure 3.5: Tree layouts

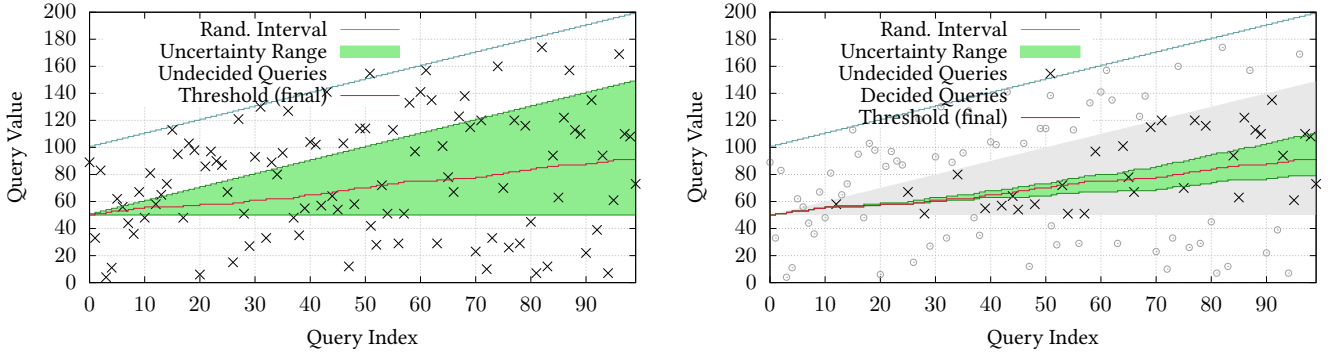


Figure 3.6: Token-wise parallel MP-BA on $p = 100$ queries before (left) and after (right) the first iteration. After the second iteration only points inside the small shaded uncertainty range will remain. This example is pathological since $p = W_{t,1}$.

3.3.4.2 Token-wise Parallelism

To obtain sub-linear time complexity, we present a token-wise parallel approach in which p processors are assigned to the same tree node t but work on p subsequent queries. Let r_1, \dots, r_p be the queries' random values and $W_{t,i}$ the threshold for the i -th token. PU i then has to compute $s_i \in \{0, 1\}$ where $s_i = 1$ iff request i is to be sent to the left subtree, i.e., $r_i < W_{t,i}$. Observe that initially only $W_{t,1}$ is known while the remaining thresholds $W_{t,i} = W_{t,1} + \sum_{j=1}^{i-1} s_j$ depend on previous decisions. However, as illustrated in Figure 3.6, their values are bound by the extreme cases in which either none or all previous tokens are assigned to the left subtree, i.e., $W_{t,1} \leq W_{t,i} \leq W_{t,1} + i - 1$. If r_i lies outside of this uncertainty interval, s_i can be computed without knowledge of s_1, \dots, s_{i-1} . This observation yields the iterative Algorithm 4 in which we fix all certain decisions in parallel and henceforth reduce the uncertainty interval for the next round. This step is repeated until all decisions are fixed. We denote the number of such iterations as I . The notion of uncertainty is formalized using an additional flag $u_i \in \{0, 1\}$ per query where $u_i = 1$ iff s_i is not yet computed. We define that initially $u_i = 1$ for all i and that $u_i = 1$ implies $s_i = 0$ as its interim value. Then, the previously stated bounds can be tightened to

$$W_{t,1} + \sum_{j=1}^{i-1} s_j \leq W_{t,i} \leq W_{t,1} + \sum_{j=1}^{i-1} s_j + \sum_{j=1}^{i-1} u_j.$$

These thresholds can be evaluated in parallel for all i in time $\mathcal{O}(\log p)$ using work-optimal prefix sums over s_i and u_i respectively [187]. Since the remaining computations per iteration are negligible, the algorithm's total time complexity is given by $\mathcal{O}(I \log p)$ where I is the number of iterations until all $u_i = 0$. In Lemma 3.3 and Corollary 3.4 we now show that $I = \mathcal{O}(1)$ with high probability if $p \leq \sqrt{W}$ where $W = W_{t,1}$ is the initial threshold. Observe that this constraint suffices for all practical settings. Nevertheless, more PUs are feasible but reduce the work-efficiency.

²PRAM is an extension of the *unit-cost Random-Access Machine* model to parallel computing: p RAM-like processing units execute a program in lockstep and access an unbounded shared memory [187].

parallel processing of layers near the root
 r_i : values to label s_i
 $W_{t,i}$: weight of t 's left subtree at arrival of r_i
 I : iterations required
 $u_i = 1$ iff i -th request is not decided yet

W_i : total tree weight
after deciding r_i

To process m tokens, the input is split into steadily growing segments of size $\sqrt{W_i}$ where W_i is the weight of the decision tree after processing the i -th segment with W_0 given by the seed graph. Hence, $\mathcal{O}(\sqrt{m})$ phases suffice and the expected total runtime of the work-efficient algorithm is given by $\mathcal{O}(\sqrt{m} \log \sqrt{m})$.

Lemma 3.3. Let $W = W_{t,1}$ be the initial weight of the decision tree, p the number of processing units and $u_i \in \{0, 1\} \forall i \in \{1, \dots, p\}$ the uncertainty flags where $u_i = 1$ iff the destination of the i -th token is not yet computed. Then, the total uncertainty after the first iteration of the token-wise parallel MP-BA is with high probability governed by

$$\sum_{i=1}^p u_i = p - \frac{W}{2} \ln \left(1 + \frac{2p}{W} \right) + \mathcal{O}(1). \quad \blacktriangleleft$$

Proof. Observe that with each edge the uncertainty range as well as the interval from which random numbers are drawn is increased by 2. Thus, the probability that token i initially falls into the uncertainty range is given by $\mathbb{P}[u_i = 1] = 2(i-1)/(W+2i-2)$. It follows

$$\mathbb{E} \left[\sum_{i=1}^p u_i \right] = \sum_{i=1}^p \mathbb{P}[u_i = 1] = p - \frac{W}{2} \left[H_{\frac{W}{2}+p} - H_{\frac{W}{2}} \right],$$

where $H_k = \sum_{i=1}^k 1/i$ is the k -th harmonic number. We substitute $H_k = \gamma + \ln(k) + \mathcal{O}(1/k)$ where γ is the Euler-Mascheroni constant which cancels out. Simplification then yields the claim. The result holds with high probability since the uncertainty of each token is an independent Bernoulli event. Therefore, the Chernoff inequality applies guaranteeing an exponentially decreasing bound on the tail distribution of $\sum_{i=1}^p u_i$. \square

Corollary 3.4. Let $W = W_{t,1}$ be the initial weight of the decision tree. In case $p \leq \sqrt{W}$ then $I = \mathcal{O}(1)$ with high probability where I is the number of iterations of the token-wise MP-BA until all tokens are fixed. \blacktriangleleft

Proof. We use Lemma 3.3 and show that

$$\lim_{W \rightarrow \infty, p = \sqrt{W}} \left(p - \frac{W}{2} \ln \left(1 + \frac{2p}{W} \right) \right) = \mathcal{O}(1).$$

We apply the Taylor series $\ln(1+x) = \sum_{i=1}^{\infty} (-1)^{i+1} \frac{x^i}{i}$ and subsequently use the bound on the number of PUs:

$$\begin{aligned} p - \frac{W}{2} \ln \left(1 + \frac{2p}{W} \right) &= p - \frac{W}{2} \left[\frac{2p}{W} - \frac{2p^2}{W^2} + \mathcal{O} \left(\frac{p^3}{W^3} \right) \right] \\ &= \frac{p^2}{W} + \mathcal{O} \left(\frac{p^3}{W^2} \right) \stackrel{p \leq \sqrt{W}}{=} \mathcal{O}(1) \end{aligned}$$

Hence, the expected uncertainty after the first phase of the algorithm is of constant size. Since in every iteration of the algorithm at least the first yet undecided token can be fixed, the uncertainty is reduced in every round. Therefore, only a constant number of iterations is expected. The result holds with high probability analogously to the proof of Lemma 3.3. \square

3.4 Implementation of MP-BA

As proof of concept we implemented a parallelized version of MP-BA targeting a heterogeneous system that contains a host CPU and a GPU co-processor.

3.4.1 Token-wise Parallel MP-BA for GPGPU

In order to sketch the high-level design of our implementation we first introduce relevant aspects of the CUDA programming model [265]: a program interacts with the GPU using memory transfers and kernel invocations. In this context, a kernel is an ordinary C-like function that is executed massively parallel on the GPU. Its instances are grouped into isolated concurrent thread arrays (CTA) which cannot be synchronized with each other during the kernel’s execution. A CTA contains a limited number of warps (32 threads executed in lock-step) which can be synchronized and may efficiently communicate using a dedicated shared memory. The co-processor’s full potential is only available if several CTAs are executed in parallel. To schedule kernel invocations and memory transfers ahead of time, several command queues (streams) are available.

While tasks within a stream are ordered, multiple streams are executed independently and potentially in parallel. Inter-stream synchronization is possible using dedicated commands which trigger an event or wait for one.

Our implementation offloads the random token generator as well as the $D_{\text{GPU}} = 9$ root-most layers of the decision tree to the co-processor. Hence, only a small portion of the decision tree has to be stored in the GPU’s main memory. The tasks are distributed over four kernels, namely the *token generator*, the *decision kernel* and two auxiliary kernels, *distribution* and *partition*, preparing the decision kernel’s output for the next stage. The last three kernels build on top of each other and are executed sequentially. Each invocation operates on all nodes of a given tree layer similar to the situation illustrated in Figure 3.5a. Thus, D_{GPU} phases are required to process a GPU-based tree.

The sequence of $\Theta(m)$ requests is split into manageable batches of about 2 million tokens stored in a self-contained structure that also keeps meta information such as the tree’s weight and decision thresholds. These buffers are passed between kernels that recursively partition the token sequence until eventually $2^{D_{\text{GPU}}+1}$ segments emerge — one for each leaf under the GPU’s tree. During this process, the memory consumption of the buffer stays the same (except for a few insignificant boundary markers). Initially, the token generator fills the buffer with a sequence of request- and vertex tokens as described in Section 3.3.3. It uses the CURAND pseudorandom generator [266].

MP-BA is distributed as follows: the *decision* kernel contains the actual algorithm executed in a single CTA with 640 threads per decision tree node. It reads 2560 tokens from the input buffer at a time and indicates their destination subtrees in a bit vector. The restriction to a single CTA allows us to execute MP-BA with only one kernel invocation; if multiple (by definition unsynchronized) CTAs were used, the prefix sum alone would need several kernels. Even worse, this setting is only sensible if the whole buffer is processed in one MP-BA-step. Consequently, the ratio between the number of tokens and the tree’s weight increases which rises the expected number of iterations I and

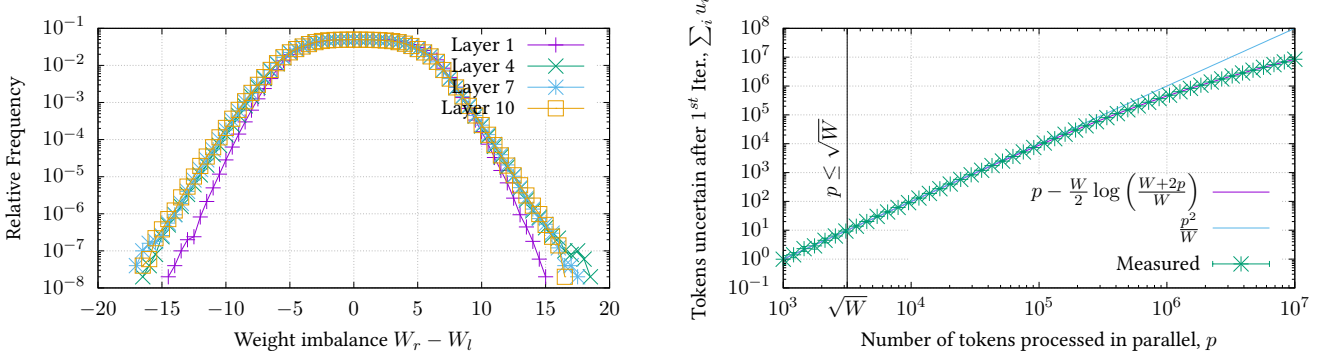


Figure 3.7: **Left (a)**: Weight imbalance in layers (root is in layer 1) $n = 10^6$, $m = 10^7$, $S = 10$. **Right (b)**: Number of uncertain tokens after 1st MP-BA iteration: $W = 10^7$, $S = 60$.

degrades the algorithm’s efficiency.

The decision kernel additionally produces auxiliary values allowing the two following kernels to rearrange the tokens according to their subtree in linear time and using multiple CTAs for I/O intensive tasks. The token-wise parallel MP-BA operates on inner nodes of the decision tree. While it can be generalized to leaves, this modification would introduce corner cases which have to be solved sequentially. Thus, our implementation first computes a complete binary tree, that has at least depth D_{GPU} , on CPU and translates it into GPU structures before engaging the co-processor.

The GPU’s occupancy is further increased based on the layer-wise pipelining described in Section 3.3.4.1 using streams: each layer corresponds to two streams (one for the decision kernel, one for the remaining kernels). The pipeline itself synchronizes using event-based control points and tolerates back-pressure. Two additional streams are used for the token generator and asynchronous data transfers to the host’s main memory. All CPU interactions are executed in bunches and (if possible) ahead of time to minimize management overhead and pipeline stalls.

3.4.2 Tree Decomposition on CPU

The CPU executes w worker threads each responsible for approximately $2^{D_{\text{GPU}}+1}/w$ independent subtrees under the GPU’s decision tree. Any such subtree corresponds to the I/O-efficient sequential case described in Section 3.3.2: all requests first traverse an in-memory decision tree with $D_{\text{CPU}} \lesssim 23$ layers and (if necessary) are then buffered in an EM-queue. In a second phase, the process is recursively applied for each queue. In this way, we construct a tree with a virtual depth of $D_{\text{GPU}} + 2D_{\text{CPU}} \approx 52$ which suffices for any conceivable practical setting. We simulate the large number of queues using one EM sorter per CPU thread which receives tokens from all leaves. Before transitioning to the aforementioned second phase, the tokens are sorted by their queue number and running index. In order to improve data locality and to make proper use of the CPU’s caches, each in-memory decision tree itself is split into three segments of depths $D_{\text{CPU}} = D_{\text{CPU}1} + D_{\text{CPU}2} + D_{\text{CPU}3}$ which are connected by small in-memory queues. While this design roughly doubles the performance compared to the queue-less

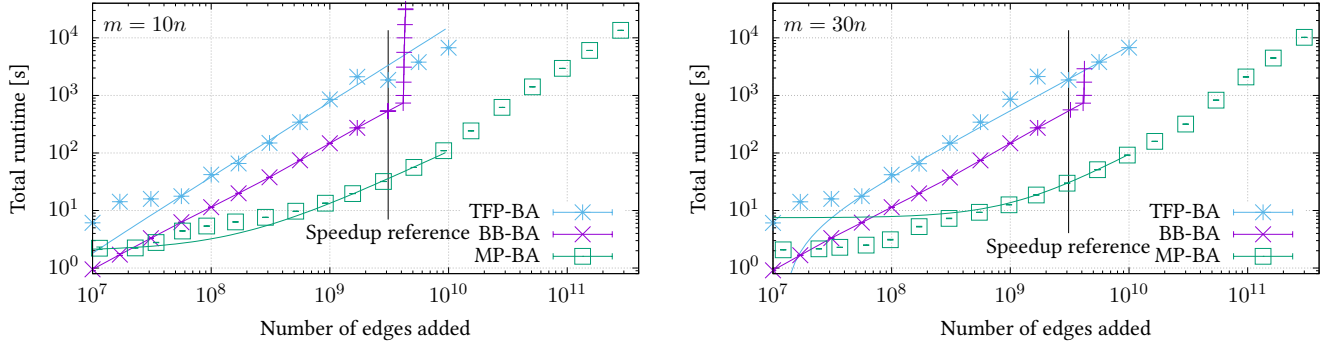


Figure 3.8: Left (a): Runtimes for $m = 10n$, $S \geq 4$. Speedup is relative to BB-BA and based on the least-squares fits plotted. At $m_{\text{ref}} = 3.1 \cdot 10^9$, the performance of our algorithms is underestimated yielding speedups $t_{\text{TFP}}/t_{\text{BA}} = (4.1 \pm 0.2)^{-1}$ and $t_{\text{MP}}/t_{\text{BA}} = 14.9 \pm 0.1$. **Right (b):** Results for $m = 30n$. $t_{\text{TFP}}/t_{\text{BA}} = (3.5 \pm 0.2)^{-1}$ and $t_{\text{MP}}/t_{\text{BA}} = 17.6 \pm 0.2$.

variant, explicit cache controlling using prefetching and non-temporal stores proved ineffective. The implementation internally uses 64 bit integers to manage vertex indices and tree weights. If possible, lower layers of the tree only store 32 bit threshold values to further improve data locality.

3.5 Experimental Results

All runtime benchmarks were conducted on the following system: Intel Xeon CPU E5-2630 v3 (8 cores, 16 threads, 2.40GHz), 64 GB 2133 MHz RAM (61.75 GiB available), GeForce GTX 980 (4 GB), 8× Samsung 850 PRO SSD (1 TB), Linux 4.0.0, GCC 4.9.2, CUDA 7.0.28, CUB 1.4.1, STXXL [102] master branch (04/15/2015).

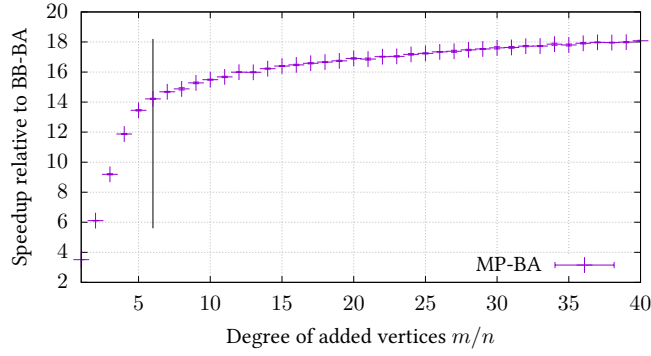
The number of repetitions per data point (with different random seeds) is denoted with S . Errorbars always correspond to the unbiased estimation of the standard deviation. All reported problem sizes correspond to the number of vertices/edges added during the preferential attachment and do not include the seed graph, a small ring with $n_0 = m_0 = 2048$. If not stated differently, measurements are conducted with $m/n = 30$ outgoing edges per added vertex which resembles the current web graph [238]. For completeness, Figure 3.8 also presents a runtime sweep for $m/n = 10$ which can be found in many real-world networks [14].

3.5.1 Balancing and Iterations in MP-BA

In Section 3.3.3 we introduced a scheme to balance the weight of two subtrees under a common node and claimed that their absolute weight difference during the preferential attachment phase is bounded by $\mathcal{O}(d)$ where d is the number of edges per new vertex. The statement is tested by simulating the sequential MP-BA for a graph with $m = 10^7$. During the execution, snapshots of all weights in the 12 root-most layers are taken at random points during the execution. Figure 3.7 presents a histogram of imbalances captured for different layers which supports our claim.

During the analysis of the token-wise parallel MP-BA we estimated the number

Figure 3.9: Speedup relative to BB-BA as a function of m/n . For $m/n \leq 6$, the number of vertices exceeds internal memory and MP-BA uses EM. $m = m_{\text{ref}}, S = 3$.



of uncertain tokens after the first iteration. Figure 3.7 plots $\mathbb{E}[\sum_{i=1}^p u_i]$ as derived in Lemma 3.3 and its simplification used in Corollary 3.4 for a dedicated initial tree weight of $W = 10^7$. The approximation is almost exact for the relevant range of $p \leq \sqrt{W}$. The simulation results included in the figure match the prediction and indicate that the constant terms in the estimations are negligible. Based on this estimation, we argue in Corollary 3.4 that I , the number of iterations of MP-BA until all decisions are fixed, is constant with high probability if at most $p = \sqrt{W}$ tokens are processed. This statement is tested by simulating the algorithm for different $W \in [10^4, 10^{14}]$. Here, the average number of iterations was found to be 1.39(48) with no observable trend. Due to hardware constraints, we fix $p = 2560$ in our implementation and estimate the expected number of iterations in the decision kernel as $\mathbb{E}[I] \ll 1.1$ during the generation of a graph with $m = 10^9$.

3.5.2 Runtime and Scalability

We implemented the three discussed algorithms and compare their runtimes for different problem sizes in Figure 3.8. To achieve comparability with other publications, the writing of the result graph to storage is disabled [12]. The time to setup data structures (e.g., to read the seed graph from storage) and for I/Os during execution are included. The implementations of BB-BA and TFP-BA rely on STXXL’s fast 64-bit random number generator. We use an optimized version of BB-BA which we consider nearly optimal.

For large graphs that still fit into main memory ($m_{\text{ref}} = 3.1 \cdot 10^9$, 49 GB data), TFP-BA is 3.5(2) times slower than BB-BA, but outperforms the latter by several orders of magnitude as soon as BB-BA’s edge list exceeds the available main memory by 2%. MP-BA is faster than BB-BA for all graphs with $m > 2.5 \cdot 10^7$ and shows a speedup of 17.6(1) at m_{ref} . It scales well for graphs far larger than main memory: the edge list of a graph with $m = 3 \cdot 10^{11}$ has a size of 4.8 TB and is generated in 170 min.

MP-BA’s runtime for constant m depends on m/n since the decision tree’s depth is a function of the number of vertices and since our balancing scheme requires operations for every vertex. Section 3.5.2 illustrates this behavior at m_{ref} and indicates that the dependency is only relevant for $m/n < 10$ which, however, seems atypical for huge networks. The small m/n dependency in our implementation of TFP-BA is less systematic and seems to be an artefact of the priority queue as already visible in Figure 3.8.

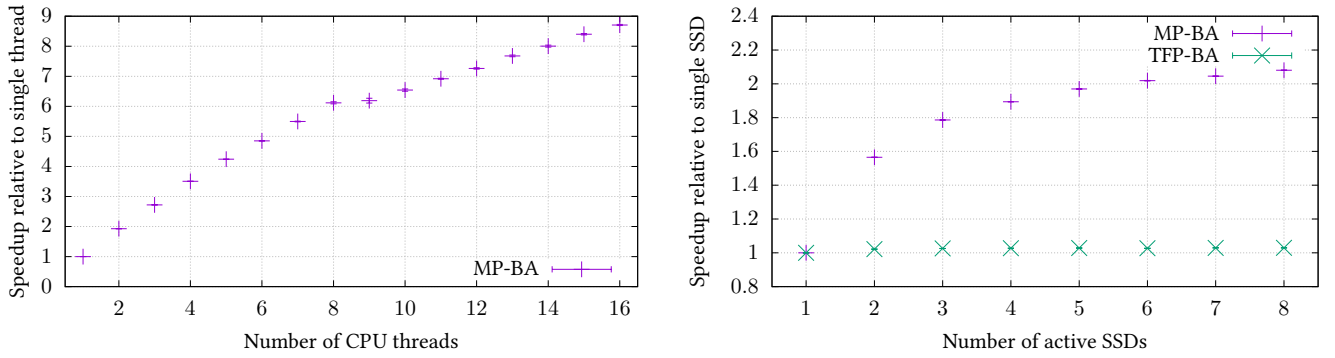


Figure 3.10: Left (a): Strong scaling in CPU threads: $n = 3.1 \cdot 10^8$, $m = 10n = m_{\text{ref}}$, $D_{\text{GPU}} = 9$, $S = 5$. **Right (b):** Strong scaling in available SSDs. Speedup relative to case with a single SSD. $m = 10n$. MP-BA: $n = 1.7 \cdot 10^9$ TFP-BA: $n = 10^8$.

In its current form, our implementation’s bottleneck is given by the CPU workers which collectively process $(103 \pm 0.9) \times 10^6$ tokens/s, while the GPU produces and copies $(214 \pm 1) \times 10^6$ tokens in the same time.³

The implementation’s CPU scalability is quantified on a single machine using the strong scalability measure: one increases the number of processing units while keeping the problem size constant. On an eight-core processor with 16 hardware threads, a speedup of 6.2 relative to a single thread is observed using eight compute threads, which can be further increased to 8.7 when saturating all hardware threads. To quantify the effect of the external memory bandwidth, Figure 3.10 shows the strong scaling of TFP-BA and MP-BA as a function of the number of SSDs available. The measurement indicates that the sequential TFP-BA is compute bound, while MP-BA produces enough data to facilitate the bandwidth of multiple drives. The speedup does not change significantly for larger graphs.

3.5.3 Large Seed Graphs

In order to produce an artificial dataset based on real data, it is crucial that the graph generator supports large seed graphs. We demonstrate this capability for MP-BA by using the ClueWeb12 Web Graph, an directed hyperlink graph crawled in the year 2013 [324]. In a preprocessing step, we interpret directed edges as undirected, remove duplicates and compute the degree distribution. The resulting graph contains $n_0 = 6,257,706,959$ vertices and $m_0 = 66,539,548,496$ edges with $m/n \approx 10.6$. We then used MP-BA to roughly double its size by adding $n = 6 \cdot 10^9$ vertices and $m = 6.6 \cdot 10^{10}$ edges in 2675(3) s.

3.5.4 Previous Results


To the best of our knowledge, all published parallelizations of BA are targeting a distributed scenario. Alam et al. use a cluster with 48 nodes each equipped with

³Initial studies suggested that a multi-CTA decision “kernel” increases the throughput by at least one order of magnitude. This may become relevant when MP-BA is adopted for distributed machines.

two Intel Xeon E5-2670 2.60GHz 8-core processors sharing 64GB of 1600MHz DDR3 RAM [12]. Each of the 96 CPUs has a similar performance to our single machine, but their memory bandwidth per node is 1.4 times lower. They produce a graph with $n = 10^9$ and $m = 5 \cdot 10^{10}$ where each new vertex introduces 50 edges. They report a runtime of 123 s using all CPUs.

For the same graph parameters, our implementation achieves a runtime of 489.1(2) s on a single machine. This corresponds to a slowdown factor of only four by using a 1/96 the number of CPUs and hence MP-BA poses a viable and cost-efficient alternative.

3.6 Preferential Attachment beyond BA

additional generators:
 [Section 2.4.2](#)

Up to this point, we focused on the BA model, easing the description of our algorithms. In this chapter, we highlight certain features of other preferential attachment models and show how TFP-BA and MP-BA can be adopted accordingly. It is worth noting that a few models explicitly forbid parallel edges which we currently do not support. However, for sufficiently large seed graphs, which can be easily constructed using a generalized BB-BA, the parallel edges become insignificant. None of the modifications presented here changes the asymptotic I/O or time complexity of the algorithms.

3.6.1 Alternative Vertex Sampling

Many application-specific networks, such as the World Wide Web, exhibit a directed structure which cannot be captured by BA graphs. A number of models have been proposed to explain the evolution of directed random graphs. Typically, such schemes do not select nodes based on their total degree but rely on distinct distributions for in- and out-degrees respectively and use them in different scenarios [275, 197, 60]. Further, in real networks some connections seem not be governed by a cumulative advantage process. This is often modeled either explicitly by selecting some neighbors uniformly from all available vertices, or implicitly encoded in the sample probability in terms of a constant offset $k_0 > 0$, i.e., $P[u] \propto \deg(u) + k_0$.

In TFP-BA, these modifications are straight-forward: to sample uniformly, the index of the neighbor vertex can be directly drawn during the token generation (the algorithm's second phase). An appropriate LINK token is then inserted into the priority queue, which allows later preferential attachment processes to take the new edge into account. While TFP-BA as in Algorithm 1 generates undirected graphs, all internal structures are in fact directed. Thus, the first entry in the lookup buffer (*last_edge*) always corresponds to the most recently introduced vertex while the second entry keeps the neighbor it links to. Therefore, QUERY-1 tokens yield results based on the out-degree while QUERY-2 tokens capture the in-degree. TFP-BA randomly chooses the token type in phase 2, effectively yielding a preferential attachment based on the total degrees.

For MP-BA, it suffices to replace the value W_t at each inner node t of the decision tree by a 3-tuple containing the number of vertices in the left child's subtree as well as their total in- and out-degrees. Balancing can then be performed in terms of a node's

most unbalanced value. Additionally, the imbalance score may be weighted by the expected frequency of the query types.

An extreme case of (partial) uniform sampling arises in the model of Dorogovtsev et al., where new vertices are initially isolated and cannot be connected by a pure preferential attachment process [111]. Even such disconnected graphs are directly supported in the generalized MP-BA, since one can easily initialize the in/out-degrees of a new vertex to zero. For TFP-BA no modifications are necessary, exploiting the fact that no lookups to the edge list are performed for uniformly chosen vertex indices.

Bandyopadhyay et al. consider the growth of a directed network in the context of food chains. At each time step a new vertex is added and connects to a constant number of existing vertices. Given a parameter $k_0 \in \mathbb{N}_{>0}$, the probability to connect to a vertex u decreases with each new successor and is governed by $\mathbb{P}[u] \propto (k_0 - \text{deg}_{\text{out}}(u))$. As this model only uses out-degrees, MP-BA can be used with a single value W_t at each inner leaf: for each new vertex descending into the left subtree W_t is increased by k_0 and decreased by one for each query.

3.6.2 Non-Uniform Node Degrees

The BA model assigns the same initial degree d to every vertex introduced during preferential attachment. While this property simplifies the analysis of the generated graphs, it can be argued that insertion patterns of real-world processes are more complex. In the context of directed graphs, the issue becomes even more pressing since currently the out-degree of a vertex cannot be changed once all initial edges have been created.

Price suggested a model for directed citation networks in which the initial out-degree of a new publication is randomly distributed with an average value \tilde{d} [275]. Since neither MP-BA nor TFP-BA rely on uniform initial edge degrees, the algorithms can be used for this more general case without changes to the data structures.

Krapivsky et al. proposed a model where in every time step a new vertex with out-degree 1 is added with probability $0 \leq p \leq 1$ [197]. Otherwise, an edge (u, v) is introduced where u is selected based on an preferential attachment process on the in-degrees and v in terms the graph's out-degrees respectively. Due to the lazy evaluation used by TFP-BA and MP-BA, both queries are answered independently and in general at different times. In TFP-BA, one can add the token classes `QUERYFROM- x/QUERYTO- x` which are processed as ordinary query tokens, however reinsert `LINKFROM` and `LINKTO` respectively instead of ordinary `LINK` tokens. If a pair of `LINKFROM` and `LINKTO` carry the same edge index, the priority queue places them next to each other and they can easily be merged to produce a single edge. A similar approach is possible for MP-BA: here one can add an edge index to a query and annotate the resulting edge with this identifier. In a post-processing step, the complete output is sorted based on these identifiers and partial results can be merged into a final edge list.

Algorithm 2: Update the decision tree without buffers (Section 3.3.2)

Input : Query token `TOKEN`; decision tree (`ROOT`, `TREEWEIGHT`) to be modified in-place
Output: Edge-based on query token and leaf found

```

1 TREEWEIGHT  $\leftarrow$  TREEWEIGHT + 1
2 NODE  $\leftarrow$  ROOT
3 while not NODE.ISLEAF do
4     if TOKEN.VALUE  $\leq$  NODE.WEIGHTLEFT then
5         NODE.WEIGHTLEFT  $\leftarrow$  NODE.WEIGHTLEFT + 1
6         NODE  $\leftarrow$  NODE.LEFTCHILD
7     else
8         TOKEN.VALUE  $\leftarrow$  TOKEN.VALUE - NODE.WEIGHTLEFT
9         NODE  $\leftarrow$  NODE.RIGHTCHILD
10 return { TOKEN.VERTEXID, NODE.VERTEXID }

```

Algorithm 3: Balancing used by MP-BA without I/O-efficient buffering.

Input : vertex token `TOKEN`; decision tree (`ROOT`, `TREEWEIGHT`) modified in-place

```

1 TREEWEIGHT  $\leftarrow$  TREEWEIGHT + TOKEN.WEIGHT
2 WEIGHT  $\leftarrow$  TREEWEIGHT
3 NODE  $\leftarrow$  ROOT
4 while not NODE.ISLEAF do
5     if NODE.WEIGHTLEFT < WEIGHT/2 then
6         NODE.WEIGHTLEFT  $\leftarrow$  NODE.WEIGHTLEFT + TOKEN.WEIGHT
7         WEIGHT  $\leftarrow$  NODE.WEIGHTLEFT
8         NODE  $\leftarrow$  NODE.LEFTCHILD
9     else
10        WEIGHT  $\leftarrow$  WEIGHT - NODE.WEIGHTLEFT
11        NODE  $\leftarrow$  NODE.RIGHTCHILD
12 NODE  $\leftarrow$   $\begin{cases} \text{WEIGHTLEFT : } & \text{WEIGHT} \\ \text{LEFTCHILD : } & \text{Leaf, vertex = NODE.VERTEXID} \\ \text{RIGHTCHILD : } & \text{Leaf, vertex = TOKEN.VERTEXID} \end{cases}$ 

```

Algorithm 4: Token-wise parallel MP-BA applied to a node whose left subtree has weight T . For simplicity, we omit vertex-tokens.

Input : (R_1, \dots, R_p) token values, T decision threshold
Output: $(s_1, \dots, s_p) \in \{0, 1\}^p$ with $s_j = 1$ iff token j belongs to left subtree

- 1 Execute program on each PU $i \in \{1 \dots p\}$ in lockstep.
- 2 $U_i \leftarrow 1$, PREFIXSUM_ $U_i \leftarrow i - 1$ // Initialize state as undecided
- 3 $S_i \leftarrow 0$, PREFIXSUM_ $S_i \leftarrow 0$
- 4 **repeat**
 - // Here invariants PREFIXSUM_ $U_i = \sum_{j=1}^{i-1} U_j$, and PREFIXSUM_ $S_i = \sum_{j=1}^{i-1} S_j$ hold
 - 5 LOWERB $\leftarrow T + \text{PREFIXSUM}_{S_i}$ // Compute bounds using the last prefix sums
 - 6 UPPERB $\leftarrow \text{LOWERB} + \text{PREFIXSUM}_{U_i}$
 - 7 **if** $R_i < \text{LOWERB}$ **then**
 - 8 $U_i \leftarrow 0$; $S_i \leftarrow 1$ // Fix tokens below the uncertainty range
 - 9 **else if** $R_i \geq \text{UPPERB}$ **then**
 - 10 $U_i \leftarrow 0$; $S_i \leftarrow 0$ // Fix tokens above the uncertainty range
 - 11 PREFIXSUM_ $S_i \leftarrow i$ -th results of excl. scan over $s[\cdot]$ // Prepare next iteration
 - 12 PREFIXSUM_ $U_i \leftarrow i$ -th results of excl. scan over $u[\cdot]$
- 13 **until** PREFIXSUM_ $U_p == 0$

I/O-Efficient Generation of Massive Graphs Following the LFR Benchmark

joint work with M. Hamann, U. Meyer, H. Tran, D. Wagner

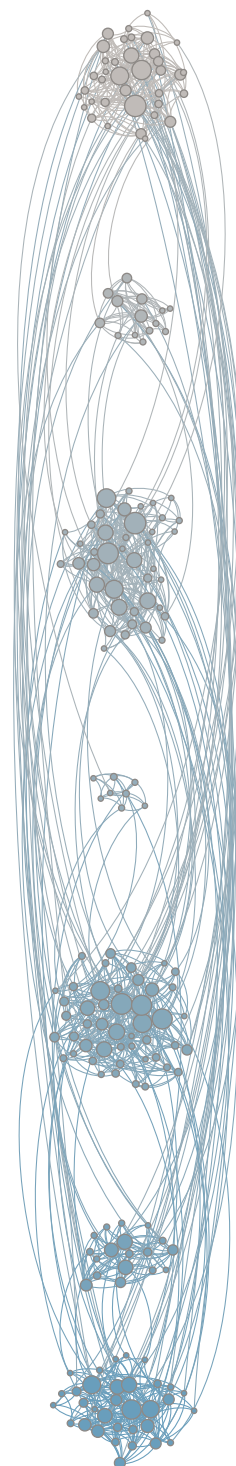
LFR is a popular benchmark graph generator used to evaluate community detection algorithms. We present EM-LFR, the first external memory algorithm able to generate massive complex networks following the *LFR* benchmark. Its most expensive component is the generation of random graphs with prescribed degree sequences which can be divided into two steps: the graphs are first materialized deterministically using the HAVEL-HAKIMI algorithm, and then randomized. Our main contributions are EM-HH and EM-ES, two I/O-efficient external memory algorithms for these two steps. We also propose EM-CM/ES, an alternative sampling scheme using the Configuration Model and rewiring steps to obtain a random simple graph. In an experimental evaluation we demonstrate their performance; our implementation is able to handle graphs with more than 37 billion edges on a single machine, is competitive with a massively parallel distributed algorithm, and is faster than a state-of-the-art internal memory implementation even on instances fitting in main memory. EM-LFR's implementation is capable of generating large graph instances orders of magnitude faster than the original implementation. We give evidence that both implementations yield graphs with matching properties by applying clustering algorithms to generated instances. Similarly, we analyze the evolution of graph properties as EM-ES is executed on networks obtained with EM-CM/ES and find that the alternative approach can accelerate the sampling process.

This chapter is based on the peer-reviewed journal article [167] extending [168]:

- [168] M. Hamann, U. Meyer, M. Penschuck, and D. Wagner. I/O-efficient generation of massive graphs following the LFR benchmark. In S. P. Fekete and V. Ramachandran, editors, *Workshop on Algorithm Engineering and Experiments ALENEX*, pages 58–72. Society for Industrial and App. Math. SIAM, 2017. doi:10.1137/1.9781611974768.5 .
- [167] M. Hamann, U. Meyer, M. Penschuck, H. Tran, and D. Wagner. I/O-efficient generation of massive graphs following the LFR benchmark. *ACM J. of Experimental Algorithmics*, 23, 2018. doi:10.1145/3230743 .

My contribution

Michael Hamann and I are the main authors contributing equal amounts to [168] and its implementation. The journal version adds new algorithmic ideas related to the *Configuration Model* based on work of Hung Tran and myself.



LFR graph with
 $n = 200$, $\mu = 0.1$,
 $d_i \sim \text{PLD}([10, 30], -2)$,
 $s_i \sim \text{PLD}([5, 50], -1)$

4.1 Introduction

Complex networks, such as web graphs or social networks, usually contain communities, also called clusters, that are internally dense but externally sparsely connected. Finding these clusters, which can be disjoint or overlapping, is a common task in network analysis. A large number of algorithms trying to find meaningful clusters have been proposed (see [131, 172, 133] for an overview). Commonly, synthetic benchmarks are used to evaluate and compare these clustering algorithms, since for most real-world networks it is unknown which communities they contain and which of them are actually detectable through structure [29, 133]. The *LFR* benchmark [210, 208] has become a standard benchmark for such experimental studies, both for disjoint and for overlapping communities [120].

With the emergence of massive networks that cannot be handled in the main memory of a single computer, new clustering schemes have been proposed for advanced models of computation [77, 351]. Since such algorithms typically use hierarchical input representations, quality results of small benchmarks may not be generalizable to larger instances [120, 169]. Often though, the quality is only evaluated on small benchmark graphs as currently available graph clustering benchmark generators are unable to generate the necessary graphs [30, 77]. Instead, computationally inexpensive random graph models such as *R-MAT* are used [278] to generate huge graphs. Using those models, it is however not possible to evaluate whether the clustering algorithm is actually able to detect communities on such a large graph as there is no ground truth community structure to compare against. Filling this gap, we propose a generator in the external memory (EM) model following the *LFR* benchmark in order to produce clustering benchmark graph instances exceeding main memory. We implement the variants of the *LFR* benchmark for unweighted, undirected graphs with either overlapping or non-overlapping communities. Our proposed graph benchmark generator has already been used to evaluate the clustering quality of distributed clustering algorithms on graphs with up to 512 million nodes and 76.6 billion edges [169].

The distributed *CKB* benchmark [95] is a step into a similar direction, however, it considers only overlapping clusters and uses a different model of communities. In contrast, our approach is a direct realization of the established *LFR* benchmark and supports both disjoint and overlapping clusters.

4.1.1 Random Graphs from a Prescribed Degree Sequence

FDSM The *LFR* benchmark uses the *Fixed-Degree-Sequence-Model (FDSM)*, also known as edge switching Markov chain algorithm (e.g., [246]), to obtain a random graph following a previously computed degree sequence. In preliminary studies, we identified this task as the main issue when transferring the *LFR* benchmark into an EM setting; both in terms of algorithmic complexity and runtime.

FDSM consists of two steps, namely (i) generating a deterministic graph from a prescribed degree sequence and (ii) randomizing this graph using random edge switches. For each edge switch, two edges are chosen uniformly at random and two of the

endpoints are swapped if the resulting graph is still simple (see Section 4.5). Each edge switch can be seen as a transition in a Markov chain. This Markov chain is irreducible [118], symmetric and aperiodic [151] and therefore converges to the uniform distribution. It also has been shown to converge in polynomial time if the maximum degree is not too large compared to the number of edges [158]. However, the current analytical bounds of the mixing time are impractically high even for small graphs.

Experimental results on the occurrence of certain motifs in networks [246] suggest that $100m$ steps should be more than enough where m is the number of edges. Further results for random connected graphs [151] suggest that the average and maximum path length and link load converge between $2m$ and $8m$ swaps. More recently, further theoretical arguments and experiments showed that $10m$ to $30m$ steps are enough [280].

A faster way to realize a given degree sequence is the *Configuration Model* which allows multi-edges and self-loops. In the *Erased Configuration Model* these illegal edges are deleted. Doing so, however, alters the graph properties and does not properly realize the skewed degree distributions required for *LFR* [296]. In this context the question arises whether edge switches starting from the Configuration Model can be used to uniformly sample simple graphs at random.

4.1.2 Our Contribution

We introduce EM-LFR¹, the first I/O-efficient *LFR* variant, and study the *FDSM* in the external memory model. After defining our notation, we summarize the original *LFR* benchmark in Section 4.3. As illustrated in Figure 4.1, EM-LFR consists of several algorithmic building blocks which we discuss in sections 4.7 to 4.8. Here, the focus lies on *FDSM* consisting of (i) generating a deterministic graph from a prescribed degree sequence (cf. EM-HH, Section 4.4) and (ii) randomizing this graph using random edge switches (cf. EM-ES, Section 4.5). For EM-HH, we describe a streaming algorithm whose internal data structure only has an I/O complexity linear in the number of different degrees if a monotonous degree sequence is provided. To execute a number of edge switches proportional to the number m of edges, EM-ES triggers $\mathcal{O}(\text{sort}(m))$ I/Os. For EM-LFR, the I/O complexity is the same as it is dominated by the edge randomization step. In Section 4.6, we additionally describe EM-CM/ES, an alternative to *FDSM*. It generates uniform random non-simple graphs using the Configuration Model in $\mathcal{O}(\text{sort}(m))$ I/Os and then obtains a simple graph by applying edge rewiring steps.

We conclude with an experimental evaluation of our algorithms and demonstrate that our EM version of the *FDSM* is faster than an existing state-of-the-art implementation even for instances still fitting into RAM. It scales well to large networks, as we demonstrate by handling a graph with 37 billion edges on a desktop computer, and almost an order of magnitude more efficient than an existing distributed parallel algorithm. Further, we compare EM-LFR to the original *LFR* implementation and show that EM-LFR is significantly faster while producing equivalent networks in terms of

¹The implementation is freely available at <https://massive-graphs.org/extmem-lfr>. Among others, it contains encapsulated implementations of EM-ES and EM-CM/ES easily reusable for novel application.

community detection algorithm performance and graph properties.

A *LFR* benchmark graph with more than $1 \cdot 10^{10}$ edges can be generated in 17 h on a single server with 64 GB RAM and 3 SSDs. We also investigate the mixing time of EM-ES and EM-CM/ES and give evidence that our alternative sampling scheme quickly yields uniform samples and that the number of swaps suggested by the original *LFR* implementation can be kept for EM-LFR.

4.2 Preliminaries and Notation

In this section, we highlight important definitions and notations used through the document, and give an introduction to the external memory model as well as *Time Forward Processing*, a crucial design-principle used in EM-ES.

4.2.1 Notation

We define the short-hand $[k] := \{1, \dots, k\}$ for $k \in \mathbb{N}_{>0}$, and write $[x_i]_{i=a}^b$ for an ordered sequence $[x_a, x_{a+1}, \dots, x_b]$.

Graphs and degree sequences. A graph $G = (V, E)$ has $n = |V|$ sequentially numbered nodes $V = \{v_1, \dots, v_n\}$ and $m = |E|$ edges. Let $\deg(v_i)$ denote the degree (i.e. number of neighbors) of node v_i . $\mathcal{D} = [d_i]_{i=1}^n$ is a degree sequence of graph G iff $\forall v_i \in V: \deg(v_i) = d_i$. Unless stated differently, graphs are assumed to be undirected and unweighted. A graph is called *simple* if it contains neither multi-edges nor self-loops. To obtain a unique representation of an *undirected* edge $\{u, v\} \in E$, we use *ordered* edges $[u, v] \in E$ implying $u \leq v$; in contrast to a directed edge, the ordering is used algorithmically but does not carry any meaning. Unless stated differently, our EM algorithms represent a graph $G = (V, E)$ as a sequence containing for every ordered edge $[u, v] \in E$ only the entry (u, v) .

Randomization and Distributions. $\text{PLD}([a, b], \gamma)$ denotes an integer PowerLaw Distribution with exponent $-\gamma \in \mathbb{R}$ for $\gamma \geq 1$ and values from the interval $[a, b]$; let X be an integer random variable drawn from $\text{PLD}([a, b], \gamma)$ then $\mathbb{P}[X=k] \propto k^{-\gamma}$ (proportional to) if $a \leq k < b$ and $\mathbb{P}[X=k] = 0$ otherwise. For $X = [x_i]_{i=1}^n$, we define the *mean* $\langle X \rangle := \sum_{i=1}^n x_i/n$ and the *second moment* $\langle X^2 \rangle := \sum_{i=1}^n x_i^2/n$ of the sequence X . A statement depending on some $x > 0$ is said to hold *with high probability* if it is satisfied with probability at least $1 - 1/x^c$ for some constant $c \geq 1$.

Also refer to Section 4.A (Appendix) for a summary of commonly used definitions.

4.2.2 External Memory Model

In contrast to classic models of computation, such as the unit-cost RAM, modern computers contain deep memory hierarchies ranging from fast registers, caches and main memory to solid-state drives (SSDs) and hard disks. Algorithms unaware of these properties may face performance penalties of several orders of magnitude. We use the commonly accepted external memory (EM) model by Aggarwal and Vitter [7] to

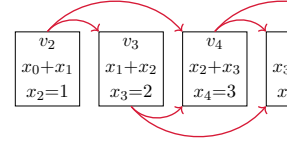
EM: external memory

Algorithm 5: Compute Fibonacci numbers using *Time Forward Processing*

```

1 PQ.PUSH((KEY = 2, VAL = 0), (KEY = 2, VAL = 1)) // Send base cases  $x_0$  &  $x_1$  to  $v_2$ 
2 for  $i \leftarrow 2, \dots, n$  do
3   SUM  $\leftarrow$  0
4   while PQ.MIN.KEY ==  $i$  do
5     SUM  $\leftarrow$  SUM + PQ.REMOVEMIN().VAL // Receive all messages for  $x_i$ 
6     PRINT( $x_i =$  SUM)
7   PQ.PUSH((KEY =  $i+1$ , VAL = SUM), (KEY =  $i+2$ , VAL = SUM))

```



reason about the influence of data locality in memory hierarchies. The model features two memory types with fast internal memory (IM) which may hold up to M data items, and a slow disk of unbounded size. The input and output of an algorithm are stored in EM while computation is only possible on values in IM. The measure of an algorithm's performance is the number of I/Os required. Each I/O transfers a block of B consecutive items between memory levels. Reading or writing n contiguous items from or to disk requires $\text{scan}(n) = \Theta(n/B)$ I/Os. Sorting n contiguous items uses $\text{sort}(n) = \Theta((n/B) \cdot \log_{M/B}(n/B))$ I/Os. For realistic values of n , B and M , $\text{scan}(n) < \text{sort}(n) \ll n$. Sorting complexity often constitutes a lower bound for intuitively non-trivial tasks [7, 242].

IM : internal memory
 M : main memory size

B : block size
 scan
 sort

4.2.3 TFP: Time Forward Processing

Time Forward Processing (TFP) is a generic technique to manage data dependencies in external memory algorithms [230]. Consider an algorithm computing values x_1, \dots, x_n where the calculation of x_i requires previously computed values. One typically models these dependencies using a directed acyclic graph $G=(V, E)$. Every node $v_i \in V$ corresponds to the computation of x_i , and an edge $(v_i, v_j) \in E$ indicates that the value x_i is necessary to compute x_j . As an example consider the Fibonacci sequence $x_0 = 0$, $x_1 = 1$, $x_i = x_{i-1} + x_{i-2} \forall i \geq 2$. Here, each node v_i with $i \geq 2$ depends on its two direct predecessors (see Algorithm 5).

In general, an algorithm needs to traverse G according to some topological order \prec_T of nodes V and also has to ensure that each v_j can access values from all v_i with $(v_i, v_j) \in E$. The TFP technique achieves this as follows: as soon as x_i has been calculated, messages of form $\langle v_j, x_i \rangle$ are sent to all successors $(v_i, v_j) \in E$. These messages are kept in a minimum priority queue sorting the items by their recipients according to \prec_T . By definition, the algorithm only starts the computation v_i once all predecessors $v_j \prec_T v_i$ are completed. Since these predecessors already removed their messages from the PQ, items addressed to v_i (if any) are currently the smallest elements in the data structure and can be dequeued. Using a suited EM PQ [23, 291], TFP incurs $\mathcal{O}(\text{sort}(k))$ I/Os, where k is the number of messages sent.

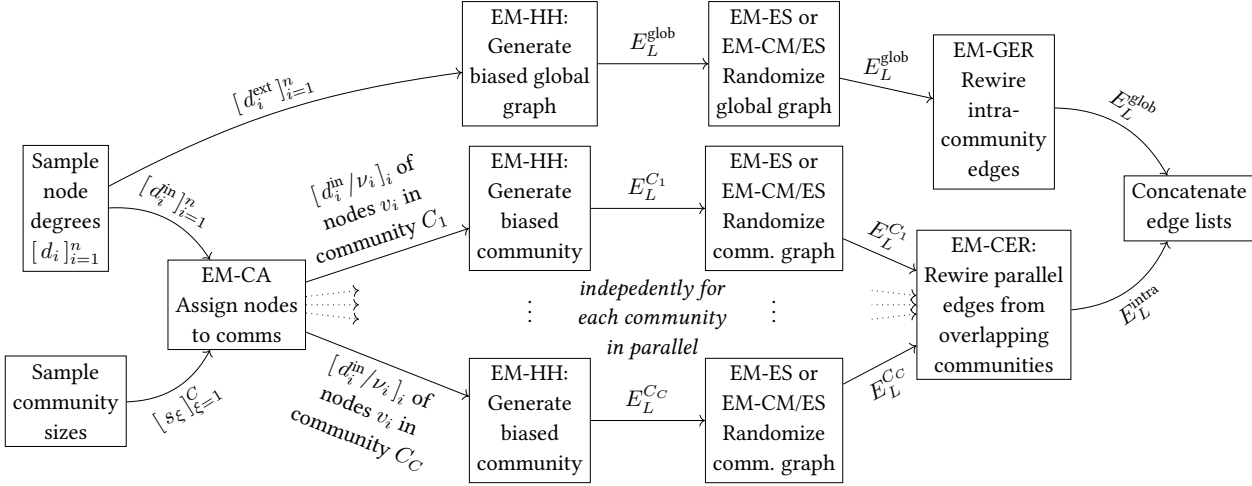


Figure 4.1: The EM-LFR pipeline: After randomly sampling the node degrees and community sizes, nodes are assigned into suited communities by EM-CA (Section 4.7). The global (inter-community) graph and each community graph is then generated independently by first materializing biased graphs using EM-HH (Section 4.4) followed by a randomization using EM-ES or EM-CM/ES (Sections 4.5 and 4.6). The global graph may contain edges between nodes of the same community which would decrease the mixing μ and are hence rewired using EM-GER (Section 4.8.1). Similarly, two overlapping communities can have identical edges which are rewired by EM-CER (Section 4.8.2).

4.3 The LFR Benchmark

In this section we introduce the properties and features of the *LFR* benchmark, outline important algorithmic challenges, and address each of them by proposing a suited EM algorithm in the following chapters (refer to Figure 4.1 for an overview).

The *LFR* benchmark [210] describes a generator for random graphs featuring node degrees and community sizes both following powerlaw distributions. The produced networks also contain a planted community structure against which the performance of detection algorithms is measured. A revised version [208] additionally introduces weighted and directed graphs with overlapping communities and changes the sampling algorithm even for the original settings. We consider the modern generator, which is also used in the author’s implementation, and focus on the most common variants for unweighted, undirected graphs and optionally overlapping communities. All its parameters are listed in Section 4.A (Appendix) and are fully supported by EM-LFR.

LFR starts by randomly sampling the degrees $\mathcal{D} = [d_i]_{i=1}^n$ of all nodes, the numbers $[\nu_i]_{i=1}^n$ of clusters they are members in, and community sizes $S = [s_\xi]_{\xi=1}^C$ such that $\sum_{\xi=1}^C s_\xi = \sum_{i=1}^n \nu_i$ according to the supplied parameters. During this process the number of communities C follows endogenously and is bounded by $C = \mathcal{O}(n)$ even if nodes are members in $\nu = \mathcal{O}(1)$ communities.²

Depending on the *mixing parameter* $0 < \mu < 1$, every node $v_i \in V$ is incident to $d_i^{\text{ext}} = \mu \cdot d_i$ inter-community edges and $d_i^{\text{in}} = (1 - \mu) \cdot d_i$ edges within its communities.

²Under the realistic assumption that the maximal community size grows with $s_{\max} = \Omega(n^\epsilon)$ for some $\epsilon > 0$, the bound improves to $C = o(n)$ whp. due to the powerlaw distributed community sizes.

we consider unweighted and undirected LFR and support overlapping communities

\mathcal{D}, d_i : node degrees
 ν_i : memberships node i
 S, s_ξ : community sizes
 C : num. of communities

μ : mixing parameter

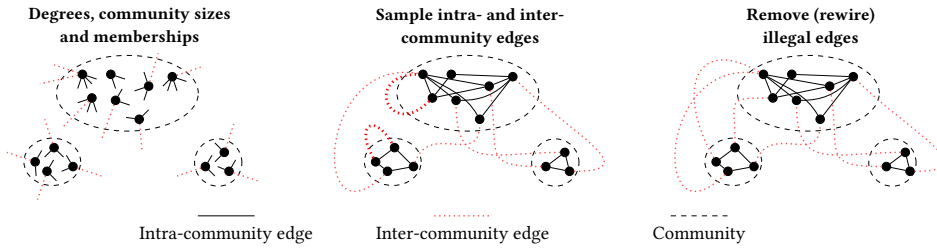


Figure 4.2: **Left:** Sample node degrees and community sizes from two powerlaw distributions. The mixing parameter μ determines the fraction of the inter-community edges. Then, assign each node to sufficiently large communities. **Center:** Sample intra-community graphs and inter-community edges independently. This may lead to illegal intra-community edges in the global graph as shown here in bold. **Right:** Lastly, remove illegal inter-community edges respective to the global graph.

In the case of overlapping communities, the internal degree is evenly split among all communities of the node. Both the computation of d_i^{in} and the division d_i^{in}/ν_i into several communities use non-deterministic rounding to avoid biases. *LFR* assigns every node v_i to either $\nu_i = 1$ or $\nu_i = \nu$ communities at random such that the requested community sizes and number of communities per node are realized. It further ensures that the desired internal degree d_i^{in}/ν_i is strictly smaller than the size s_ξ of its community ξ .

As illustrated in Figure 4.2, the *LFR* benchmark then generates the inter-community graph using *FDSM* on the degree sequence $[d_i^{\text{ext}}]_{i=1}^n$. In order not to violate the mixing parameter μ , rewiring steps are applied to the global inter-community graph to replace edges between two nodes sharing a community. Analogously, an intra-community graph is sampled for each community. In the overlapping case, rewiring steps may be necessary to remove edges that exist in multiple communities and would result in duplicate edges in the final graph.

4.4 EM-HH: Deterministic Edges from a Degree Sequence

In this section, we address the issue of generating a graph from prescribed degrees and introduce an EM-variant of the well known HAVEL-HAKIMI scheme. It takes a positive non-decreasing degree sequence $\mathcal{D} = [d_i]_{i=1}^n$ and, if possible, outputs a graph $G_{\mathcal{D}}$ realizing these degrees.³ EM-LFR uses this algorithm (cf. Figure 4.1) to first obtain a legal but biased graph following \mathcal{D} and then randomizes $G_{\mathcal{D}}$ in a subsequent step.

A sequence \mathcal{D} is called *graphical* if a matching simple graph $G_{\mathcal{D}}$ exists. Havel and Hakimi independently gave inductive characterizations of graphical sequences which directly lead to a graph generator [173, 165]: given \mathcal{D} , connect the first node v_1 with degree d_1 (minimal among all nodes) to d_1 -many high-degree vertices by emitting edges $\{ \{v_1, v_{n-i}\} \mid 0 \leq i < d_1 \}$. Then obtain an updated sequence \mathcal{D}' by

graphical degree sequence

³EM-LFR directly generates a monotonic degree sequence by first sampling a monotonic uniform sequence [39, 334] and then applying the inverse sampling technique (carrying over the monotonicity) for a powerlaw distribution. Thus, no additional sorting steps are necessary for the inter-community graph.

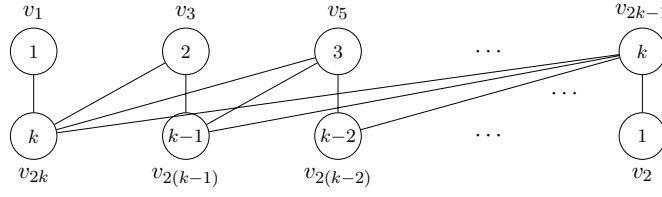


Figure 4.3: Graph with $\mathcal{D}_k = (1, 1, 2, 2, \dots, k, k)$ maximizes EM-HH’s memory consumption asymptotically as $D(\mathcal{D}_k) = k = \Theta(n)$. Node are labeled with their degrees.

removing d_1 from \mathcal{D} and decrementing the remaining degree of every new neighbor $\{v_{n-i} \mid 0 \leq i < d_1\}$.⁴ Subsequently, remove zero-entries and sort \mathcal{D}' while keeping track of the original positions to be able to output the correct node indices. Finally, recurse until no more positive entries remain. After every iteration, the size of \mathcal{D} is reduced by at least one resulting in $\mathcal{O}(n)$ rounds.

For an implementation, it is non-trivial to keep the sequence ordered after decrementing the neighbors’ degrees. Internal memory solutions typically employ priority queues optimized for integer keys, such as bucket-lists [316, 332]. This approach incurs $\Theta(\text{sort}(n + m))$ I/Os using a naïve EM PQ since every edge triggers an update to the pending degree of at least one endpoint.

We hence propose the Havel-Hakimi variant EM-HH which, for virtually all realistic powerlaw degree distributions, avoids accesses to disk besides writing the result. The algorithm emits a stream of edges in lexicographical order which can be fed to any single-pass streaming algorithm without a round trip to disk. Thus, we consider only internal I/Os and emphasize that storing the output –if necessary by the application– requires $\mathcal{O}(m)$ time and $\mathcal{O}(\text{scan}(m))$ I/Os where m is the number of edges produced. Additionally, EM-HH may be used to test in time $\mathcal{O}(n)$ whether a degree sequence \mathcal{D} is graphical or to drop problematic edges yielding a graphical sequence (Section 4.6).

4.4.1 Data Structure

$D(\mathcal{D})$: number of unique degrees

L and g_j

Instead of maintaining the degree of every node in \mathcal{D} individually, EM-HH compacts nodes with equal degrees into a group, yielding $D(\mathcal{D}) := |\{d_i \mid 1 \leq i \leq n\}|$ groups. Since \mathcal{D} is monotonic, such nodes have consecutive ids and the compaction can be performed in a streaming fashion.⁵ The sequence is then stored as a doubly linked list $L = [g_j]_{1 \leq j \leq D(\mathcal{D})}$ where group $g_j = (b_j, n_j, \delta_j)$ encodes that the n_j nodes $[v_{b_j+i}]_{i=0}^{n_j-1}$ have degree δ_j . At the beginning of every iteration of EM-HH, L satisfies the following invariants which guarantee a compact representation:

(I1) The groups contain strictly increasing degrees, i.e. $\delta_j < \delta_{j+1} \quad \forall 1 \leq j < |L|$

(I2) There are no gaps in the node ids, i.e. $b_j + n_j = b_{j+1} \quad \forall 1 \leq j < |L|$

⁴This variant is due to [165]; in [173], the node of maximal degree is picked and connected.

⁵While direct sampling of the group’s multinomial distribution is not beneficial in *LFR*, it may be used to omit the compaction phase for other applications.

These invariants allow us to bound the memory footprint in two steps: first observe that a list L of size $D(\mathcal{D})$ describes a graph with at least $\sum_{i=1}^{D(\mathcal{D})} i/2$ edges due to (I1). Thus, materializing an arbitrary L of size $|L| = \Theta(M)$ emits $\Omega(M^2)$ edges.

Remark 4.1. With as little as 2 GB RAM, this amounts to an edge list exceeding 1 PB in size.⁶ Therefore, even in the worst-case the whole data structure can be kept in IM for all practical scenarios. On top of this, a probabilistic argument applies: while there exist graphs with $D(\mathcal{D}) = \Theta(n)$ as illustrated in Figure 4.3, Lemma 4.2 gives a sub-linear bound on $D(\mathcal{D})$ if \mathcal{D} is sampled from a powerlaw distribution. ◀

in practice EM-HH's state can be kept in IM

Lemma 4.2. Let \mathcal{D} be a degree sequence of n nodes sampled from $\text{PLD}([1, n], \gamma)$. Then, there are $\mathcal{O}(n^{1/\gamma})$ unique degrees $D(\mathcal{D}) = |\{d_i \mid 1 \leq i \leq n\}|$ whp.. ◀

Proof. Consider random variables (X_1, \dots, X_n) sampled i.i.d. from $\text{PLD}([1, n], \gamma)$ as an unordered degree sequence. Fix an index $1 \leq j \leq n$. Due to the powerlaw distribution, X_j is likely to have a small degree. Even if all degrees $1, \dots, n^{1/\gamma}$ were realized, their occurrences would be covered by the claim. Thus, it suffices to bound the number of realized degrees larger than $n^{1/\gamma}$.

We first show that their total probability mass is small. Then we can argue that $D(\mathcal{D})$ is asymptotically unaffected by their rare occurrences:

$$\begin{aligned} \mathbb{P}[X_j > n^{1/\gamma}] &= \sum_{i=n^{1/\gamma}+1}^{n-1} \mathbb{P}[X_j = i] = \frac{\sum_{i=n^{1/\gamma}+1}^{n-1} i^{-\gamma}}{\sum_{i=1}^{n-1} i^{-\gamma}} \stackrel{(i)}{=} \frac{\sum_{i=n^{1/\gamma}+1}^{n-1} i^{-\gamma}}{\zeta(\gamma) - \sum_{i=1}^{n^{1/\gamma}} i^{-\gamma}} \\ &\stackrel{(ii)}{\leq} \frac{\int_{n^{1/\gamma}}^{n-1} x^{-\gamma} dx}{\zeta(\gamma) - \int_n^{\infty} x^{-\gamma} dx} = \frac{\frac{1}{1-\gamma} [(n-1)^{1-\gamma} - n^{1-\gamma}]}{\zeta(\gamma) + \frac{1}{1-\gamma} n^{1-\gamma}} \\ &= \frac{n^{1/\gamma}/n - (n-1)^{1-\gamma}}{(\gamma-1)\zeta(\gamma) - n^{1-\gamma}} = \mathcal{O}(n^{1/\gamma}/n), \end{aligned}$$

where (i) $\zeta(\gamma) = \sum_{i=1}^{\infty} i^{-\gamma}$ is the Riemann zeta function which satisfies $\zeta(\gamma) \geq 1$ for all $\gamma \in \mathbb{R}$, $\gamma \geq 1$. In step (ii), we exploit the series' monotonicity to bound it in between the two integrals $\int_a^{b+1} x^{-\gamma} dx \leq \sum_{i=a}^b i^{-\gamma} \leq \int_{a-1}^b x^{-\gamma} dx$.

In order to bound the number of occurrences, define Boolean indicator variables Y_i with $Y_i = 1$ iff $X_i > n^{1/\gamma}$ and observe that they model Bernoulli trials $Y_i \in B(p)$ with $p = \mathcal{O}(n^{1/\gamma}/n)$. Thus, the expected number of high degrees is $\mathbb{E}[\sum_{i=1}^n Y_i] = \sum_{i=1}^n \mathbb{P}[X_i > n^{1/\gamma}] = \mathcal{O}(n^{1/\gamma})$. Chernoff's inequality gives an exponentially decreasing bound on the tail distribution of the sum which thus holds with high probability. ◻

Remark 4.3. Experiments in Section 4.10.2 indicate that the hidden constants in Lemma 4.2 are small for realistic γ . ◀

⁶A single item of L can be naïvely represented by its three values and two pointers, i.e. a total of $5 \cdot 8 = 40$ bytes per item (assuming 64 bit integers and pointers). Just 2 GB of IM suffice for storing $5 \cdot 10^7$ items, which result in at least $6.25 \cdot 10^{14}$ edges, i.e. storing just two bytes per edge would require more than one Petabyte. Observe that standard tricks, such as exploiting the redundancy due to (I2), allow to reduce the memory footprint of L .

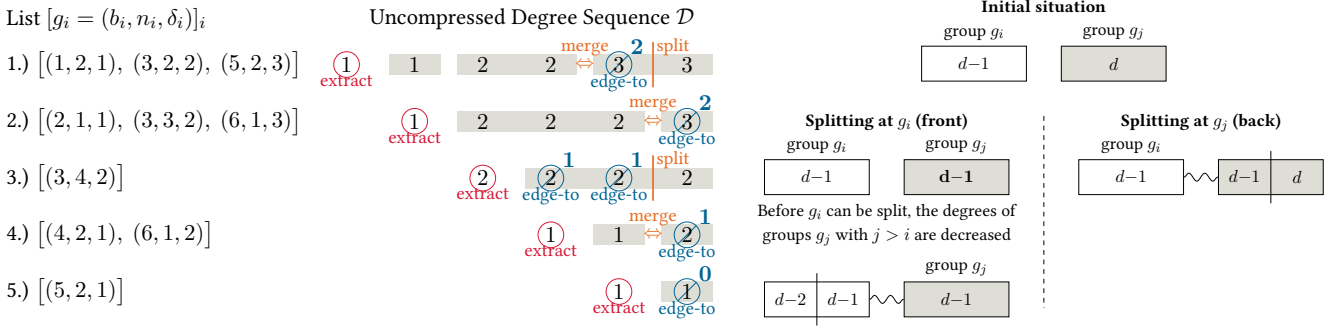


Figure 4.4: Left: EM-HH on $\mathcal{D} = (1, 1, 2, 2, 3, 3)$. L and \mathcal{D} in row i indicate the state at the begin iteration i . The number next to an EDGE-TO symbol indicates the new degree. After these updates, splitting and merging takes place. For instance, in the initial round the first node v_1 is extracted from g_1 and connected to the first node v_5 of the last group. Hence group g_3 of nodes with degree 3 is split, into node v_5 with now $\deg(v_5) = 2$ and v_6 remaining at $\deg(v_6) = 3$. Since group g_2 of nodes $\{v_3, v_4\}$ has also degree 2 it is merged with the new group of v_5 .

Right: Consider two adjacent groups g_i, g_j with degrees $d-1$ and d . A split of g_i (left) or g_j (right) directly triggers a merge, so the number of groups remains the same.

Corollary 4.4. Graphs with $m = \mathcal{O}(M^{2\gamma})$ edges and a powerlaw degree distribution are processed without I/O whp.. \blacktriangleleft

Proof. Due to Lemma 4.2 the number of unique degrees $D(\mathcal{D})$ is bounded by $\mathcal{O}(n^{1/\gamma})$ with high probability. Consequently, a list of size $D(\mathcal{D})$ filling the whole IM supports $n = \mathcal{O}(M^\gamma)$ many nodes and thus $m = \mathcal{O}(M^{2\gamma})$ many edges with high probability. \square

4.4.2 Algorithm

EM-HH works in n rounds, where every iteration corresponds to a recursion step of the original formulation. Each time it extracts node v_{b_1} with the smallest available id and with minimal degree δ_1 . The extraction is achieved by incrementing the lowest node id ($b'_1 \leftarrow b_1 + 1$) of group g_1 and decreasing its size ($n'_1 \leftarrow n_1 - 1$). If the group becomes empty ($n'_1 = 0$), it is removed from L at the end of the iteration; Figure 4.4 illustrates this situation in step 2. We now connect node v_{b_1} to δ_1 nodes from the end of L . Let g_j be the group of smallest index to which v_{b_1} connects to. Then there are two cases:

- (C1): connect to **all** nodes in the group g_j
- (C1) If node v_{b_1} connects to all nodes in g_j , we directly emit all relevant edges $\{[v_{b_1}, x] \mid n - \delta_1 < x \leq n\}$ and decrement the degrees of all groups $g_j, \dots, g_{|L|}$ accordingly. Since degree δ_{j-1} remains unchanged, it may now match the decremented δ_j . This violation of (I1) is resolved by *merging* both groups. Due to (I2), the union of g_{j-1} and g_j contains consecutive ids and it suffices to grow $n_{j-1} \leftarrow n_{j-1} + n_j$ and to delete group g_j (see Figure 4.4 step 2 in which the degree of g_3 is reduced to $d_3 = 2$ triggering a merge with g_2).
- (C2): connect to **some** nodes in the group g_j
- (C2) If v_{b_1} connects only to a number $a < n_j$ of nodes in group g_j , we *split* g_j into two groups g'_j and g''_j containing nodes $[v_{b_j+i}]_{i=0}^{a-1}$ and $[v_{b_j+i}]_{i=a}^{n_j}$ respectively. We then connect node u to all a nodes in the first fragment g'_j and hence need to

decrease its degree. Thus, a merge analogous to (C1) may be required if degree δ_{j-1} matches the decreased degree of group g'_j (see Figure 4.4 step 1 in which group g_3 is split into two fragments with degrees $d_{3'} = 2$ and $d_3 = 3$ respectively, triggering a merge between group g_2 and fragment $g_{3'}$). Afterwards, the degrees of groups $g_{j+1}, \dots, g_{|L|}$ are decreased wholly as in (C1).

If the requested degree δ_1 cannot be met (i.e., $\delta_1 > \sum_{k=1}^{|L|} n_k$), the input is not graphical [165]. However, a sufficiently large random powerlaw degree sequence contains at most very few nodes that cannot be materialized as requested since the vast majority of nodes have low degrees. Thus, we do not explicitly ensure that the sampled degree sequence is graphical and rather correct the negligible inconsistencies later on by ignoring the unsatisfiable requests.

4.4.3 Improving the I/O Complexity

In EM-HH's current formulation, it requires $\mathcal{O}(m)$ time which is already optimal in case edges have to be emitted. Testing whether \mathcal{D} is graphical however is sub-optimal. We thus introduce a simple optimization, which also yields optimality for these tests, improves constant factors and gives I/O-optimal accesses.

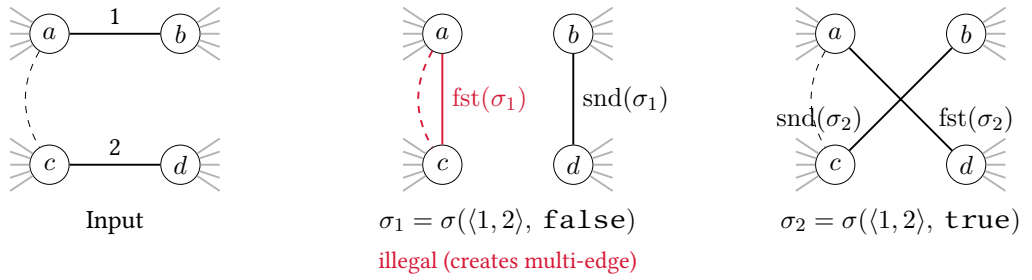
Observe that only groups in the vicinity of g_j can be split or merged; we call these the *active frontier*. In contrast, the so-called *stable* groups $g_{j+1}, \dots, g_{D(\mathcal{D})}$ keep their relative degree differences as the pending degrees of all their nodes are decremented by one in each iteration. Further, they will become neighbors to all subsequently extracted nodes until group g_{j+1} eventually becomes an active merge candidate. Thus, we do not have to update the degrees of stable groups in every round, but rather maintain a single global iteration counter I and count how many iterations a group remained stable: when a group g_k becomes stable in iteration I_0 , we annotate it with I_0 by adding $\delta_k \leftarrow \delta_k + I_0$. If g_k has to be activated again in iteration $I > I_0$, its updated degree is $\delta_k \leftarrow \delta_k - I$. The degree δ_k remains positive since (I1) enforces a timely activation.

Lemma 4.5. The optimized EM-HH needs $\mathcal{O}(\text{scan}(D(\mathcal{D})))$ I/Os if L is an EM list. ◀

Proof. An external memory list requires $\mathcal{O}(\text{scan}(k))$ I/Os to execute any sequence of k sequential read, insertion, and deletion requests to adjacent positions (i.e. if no seeking is necessary) [230]. We will argue that EM-HH scans L roughly twice, starting simultaneously from the front and back.

Every iteration starts by extracting a node of minimal degree. Doing so corresponds to accessing and eventually deleting the list's first element g_i . If the list's head block is cached, we only incur an I/O after deleting $\Theta(B)$ head groups, yielding $\mathcal{O}(\text{scan}(D(\mathcal{D})))$ I/Os during the whole execution. The same is true for accesses to the back of the list: the minimal degree increases monotonically during the algorithm's execution until the extracted node has to be connected to all remaining vertices. In a graphical sequence, this implies that only one group remains and we can ignore the simple base

Figure 4.5: A swap is defined by the two edge ids (rank in E_L), and a direction bit. Swap σ_1 is illegal as it adds the already present edge $\{a, c\}$.



case asymptotically. Neglecting splitting and merging, the distance between the list's head and the *active* frontier decreases monotonically triggering $\mathcal{O}(\text{scan}(D(\mathcal{D})))$ I/Os.

merging

As described before, it may be necessary to reactivate *stable* groups, i.e. to reload the group behind the active frontier (towards L 's end). Thus, we not only keep the block F containing the frontier cached, but also block G behind it. It does not incur additional I/O, since we are scanning backwards through L and already read G before F . The reactivation of *stable* groups hence only incurs an I/O when the whole block G is consumed and deleted. Since this does not happen before $\Omega(B)$ merges take place, reactivations may trigger $\mathcal{O}(\text{scan}(D(\mathcal{D})))$ I/Os in total.

splitting

Splitting does not influence EM-HH's asymptotic I/O complexity: Only an active group of degree d can be split yielding two fragments of degrees $d-1$ and d respectively. A second split of one of these fragments does not increase the number of groups since two of the three involved fragments have to be merged (cf. Figure 4.4). As a result splitting can at most double L 's size. \square

4.5 EM-ES: I/O-efficient Edge Switching

EM-ES implements an external memory edge switching algorithm to randomize networks. Following *LFR*'s original usage of *FDSM*, EM-ES is crucial in EM-LFR to randomize the inter-community graph as well as all communities independently (cf. Figure 4.1), and additionally functions as a building block to rewire illegal edges (cf Sections 4.6 and 4.8). As discussed in Section 4.10.6, the algorithm also has applications as a standalone tool in network analysis.

EM-ES applies a sequence $S = [\sigma_s]_{s=1}^k$ of edge swaps σ_s to a simple graph $G = (V, E)$, where the parameter k is typically chosen as $k \in [1m, 100m]$. The graph is represented by a lexicographically ordered edge list $E_L = [e_i]_{i=1}^m$ which contains for every ordered edge $[u, v] \in E$ (i.e. $u < v$) only the entry (u, v) and omits (v, u) . We encode a swap $\sigma(\langle a, b \rangle, d)$ as a three-tuple with a direction bit d and the two indices a, b of the edges $e_a, e_b \in E_L$ that are supposed to be swapped.

As illustrated in Figure 4.5, a swap simply exchanges one of the two incident nodes of each edge where d selects which one. More formally, we denote the two resulting

edges as $\text{fst}(\sigma(\langle a, b \rangle, d))$ and $\text{snd}(\sigma(\langle a, b \rangle, d))$ with

$$\begin{aligned} \text{fst}(\sigma(\langle a, b \rangle, d)) &:= \begin{cases} \{\alpha_1, \beta_1\} & \text{if } d = \text{FALSE} \\ \{\alpha_1, \beta_2\} & \text{if } d = \text{TRUE} \end{cases}, \quad \text{and} \\ \text{snd}(\sigma(\langle a, b \rangle, d)) &:= \begin{cases} \{\alpha_2, \beta_2\} & \text{if } d = \text{FALSE} \\ \{\alpha_2, \beta_1\} & \text{if } d = \text{TRUE} \end{cases}, \end{aligned}$$

where $[\alpha_1, \alpha_2] = e_a$ and $[\beta_1, \beta_2] = e_b$ are the edges at ranks a and b in the edge list E_L .

In unambiguous cases, we shorten the expressions to $\text{fst}(\sigma)$ and $\text{snd}(\sigma)$ respectively. The swap's constituents a and b are typically drawn independently and uniformly at random. Thus, the sequence can contain illegal swaps that would introduce either multi-edges or self-loops. Such illegal swaps are simply skipped. In order to do so, the following tasks have to be addressed for each $\sigma(\langle a, b \rangle, d)$:

- (T1) Gather the nodes incident to edges e_a and e_b .
- (T2) Compute $\text{fst}(\sigma)$ and $\text{snd}(\sigma)$ and skip if a self-loop arises.
- (T3) Verify that the graph remains simple, i.e. skip if edge $\text{fst}(\sigma)$ or $\text{snd}(\sigma)$ already exist in E_L .
- (T4) Update the graph representation E_L .

If the whole graph fits in IM, a hash set per node storing all neighbors can be used for adjacency queries and updates in expected constant time (e.g., VL-ES [332]). Then, (T3) and (T4) can be executed for each swap in expected time $\mathcal{O}(1)$. However, in the EM model this approach incurs $\Omega(1)$ I/Os per swap with high probability for a graph with $m \geq cM$ and any constant $c > 1$.

We address this issue by processing the sequence of swaps S batchwise in chunks of size $r = \Theta(m)$ which we call *runs*. As illustrated in Figure 4.6, EM-ES executes several phases for each run. While they roughly correspond to the four tasks outlined above, the algorithm is more involved as it has to explicitly track data dependencies between swaps within a batch. There are two types: A *source edge dependency* occurs if (at least) two swaps share the same edge id as source. In this case, successfully executing the first swap will replace the edge by another one. This update has to be communicated to all later swaps involving this edge id. *Target edge dependencies* exist because swaps must not introduce multi-edges. Therefore each swap has to assert that none of its new edges (target edges) are already present in the graph. For this reason, EM-ES has to inform a swap about the creation or deletion of target edges that occurred earlier in the run.

4.5.1 EM-ES for Independent Swaps

For simplicity's sake, we first assume that all swaps are independent, i.e. that there are neither *source edge* nor *target edge* dependencies in a run. Section 4.5.6 contains the algorithmic modifications necessary to account for dependencies.

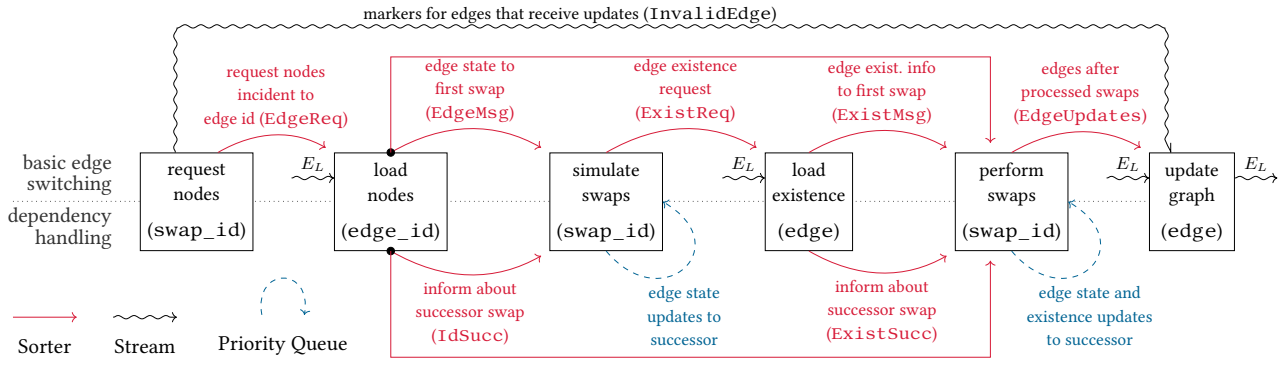


Figure 4.6: Data flow of EM-ES. Communication between phases is uses EM sorters, self-loops use a PQs (*TFP*). Brackets within a phase indicate the elements iterated over. If multiple input streams are used, they are joined with this key. Independent swaps as in Section 4.5.1 require only communication via sorters as shown on the upper half.

The design of EM-ES is driven by the intuition that there are three types of cross-referenced data, namely (i) the sequence of swaps ranked in the order they were issued, (ii) edges addressed by their indices (e.g., to load and store their incident nodes) and (iii) edges referenced by their constituents (in order to query their existence). To resolve these unstructured references, the algorithm is decomposed into several phases and iterates in each phase over one of these data types in order. There is no pipelining, so a new phase only starts processing when the previous is completed. Similarly to *Time Forward Processing*, phases communicate by sending messages addressed to the key of the receiving phase. The messages are pushed into a sorter⁷ to later be processed in the order dictated by the data source of the receiving end. EM-ES uses the following phases:

4.5.2 Phases *Request nodes* and *load nodes*

The goal of these two phases is to load the constituents of the edges referenced by the run’s swaps. We iterate over the sequence S of swaps. For the s -th swap $\sigma(\langle a, b \rangle, d)$, we push two messages $\text{EDGE_REQ}(a, s, 0)$ and $\text{EDGE_REQ}(b, s, 1)$ into the sorter EDGEREQ . A message’s third entry encodes whether the request is issued for the first or second edge of a swap. This information only becomes relevant when we allow dependencies. EM-ES then scans in parallel through the edge list E_L and the requests EDGEREQ , which are now sorted by edge ids. If there is a request $\text{EDGE_REQ}(i, s, p)$ for an edge $e_i = [u, v]$, the edge’s node pair is sent to the requesting swap by pushing a message $\text{EDGE_MSG}(s, p, (u, v))$ into the sorter EDGEMSG .

Additionally, for every edge we push a bit into the sequence INVALIDEDGE indicating whether an edge received a request. Such edges will be deleted when updating the graph in Section 4.5.5. Since both phases produce only a constant amount of data per input

⁷The term *sorter* refers to a data structure with two modes of operation: items are first pushed into the write-only sorter in an arbitrary order by some algorithm. After an explicit switch, the filled data structure becomes read-only and the elements are provided as a lexicographically non-decreasing stream. It can be rewound at any time. While a sorter is functionally equivalent to sorting an EM vector, the restricted access model reduces constant factors in the implementation’s runtime and I/O complexity [37].

element, we obtain an I/O complexity of $\mathcal{O}(\text{sort}(r) + \text{scan}(m))$.

4.5.3 Phases *Simulate swaps and load existence*

The two phases gather all information required to decide whether a swap is legal. EM-ES scans through the sequence S of swaps and EDGEMSG in parallel: For the s -th swap $\sigma(\langle a, b \rangle, d)$, there are exactly two messages $\text{EDGE_MSG}(s, 0, e_a)$ and $\text{EDGE_MSG}(s, 1, e_b)$ in EDGEMSG. This information suffices to compute the switched edges $\text{fst}(\sigma)$ and $\text{snd}(\sigma)$, but not to test for multi-edges.

It remains to check whether the switched edges already exist; we push the existence requests $\text{EXIST_REQ}(\text{fst}(\sigma), s)$ and $\text{EXIST_REQ}(\text{snd}(\sigma), s)$ into the sorter EXISTREQ. Observe that for *request nodes* we use the node pairs rather than edge ids, which are not well defined here. Afterwards, a parallel scan through the edge list E_L and EXISTREQ is performed to answer the requests. Only if an edge e requested by swap id s is found, the message $\text{EXIST_MSG}(s, e)$ is pushed into the sorter EXISTMSG. Both phases hence incur a total of $\mathcal{O}(\text{sort}(r) + \text{scan}(m))$ I/Os.

4.5.4 Phase *Perform swaps*

We rewind the EDGEMSG sorter and jointly scan through the sequence of swaps S and the sorters EDGEMSG and EXISTMSG. As described in the simulation phase, EM-ES computes the switched edges $\text{fst}(\sigma)$ and $\text{snd}(\sigma)$ from the original state e_a and e_b . The swap is considered illegal if a switched edge is a self-loop or if an existence info is received via EXISTMSG. If σ is legal we push the switched edges $\text{fst}(\sigma)$ and $\text{snd}(\sigma)$ into the sorter EDGEUPDATES, otherwise we propagate the unaltered source edges e_a and e_b . This phase requires $\mathcal{O}(\text{sort}(r))$ I/Os.

4.5.5 Phase *Update edge list*

The new edge list E'_L is obtained by merging the original lexicographic increasing list E_L and the sorted updated edges EDGEUPDATES, triggering $\mathcal{O}(\text{scan}(m))$ I/Os. During this process, we skip all edges in E_L that are flagged invalid in the bit stream INVALIDEDGE. The result is a sorted new E'_L with $|E'_L| = m$ edges that can be fed into the next run.

4.5.6 Phase *Inter-Swap Dependencies*

In this section, we introduce the modifications necessary due to dependencies between swaps within a run. In its final version, EM-ES produces the same result as a sequential processing of S . Source edge dependencies are detected during the *load nodes* phase since multiple requests for the same edge id arrive. We record these dependencies as an explicit dependency chain along which intermediate updates can be propagated. Target edge dependencies surface in the *load existence* phase since multiple existence requests and notifications arrive for the same edge. Again, an explicit dependency chain is computed. During the *perform swaps* phase, EM-ES uses both dependency chains to forward the source edge states and existence updates to successor swaps.

4.5.7 Target Edge Dependencies

Consider the case where a swap $\sigma_{s_1}(\langle a, b \rangle, d)$ changes the state of edges e_a and e_b to $\text{fst}(\sigma_1)$ and $\text{snd}(\sigma_1)$ respectively. Later, a second swap σ_2 inquires about the existence of either of the four edges which has obviously changed compared to the initial state. We extend the simulation phase to track such edge modifications and not only push messages $\text{EXIST_REQ}(\text{fst}(\sigma_1), s_1)$ and $\text{EXIST_REQ}(\text{snd}(\sigma_1), s_1)$ into sorter `EDGE_REQ`, but also report that the original edges may change (during simulation phase it is unknown whether the swap has to be skipped). To this end, we push the messages $\text{EXIST_REQ}(e_a, s_1, \text{MAY_CHANGE})$ and $\text{EXIST_REQ}(e_b, s_1, \text{MAY_CHANGE})$ into the same sorter.

In case of dependencies, multiple messages are received for the same edge e during the *load existence* phase. If so, only the request of the first swap involved is answered as before. Also, every swap σ_{s_1} is informed about its direct successor σ_{s_2} (if any) by pushing the message $\text{exist_succ}(s_1, e, s_2)$ into the sorter `EXIST_SUCC`, yielding the aforementioned dependency chain. As an optimization, `MAY_CHANGE` requests at the end of a chain are discarded since no recipient exists.

During the *perform swaps* phase, EM-ES executes the same steps as described earlier. The swap may receive a successor from every edge it sent an existence request to, and—in turn— send each successor swapped edge state.

4.5.8 Source Edge Dependencies

Consider two swaps $\sigma_{s_1}(\langle a_1, b_1 \rangle, d_1)$ and $\sigma_{s_2}(\langle a_2, b_2 \rangle, d_2)$ with $s_1 < s_2$ which share a source edge id, i.e. $\{a_1, b_1\} \cap \{a_2, b_2\}$ is non-empty. This dependency is detected during the *load nodes* phase since requests $\text{EDGE_REQ}(e_i, s_1, p_1)$ and $\text{EDGE_REQ}(e_i, s_2, p_2)$ arrive for edge id e_i . In this case, we answer only the request of s_1 and build a dependency chain as before using messages $\text{ID_SUCC}(s_1, p_1, s_2, p_2)$ pushed into the sorter `ID_SUCC`.

During the simulation phase, EM-ES cannot yet decide whether a swap is legal. Thus, s_1 sends for every conflicting edge its original state as well as the updated state to the p_2 -th slot of s_2 using a PQ. If a swap receives multiple edge states per slot, it simulates the swap for all possible combinations.

During the *perform swaps* phase, EM-ES operates as described in the independent case: it computes the swapped edges and determines whether the swap has to be skipped. If a successor exists, the new state is not pushed into the `EDGE_UPDATES` sorter but rather forwarded to the successor in a *TFP* fashion. This way, every invalidated edge id receives exactly one update in `EDGE_UPDATES` and the merging remains correct.

4.5.9 Complexity

Due to source edge dependencies, EM-ES's complexity increases with the number of swaps that share the same edge id. This number is low in case $r = \mathcal{O}(m)$: let X_i be a random variable expressing the number of swaps that reference edge e_i . Since every swap constitutes two independent Bernoulli trials towards e_i , the indicator X_i is binomially distributed with $p = 1/m$, yielding an expected chain length of $2r/m$. Also, for $r = m/2$ swaps, $\max_{1 \leq i \leq n}(X_i) = \mathcal{O}(\ln(m)/\ln \ln(m))$ holds with high

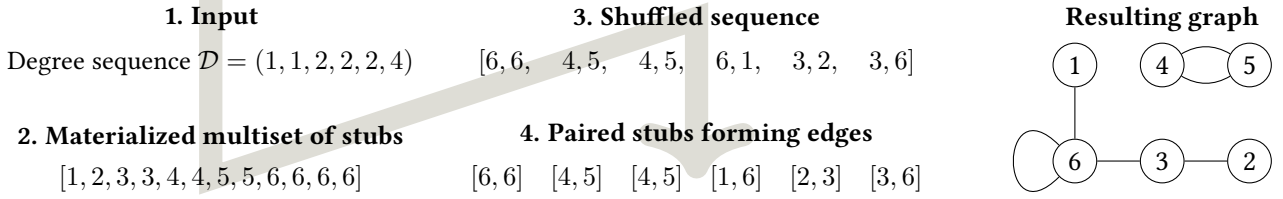


Figure 4.7: Possible execution path of *Configuration Model* with the degree sequence $\mathcal{D} = (1, 1, 2, 2, 2, 4)$ as input.

probability based on a balls-into-bins argument [252]. Thus, we can bound the largest number of edge states simulated with high probability by $\mathcal{O}(\text{poly } \log(m))$, assuming non-overlapping dependency chains. Further observe that X_i converges towards an independent Poisson distribution for large m . Then the expected state space per edge is $\mathcal{O}(1)$. The experiments in Section 4.10.3 suggest that this bound also holds for overlapping dependency chains.

In order to keep the dependency chains short, EM-ES splits the sequence of swaps S into runs of equal size. Our experimental results show that a run size of $r = m/8$ is a suitable choice. For every run, the algorithm executes the six phases as described before. Each time the graph is updated, the mapping between an edge and its id may change. The switching probabilities, however, remain unaltered due to the initial assumption of uniformly distributed swaps. Thus EM-ES triggers $\mathcal{O}(r/m \text{ sort}(m))$ I/Os in total whp.

4.6 EM-CM/ES: Sampling graphs with Prescribed Degree Sequence

In this section, we propose an alternative approach to generate a graph from a prescribed degree sequence. In contrast to EM-HH which generates a highly biased but simple graph, we use the Configuration Model to sample a random but in general non-simple graph. Thus, the resulting graph may contain self-loops and multi-edges which we then rewire to obtain a simple graph. As experimental data suggests (cf. Section 4.6.2), this still results in a biased realization of the degree sequence requiring additional edge switching randomization steps.

4.6.1 Configuration Model

Let $\mathcal{D} = [d_i]_{i=1}^n$ be a degree sequence with n nodes. The Configuration Model builds a multiset of node ids which can be thought of as *half-edges* (or stubs). It produces a total of d_i *half-edges* labeled v_i for each node v_i . The algorithm then chooses two half-edges uniformly at random and creates an edge according to their labels. It repeats the last step with the remaining half-edges until all are paired. A naïve implementation of this algorithm requires with high probability $\Omega(m)$ I/Os if $m \geq cM$ and any constant $c > 1$. It is therefore impractical in the fully external setting.

We rather materialize the multiset as a sequence in which each node appears d_i times similar to the approach of [201]. Subsequently, the sequence is shuffled to obtain a random permutation with $\mathcal{O}(\text{sort}(m))$ I/Os by sorting the sequence according to a

uniform variate drawn for each half-edge⁸. Finally, we scan over the shuffled sequence and match pairs of adjacent half-edges into edges.

As illustrated in Figure 4.7, the Configuration Model gives rise to self-loops and multi-edges which then need to be rewired, cf. section 4.6.2. Consequently, the rewiring process depends on the number of introduced illegal edges. In the following lemma, we bound their number from above.

Lemma 4.6. Let \mathcal{D} be drawn from $\text{PLD}([a, b], 2)$. The expected number of self-loops and multi-edges are bound by:

$$\begin{aligned} \mathbb{E}[\#\text{self-loops}] &\leq \frac{1}{2} \left(\frac{b-a+1}{\ln(b+1) - \ln(a)} \right) \\ \mathbb{E}[\#\text{multi-edges}] &\leq \frac{1}{2} \left(\frac{b-a+1}{\ln(b+1) - \ln(a)} \right)^2 \quad \blacktriangleleft \end{aligned}$$

Proof. For an arbitrary degree sequence \mathcal{D} , [20] and [260] derive expectation values in terms of \mathcal{D} 's mean $\langle \mathcal{D} \rangle$ and its second moment $\langle \mathcal{D}^2 \rangle$. For $n \rightarrow \infty$, the authors show:

$$\mathbb{E}[\#\text{self-loops}(\mathcal{D})] = \frac{\langle \mathcal{D}^2 \rangle - \langle \mathcal{D} \rangle}{2(\langle \mathcal{D} \rangle - \frac{1}{n})} \rightarrow \frac{\langle \mathcal{D}^2 \rangle - \langle \mathcal{D} \rangle}{2\langle \mathcal{D} \rangle} \quad (4.1)$$

$$\mathbb{E}[\#\text{multi-edges}(\mathcal{D})] \leq \frac{1}{2} \left(\frac{(\langle \mathcal{D}^2 \rangle - \langle \mathcal{D} \rangle)^2}{(\langle \mathcal{D} \rangle - \frac{1}{n})(\langle \mathcal{D} \rangle - \frac{3}{n})} \right) \rightarrow \frac{1}{2} \left(\frac{\langle \mathcal{D}^2 \rangle - \langle \mathcal{D} \rangle}{\langle \mathcal{D} \rangle} \right)^2 \quad (4.2)$$

We now bound $\langle \mathcal{D} \rangle$ and $\langle \mathcal{D}^2 \rangle$ in the case that \mathcal{D} is drawn from the powerlaw distribution $\text{PLD}([a, b], \gamma)$. Since each entry in \mathcal{D} is independently drawn, it suffices to bound the expected value and the second moment of the underlying distribution. Let $C_{\mathcal{D}} = \sum_{i=a}^b i^{-\gamma}$, then they follow as:

$$\langle \mathcal{D} \rangle = \sum_{i=a}^b i^{-\gamma+1} / C_{\mathcal{D}} \quad \text{and} \quad \langle \mathcal{D}^2 \rangle = \sum_{i=a}^b i^{-\gamma+2} / C_{\mathcal{D}}$$

Both numerators are sandwiched between $\int_a^{b+1} x^q dx \leq \sum_{i=a}^b i^q \leq \int_{a-1}^b x^q dx$ where $q = 1 - \gamma$ or $q = 2 - \gamma$, respectively. In the case of $\gamma = 2$, the second moment hence simplified to $\langle \mathcal{D}^2 \rangle = (\sum_{i=a}^b 1) / C_{\mathcal{D}} = (b-a+1) / C_{\mathcal{D}}$. Applying this identity and the lower bound $(\int_a^{b+1} x^{-1} dx) / C_{\mathcal{D}} \leq \langle \mathcal{D} \rangle$ to Section 4.6.1, directly yields the claim. \square

4.6.2 Edge Rewiring for Non-Simple Graphs

Graphs generated using the Configuration Model may contain multi-edges and self-loops. In order to obtain a simple graph we need to detect these illegal edges and rewire them. After sorting the edge list lexicographically, illegal edges can be detected in a single scan. For each self-loop we issue a swap with a randomly selected partner edge. Similarly, for each group of parallel edges, we generate swaps with random partner

⁸If $M > \sqrt{mB}(1+o(1)) + \mathcal{O}(B)$ this can be improved to $\mathcal{O}(\text{scan}(m))$ I/Os [290] which does however not affect the total complexity of our pipeline.

edges for all but one multi-edge. Subsequently, we execute the provisioned swaps using a variant of EM-ES (see below). The process is repeated until all illegal edges have been removed. To accelerate the endgame, we double the number of swaps for each remaining illegal edge in every iteration.

Since EM-ES is employed to remove parallel edges based on targeted swaps, it needs to process non-simple graphs. Analogous to the initial formulation, we forbid swaps that introduce multi-edges even if they would reduce the multiplicity of another edge (cf. [353]). Nevertheless, EM-ES requires slight modifications for non-simple graphs.

Consider the case where the existence of a multi-edge is inquired several times. Since E_L is sorted, the initial edge multiplicities can be counted while scanning E_L during the *load existence* phase. In order to correctly process the dependency chain, we have to forward the (possibly updated) multiplicity information to successor swaps. We annotate the existence tokens `exist_msg($s, e, \#(e)$)` with these counters where $\#(e)$ is the multiplicity of edge e .

More precisely, during the *perform swaps* phase, swap $\sigma_1 = \sigma(\langle a, b \rangle, d)$ is informed (among others) of multiplicities of edges $e_a, e_b, \text{fst}(\sigma_1)$ and $\text{snd}(\sigma_1)$ by incoming existence messages. If σ_1 is legal, we send requested edges and multiplicities of the swapped state to any successor σ_2 of σ_1 provided in `EXISTSUCC`. Otherwise, we forward the edges and multiplicities of the unchanged initial state. As an optimization, edges which have been removed (i.e. have multiplicity zero) are omitted.

4.7 EM-CA: Community Assignment

In the *LFR* benchmark, every node belongs to one (non-overlapping) or more (overlapping) communities. EM-CA finds such a random assignment subject to the two constraints that all communities get as many nodes as previously determined (see Figure 4.1) and that for a node v_i all its assigned communities have enough other members to satisfy the node's intra-community degree d_i^{in}/ν_i .

For the sake of simplicity, we first restrict ourselves to the non-overlapping case, in which every node belongs to exactly one community. Consider a sequence of community sizes $S = [s_\xi]_{\xi=1}^C$ with $n = \sum_{\xi=1}^C s_\xi$ and a sequence of intra-community degrees $\mathcal{D} = [d_i^{\text{in}}]_{i=1}^n$. Let S and \mathcal{D} be non-decreasing and positive. The task is to find a random surjective assignment $\chi: V \rightarrow [C]$ with:

- (R1) Every community ξ is assigned s_ξ nodes as requested, with

$$s_\xi := |\{v \mid v \in V \wedge \chi(v) = \xi\}|.$$
- (R2) Every node $v \in V$ becomes member of a sufficiently large community $\chi(v)$ with

$$s_{\chi(v)} > d_v^{\text{in}}.$$

Observe that χ can be interpreted as a bipartite graph where the partition classes are given by the communities $[C]$ and nodes $[v_i]_{i=1}^n$ respectively, and each edge corresponds to an assignment.

4.7.1 A Simple, Iterative, But not yet Complete Algorithm

To ease the description of the algorithm, let us also ignore (R2) for now, and discuss the changes needed in Section 4.7.2. Then the assignment graph can be sampled in the spirit of the Configuration Model (cf. Section 4.6). To do so, we draw a permutation π of nodes uniformly at random, and assign nodes $[v_{\pi(i)}]_{i=x_\xi+1}^{x_\xi+s_\xi}$ to community ξ where $x_\xi := \sum_{i=1}^{\xi-1} s_i$ is the number of slots required for communities with indices below ξ .

To ease later modifications, we prefer an equivalent iterative formulation: while there exists a yet unassigned node u , draw a community X with probability proportional to the number of its remaining free slots (i.e. $\mathbb{P}[X=\xi] \propto s_\xi$). Assign node u to X , reduce the community's probability mass by decreasing $s_X \leftarrow s_X - 1$ and repeat. By construction, the first scheme is unbiased and the equivalence of both approaches follows as a special case of Lemma 4.7 (see below).

We implement the random selection process efficiently based on a binary tree where each community corresponds to a leaf with a weight equal to the number of free slots in the community. Inner nodes store the total weight of their left subtree. In order to draw a community, we sample an integer $Y \in [0, W_C)$ uniformly at random where $W_C := \sum_{\xi=1}^C s_\xi$ is the tree's total weight. Following the tree according to Y yields the leaf corresponding to community X . An I/O-efficient data structure [239] based on lazy evaluation for such dynamic probability distributions enables a fully external algorithm with $\mathcal{O}\left(n/B \cdot \log_{M/B}(C/B)\right) = \mathcal{O}(\text{sort}(n))$ I/Os. However, if $C < M$, we can store the tree in IM, allowing a semi-external algorithm which only needs to scan through \mathcal{D} , triggering $\mathcal{O}(\text{scan}(n))$ I/Os.

4.7.2 Enforcing Constraint on Community Size (R2)

To enforce (R2), we additionally ensure that all nodes are assigned to a sufficiently large community such that they find enough neighbors to connect to. We exploit that S and \mathcal{D} are non-decreasing and define $p_v := \max\{\xi \mid s_\xi > d_v^{\text{in}}\}$ as the index of the smallest community node v may be assigned to. Since $[p_v]_v$ is therefore monotonic itself, it can be computed online with $\mathcal{O}(1)$ additional IM and $\mathcal{O}(\text{scan}(n))$ I/Os in the fully external setting by scanning through S and \mathcal{D} in parallel. To restrict the random sampling to the communities $\{1, \dots, p_v\}$, we reduce the aforementioned random interval to $[0, W_v)$ where the partial sum $W_v := \sum_{\xi=1}^{p_v-1} s_\xi$ is available while computing p_v . We generalize the notation of uniform assignments subject to (R2) as follows:

Lemma 4.7. Given $S = [s_\xi]_{\xi=1}^C$ and \mathcal{D} , let $u, v \in V$ be two nodes with the same constraints $p_u = p_v$ and let c be an arbitrary community. Further, let χ be an assignment generated by EM-CA. Then, $\mathbb{P}[\chi(u)=c] = \mathbb{P}[\chi(v)=c]$. ◀

Proof. Without loss of generality, assume that $p_u = p_1$, i.e. u is one of the nodes with the tightest constraints. If this is not the case, we just execute EM-CA until we reach a node u' which has the same constraints as u does (i.e. $p_{u'} = p_u$), and apply the Lemma inductively. This is legal since EM-CA streams through \mathcal{D} in a single pass, and

is oblivious to any future values. In case $c > p_1$, neither u nor v can become a member of c . Therefore, $\mathbb{P}[\chi(u)=c] = \mathbb{P}[\chi(v)=c] = 0$ and the claim follows trivially.

Now consider the case $c \leq p_1$. Let $s_{c,i}$ be the number of free slots in community c at the beginning of round $i \geq 1$ and $\mathcal{W}_i = \sum_{j=1}^C s_{j,i}$ their sum at that time. By definition, EM-CA assigns node u to community c with probability $\mathbb{P}[\chi(u)=c] = s_{c,u}/\mathcal{W}_u$. Further, the algorithm has to update the number of free slots. Thus, initially we have $s_{1c} = s_c$ and for iteration $1 < i \leq n$ it holds that

$$s_{,i}c = \begin{cases} s_c^{(i-1)} - 1 & \text{if } v_{i-1} \text{ was assigned to } c \\ s_c^{(i-1)} & \text{otherwise} \end{cases}.$$

The number of free slots is reduced by one in each step $\mathcal{W}_i = \mathcal{W}_1 - i + 1 = \left(\sum_{j=1}^C S_j\right) - i + 1$. The claim follows by transitivity if we show $\mathbb{P}[\chi(u)=c] = s_{c,u}/\mathcal{W}_u = s_{c,1}/\mathcal{W}_1$. For $u = 1$ it holds by definition. Now, consider the induction step for $u > 1$:

$$\begin{aligned} \mathbb{P}[\chi(u)=c] &= s_{c,u}/\mathcal{W}_u = \mathbb{P}[\chi(u-1)=c] \frac{s_{c,u-1} - 1}{\mathcal{W}_u} + \mathbb{P}[\chi(u-1) \neq c] \frac{s_{c,u-1}}{\mathcal{W}_u} \\ &= \frac{s_{c,u-1}}{\mathcal{W}_{u-1}} \frac{s_{c,u-1} - 1}{\mathcal{W}_u} + \left(1 - \frac{s_{c,u-1}}{\mathcal{W}_{u-1}}\right) \frac{s_{c,u-1}}{\mathcal{W}_u} \\ &= \frac{s_{c,u-1} \cdot \mathcal{W}_{u-1} - s_{c,u-1}}{\mathcal{W}_{u-1} \cdot \mathcal{W}_u} = \frac{s_{c,u-1}(\mathcal{W}_{u-1} - 1)}{\mathcal{W}_{u-1} \cdot (\mathcal{W}_{u-1} - 1)} \\ &= \frac{s_{c,u-1}}{\mathcal{W}_{u-1}} \stackrel{\text{Ind. Hyp.}}{=} \frac{s_{c,1}}{\mathcal{W}_1} \quad \square \end{aligned}$$

4.7.3 Assignment with Overlapping Communities

In the overlapping case, the weight of S increases to account for nodes with multiple memberships. There is further an additional input sequence $[\nu_i]_{i=1}^n$ corresponding to the number of memberships node v_i shall have, each of which has d_i^m/ν_i intra-community neighbors. We then sample not only one community per node v_i , but ν_i different ones.

Since the number of memberships $\nu_v \ll M$ is small, a duplication check during the repeated sampling is easy in the semi-external case and does not change the I/O complexity. However, it is possible that near the end of the execution there are less free communities than memberships requested. We address this issue by switching to an offline strategy for the last $\Theta(M)$ assignments and keep them in IM. As $\nu = \mathcal{O}(1)$, there are $\Omega(\nu)$ communities with free slots for the last $\Theta(M)$ vertices and a legal assignment exists with high probability. The offline strategy proceeds as before until it is unable to find ν different communities for a node. In that case, it randomly picks earlier assignments until swapping the communities is possible.

In the fully external setting, the I/O complexity grows linearly in the number of samples taken and is thus bounded by $\mathcal{O}(\nu \text{ sort}(n))$. However, the community memberships are obtained lazily and out-of-order which may assign a node several times to the same community. This corresponds to a multi-edge in the bipartite assignment graph. It can be removed using the rewiring technique detailed in Section 4.6.2.

4.8 EM-GER/EM-CER: Merging Intra- and Inter-Community Graphs

As illustrated in Figure 4.1, *LFR* samples the inter-community graph and all intra-community graphs independently. As a result, they may exhibit minor inconsistencies which EM-LFR resolves in accordance with the original version by applying additional rewiring steps which are discussed in this section.

4.8.1 EM-GER: Global Edge Rewiring

The global graph is materialized without taking the community structure into account. As illustrated in Figure 4.2 (center), it therefore can contain edges between nodes that share a community. Those edges have to be removed as they decrease the mixing parameter μ . We rewire these edges by performing an edge swap for each forbidden edge with a randomly selected partner. Since it is unlikely that such a random swap introduces another illegal edge (if sufficiently many communities exist), this probabilistic approach effectively removes forbidden edges. We apply this idea iteratively and perform multiple rounds until no forbidden edges remain.

To detect illegal edges, EM-GER considers the community assignment's output which is a lexicographically ordered sequence χ of (v, ξ) -pairs containing the community ξ for each node v . For nodes that join multiple communities several such pairs exist. Based on this, we annotate every edge with the communities of both incident vertices by scanning through the edge list twice: once sorted by source nodes and once by target nodes. For each forbidden edge, a swap is generated by drawing a random partner edge id and a swap direction. Subsequently, all swaps are executed using EM-ES which now also emits the set of edges involved. It suffices to restrict the scan for illegal edges to this set since all edges not contained are legal by construction.

Complexity. Each round requires $\mathcal{O}(\text{sort}(m))$ I/Os for selecting the edges and executing the swaps. The number of rounds is usually small but depends on the community size distribution: the probability that a randomly placed edge lies within a community increases with the size of the community.

4.8.2 EM-CER: Community Edge Rewiring

In the case of overlapping communities, the same edge can be generated as part of multiple communities. We iteratively apply semi-random swaps to remove those parallel edges similarly to Sections 4.6.2 and 4.8. The selection of random partners is however more involved for EM-CER as it has to ensure that all swaps take place between two edges of the same community. This way, the rewired edges keep the same memberships as their sources and the community sizes do not change. The rewiring itself is easy to achieve by considering all communities independently.

Unfortunately, EM-CER needs to process all communities conjointly to detect forbidden edges: we augment each edge $[u_i, v_i]$ with its community id c_i and concatenate these lists into one annotated graph possibly containing multi-edges. During a scan through the lexicographically sorted and annotated edge list $[(u_i, v_i, c_i)]_i$, parallel edges are

easily found as they appear next to each other. We select all but one from each group for rewiring. Each partner is selected by a uniform edge id e_b addressing the e_b -th edge of the community at hand. In a fully external setting, it suffices to sort the selected candidates, their partners and the edge list by community to gather all information required to invoke EM-ES.

EM-CER avoids the expensive step of sorting all edges if we can store $O(1)$ items per illegal edge in IM (which is almost certainly the case since there are typically few illegal edges). It then sorts the edge ids of partners for every community independently and keeps pointers to the smallest requested partner edge id of each community. While scanning through the concatenated edge list, we count for each community the number of edges seen so far. When the counter matches the smallest requested id of the current edge's community, we load the edge and advance the pointer to the next request.

Complexity. The fully external rewiring requires $\mathcal{O}(\text{sort}(m))$ I/Os for the initial step and each following round. The semi-external variant triggers only $\mathcal{O}(\text{scan}(m))$ I/Os per round. The number of rounds is usually small and the overall runtime spent on this step is insignificant. Nevertheless, the described scheme is a Las-Vegas algorithm and there exist (unlikely) instances on which it will fail.⁹ To mitigate this issue, we allow a small fraction of edges (e.g., 10^{-3}) to be removed if we detect a slow convergence. To speed up the endgame, we also draw additional swaps uniformly at random from communities which contain a multi-edge.

4.9 Implementation

We implemented the proposed algorithms in C++ based on the STXXL library [102], providing implementations of EM data structures, a parallel EM sorter, and an EM priority queue. Among others, we applied the following optimizations for EM-ES:

- Most message types contain both a swap id and a flag indicating which of the swap's edges is targeted. We encode both of them in a single integer by using all but the least significant bit for the swap id and store the flag in there. This significantly reduces the memory volume and yields a simpler comparison operator since the standard integer comparison already ensures the correct lexicographic order.
- Instead of storing and reading the sequence of swaps several times, we exploit the implementation's pipeline structure and directly issue edge id requests for every arriving swap. Since this is the only time edge ids are read from a swap, only the remaining direction flag is stored in an efficient EM vector, which uses one bit per flag and supports I/O-efficient writing and reading. Both steps can be overlapped with an ongoing EM-ES run.
- Instead of storing each edge in the sorted external edge list as a pair of nodes, we only store each source node once and then list all targets of that node. This still

⁹Consider a node which is a member of two communities in which it is connected to all other nodes. If only one of its neighbors also appears in both communities, the multi-edge cannot be rewired.

supports sequential scan and merge operations which are the only operations we need. This almost halves the I/O volume of scanning or updating the edge list.

- During the execution of several runs we can delay the updating of the edge list and combine it with the *load nodes* phase of the next run. This reduces the number of scans per additional run from three to two.
- We use asynchronous stream adapters for tasks such as streaming from sorters or the generation of random numbers. These adapters run in parallel in the background to preprocess and buffer portions of the stream in advance and hand them over to the main thread.

Besides parallel sorting and asynchronous pipeline stages, the current EM-LFR implementation facilitates parallelism during the generation and randomization of intra-community graphs which can be computed without any synchronization. While the algorithms themselves are sequential, this pipelining and parallelization of independent tasks within EM-LFR leads to a consistent utilization of available threads in our test system (cf. Section 4.10).

4.10 Experimental Results

4.10.1 Notation and Setup

The number of repetitions per data point (with different random seeds) is denoted with S . Error bars correspond to the unbiased estimation of the standard deviation. For *LFR* we perform experiments based on two different scenarios:

- **lin** – The maximal degrees and community sizes scale linearly as a function of n . For a particular n and ν the parameters are chosen as: $\mu \in \{0.2, 0.4, 0.6\}$, $d_{\min}=10\nu$, $d_{\max}=n\nu/20$, $\gamma=2$, $s_{\min}=20$, $s_{\max}=n/10$, $\beta=1$, $O=n$.
- **const** – We keep the community sizes and the degrees constant and consider only non-overlapping communities. The parameters are chosen as: $d_{\min}=50$, $d_{\max}=10,000$, $\gamma=2$, $s_{\min}=50$, $s_{\max}=12,000$, $\beta=1$, $O=n$.

Real-world networks have been shown to have increasing average degrees as they become larger [215]. Increasing the maximum degree as in our first setting **lin** increases the average degree. Having a maximum community size of $n/10$ means, however, that a significant proportion of the nodes belongs to huge communities which are not very tightly knit due to the large number of nodes of low degree. While a more limited growth is probably more realistic, the exact parameters depend on the network model.

Our second parameter set **const** shows an example of much smaller maximum degrees and community sizes. We chose the parameters such that they approximate the degree distribution of the Facebook network in May 2011 when it consisted of 721 million active users as reported in [328]. The same study however found that strict powerlaw models are unable to accurately mimic Facebook’s degree distribution. Further, the

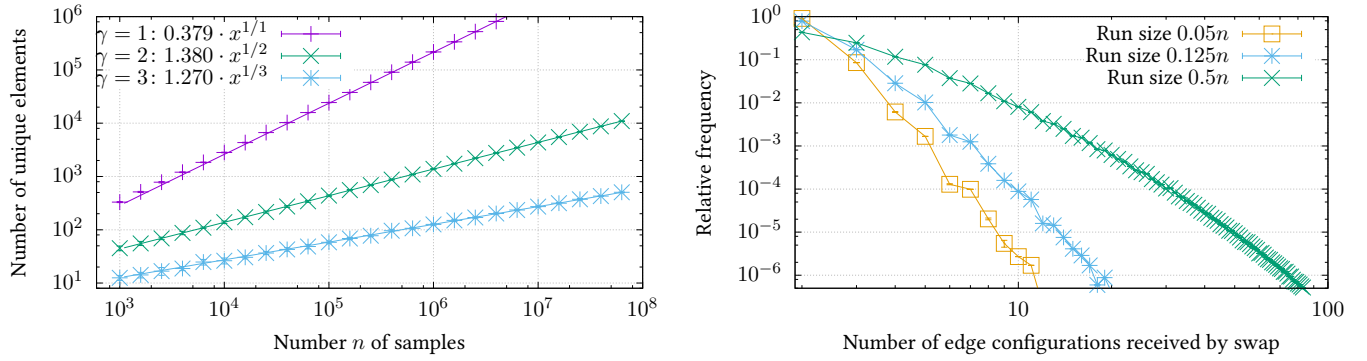


Figure 4.8: **Left:** Number of distinct elements in n samples (i.e. node degrees in a degree sequence) taken from PLD $([1, n], \gamma)$; cf. Section 4.10.2. **Right:** Overhead induced by tracing inter-swap dependencies. Fraction of swaps as function of the number of edge configurations they receive during the simulation phase (cf. Section 4.10.3).

authors show that the degree distribution of the U.S. users (removing connections to non-U.S. users) is very similar to the one of the Facebook users of the whole world, supporting our use of just one parameter set for different graph sizes.

The minimum degree of the Facebook network is 1, but such small degrees are significantly less prevalent than a powerlaw degree sequence would suggest, which is why we chose a value of 50. Our maximum degree of 10,000 is larger than the one reported for Facebook (5000 which is an arbitrarily enforced limit by Facebook). The expected average degree of this degree sequence is 264, which is slightly higher than the reported 190 (world) or 214 (U.S. only). Our parameters are chosen such that the median degree is approximately 99 matching the worldwide Facebook network. Similar to the first parameter set, we chose the maximum community size slightly larger than the largest degree.

4.10.2 EM-HH's State Size

In Lemma 4.2, we bound EM-HH's internal memory consumption by showing that a sequence of n numbers randomly sampled from PLD $([1, n], \gamma)$ contains only $\mathcal{O}(n^{1/\gamma})$ distinct values with high probability.

In order to support Lemma 4.2 and to estimate the hidden constants, samples of varying size between 10^3 and 10^8 are taken from distributions with exponents $\gamma \in \{1, 2, 3\}$. Each time, the number of unique elements is computed and averaged over $S = 9$ runs with identical configurations but different random seeds. The results illustrated in Figure 4.8 support the predictions with small constants and negligible deviations. For the commonly used exponent 2, we find $1.38\sqrt{n}$ distinct elements in a sequence of length n .

4.10.3 Inter-Swap Dependencies

Whenever multiple swaps target the same edge, EM-ES simulates all possible states to be able to retrieve conflicting edges. In Section 4.5.9, we argue that the number of

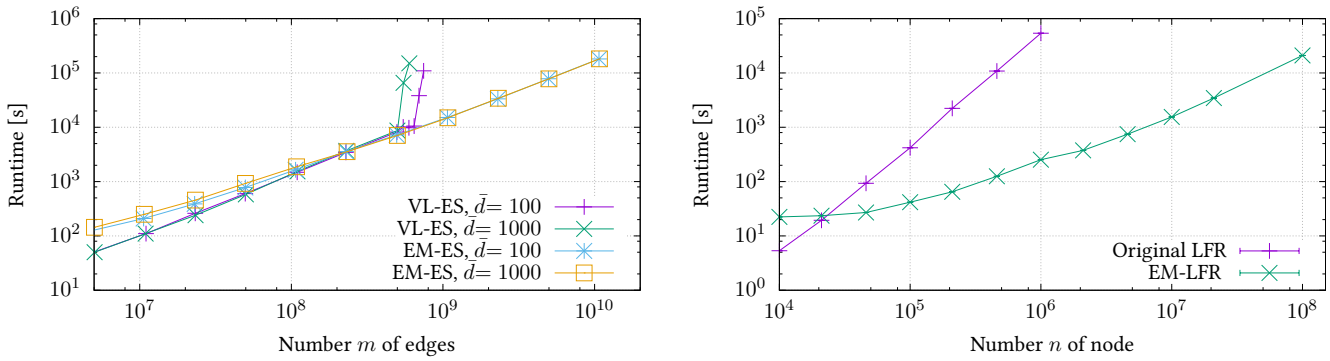


Figure 4.9: Left: Runtime on SysB of VL-ES and EM-ES on graph with m edges and avg. deg. \bar{d} executing $k=10m$ swaps (cf. Section 4.10.6). **Right:** Runtime on SysA of the original *LFR* implementation and EM-LFR for $\mu=0.2$ (cf. Section 4.10.9).

dependencies and the state size remains manageable if the sequence of swaps is split into sufficiently short runs. We found that for m edges and k swaps, $8k/m$ runs minimize the runtime for large instances of **lin**. As indicated in Figure 4.8, in this setting 78.7% of swaps receive the two requested edge configurations with no additional overhead during the simulation phase. Less than 0.4% consider more than four additional states (i.e. more than six messages in total). Similarly, 78.6% of existence requests remain without dependencies.

4.10.4 Test Systems

Runtime measurements were conducted on the following systems:

inexpensive server **SysA** Intel E5-2630 v3 (8 core, 2.4GHz), 64 GB RAM, 3× Samsung 850 PRO SATA SSD

commodity hardware **SysB** Intel Core i7 970 (6 core, 3.2GHz), 12 GB RAM, 1× Samsung 850 PRO SATA SSD

Since edge switching scales linearly in the number of swaps (in case of EM-ES in the number of runs), some of the measurements beyond 3 h runtime are extrapolated from the progress until then. We verified that errors stay within the indicated margin using reference measurements without extrapolation.

4.10.5 Performance of EM-HH

EM-HH:
 Section 4.4

Our implementation of EM-HH produces 180(5) million edges per second on SysA up to at least $2 \cdot 10^{10}$ edges. Here, we include the computation of the input degree sequence, EM-HH’s compaction step, as well as the writing of the output to external memory.

4.10.6 Performance of EM-ES

Figure 4.9 presents the runtime required on SysB to process $k = 10m$ swaps in an input graph with m edges and for the average degrees $\bar{d} \in \{100, 1000\}$. For reference, we include the performance of the existing internal memory edge swap algorithm VL-ES

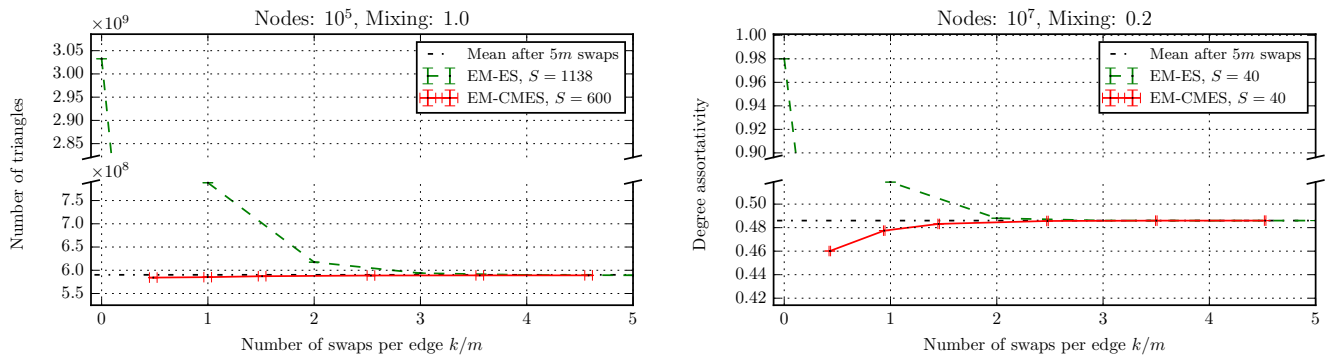


Figure 4.10: **Left:** Number of triangles on **const** with $n = 1 \cdot 10^5$ and $\mu = 1.0$. **Right:** Degree assortativity on **const** with $n = 1 \cdot 10^7$ and $\mu = 0.2$. In order to factor in the increased runtime of EM-CM/ES compared to EM-HH, plots of EM-CM/ES are shifted by the runtime of this phase relative to the execution of EM-ES. As EM-CM/ES is a Las-Vegas algorithm, this incurs an additional error along the x-axis.

based on the authors' implementation [332].¹⁰ VL-ES slows down by a factor of 25 if the data structure exceeds the available internal memory by less than 10%. We observe an analogous behavior on machines with larger RAM. EM-ES is faster than VL-ES for all instances with $m > 2.5 \cdot 10^8$ edges; those graphs still fit into main memory.

FDSM has applications beyond synthetic graphs, and is for instance used on real data to assess the statistical significance of observations [296]. In that spirit, we execute EM-ES on an undirected version of the crawled ClueWeb12 graph's core [324] which we obtain by deleting all nodes corresponding to uncrawled URLs.¹¹ Performing $k = m$ swaps on this graph with $n \approx 9.8 \cdot 10^8$ nodes and $m \approx 3.7 \cdot 10^{10}$ edges is feasible in less than 19.1 h on SysB.

Bhuiyan et al. propose a distributed edge switching algorithm and evaluate it on a compute cluster with 64 nodes each equipped with two Intel Xeon E5-2670 2.60GHz 8-core processors and 64GB RAM [43]. The authors report to perform $k = 1.15 \cdot 10^{11}$ swaps on a graph with $m = 10^{10}$ generated in a preferential attachment process in less than 3 h. We generate a preferential attachment graph using an EM generator [239] matching the aforementioned properties and carried out edge swaps using EM-ES on SysA. We observe a slowdown of only 8.3 on a machine with 1/128 the number of comparable cores and 1/64 of internal memory.

4.10.7 EM-CM/ES's Performance and Mixing Comparison with EM-ES

In Section 4.6, we describe an alternative graph sampling method. Instead of seeding EM-ES with a highly biased graph using EM-HH, we employ the Configuration Model to generate a non-simple random graph and then obtain a simple graph using several

¹⁰Here we report only on the edge swapping process excluding any precomputation. To achieve comparability, we removed connectivity tests, fixed memory management issues, and adopted the number of swaps. Further, we extended counters for edge ids and accumulated degrees to 64 bit integers in order to support experiments with more than 2^{30} edges.

¹¹We consider such vertices atypically simple as they have degree 1 and account for $\approx 84\%$ of nodes.

EM-ES:

[Section 4.5](#)

MP-BA:

[Chapter 3](#)

EM-ES runs in a Las-Vegas fashion.

Since EM-ES scans through the edge list in each iteration, runs with very few swaps are inefficient. For this reason, we start the subsequent Markov chain to further randomize the graph early: First identify all multi-edges and self-loops and generate swaps with random partners. In a second step, we then introduce additional random swaps until the run contains at least $m/10$ operations.¹²

EM-CM/ES:
 Section 4.6

For an experimental comparison between EM-ES and EM-CM/ES, we consider the runtime until both yield a sufficiently uniform random sample. Of course, the uniformity is hard to quantify; similarly to related studies (cf. Section 4.1.1), we estimate the mixing times of both approaches as follows.

Starting from a common seed graph $G^{(0)}$, we generate an ensemble $\{G_1^{(k)}, \dots, G_S^{(k)}\}$ of $S \gg 1$ instances by applying independent random sequences of $k \gg m$ swaps each. During this process, we regularly export snapshots $G_i^{(jm)}$ of the intermediate instances $j \in [k/m]$ of graph G_i . For EM-CM/ES, we start from the same seed graph, apply the algorithm and then carry out k swaps as described above.

For each snapshot, we compute several metrics, such as the average local clustering coefficient (ACC), the number of triangles, and degree assortativity.¹³ We then investigate how the distribution of these measures evolves within the ensemble as we carry out an increasing number of swaps. We omit results for ACC since they are less sensitive compared to the other measures (see Section 4.10.8).

As illustrated in Figure 4.10 and Section 4.C (Appendix), all proxy measures converge within $5m$ swaps with a very small variance. No statistically significant change can be observed compared to a Markov chain with $30m$ operations (which was only computed for a subset of each ensemble due to its computational cost). EM-HH generates biased instances with special properties, such as a high number of triangles and correlated node degrees, while the features of EM-CM/ES's output nearly match the converged ensemble. This suggests that the number of swaps to obtain a sufficiently uniform sample can be reduced for EM-CM/ES.

Due to computational costs, the study was carried out on multiple machines executing several tasks in parallel. Hence, absolute running times are not meaningful, and we rather measure the computational costs in units of time required to carry out $1m$ swaps by the same process. This accounts for the offset of EM-CM/ES's first data point.

The number of rounds required to obtain a simple graph depends on the degree distribution. For **const** with $n = 1 \cdot 10^5$ and $\mu = 1$, a fraction of 5.1% of the edges produced by the Configuration Model are illegal. EM-ES requires 18(2) rewiring runs in case a single swap is used per round to rewire an illegal edge. In the default mode of operation, 5.0 rounds suffice as the number of rewiring swaps per illegal edge is doubled in each round. For larger graphs with $n = 1 \cdot 10^7$, only 0.07% of edges are illegal and need 2.25(40) rewiring runs.

¹²Chosen to yield execution times similar to the $m/8$ -setting of EM-ES on simple graphs.

¹³In preliminary experiments, we also included spectral properties (such as extremal eigenvalues of the adjacency/Laplacian matrix) and the closeness centrality of fixed nodes. As these are more expensive to compute and yield qualitatively similar results, we decided not to include them in the larger trials.

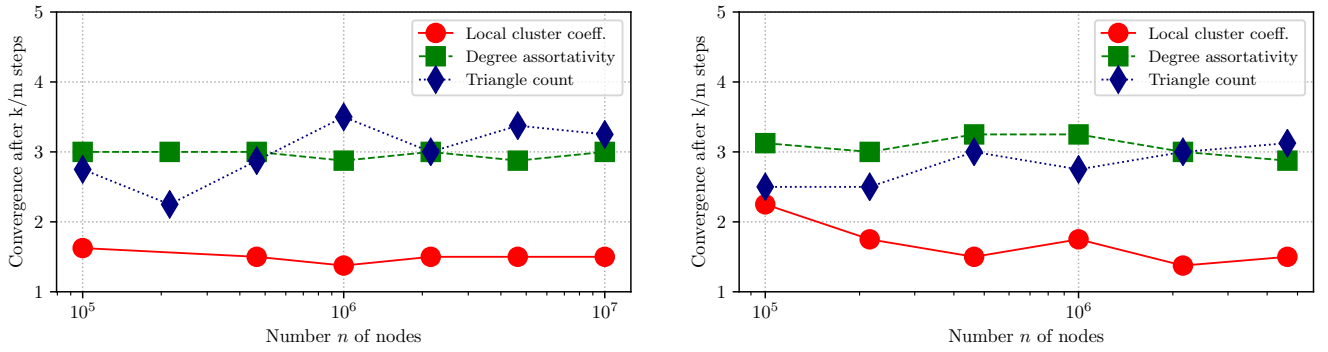


Figure 4.11: Number of swaps per edge after which ensembles of graphs with the following parameters converge: **const**, $1 \cdot 10^5 \leq n \leq 1 \cdot 10^7$ and $\mu = 0.4$ (left) and $\mu = 0.6$ (right). Due to computational costs, the ensemble size is reduced from $S > 100$ to $S > 10$ for large graphs.

4.10.8 Convergence of EM-ES

In a similar spirit to the previous section, we indirectly investigate the Markov chain’s mixing time as a function of the number of nodes n . To do so, we generate ensembles as before with $1 \cdot 10^5 \leq n \leq 1 \cdot 10^7$ and compute the same graph metrics. For each group and measure, we then search for the first snapshot p in which the measure’s mean is within an interval of half the standard deviation of the final values and subsequently remains there for at least three phases. We then interpret p as a proxy for the mixing time. As depicted in Figure 4.11, no measure shows a systematic increase over the two orders of magnitude considered. It hence seems plausible not to increase the number of swaps performed by EM-LFR compared to the original implementation.

4.10.9 Performance of EM-LFR

Figure 4.9 reports the runtime of the original *LFR* implementation and EM-LFR as a function of the number of nodes n and $\nu = 1$. EM-LFR is faster for graphs with $n \geq 2.5 \cdot 10^4$ nodes which feature approximately $5 \cdot 10^5$ edges and are well in the IM domain. Further, the implementation is capable of producing graphs with more than $1 \cdot 10^{10}$ edges in 17 h.¹⁴ Using the same time budget, the original implementation generates graphs more than two orders of magnitude smaller.

4.10.10 Qualitative Comparison of EM-LFR

When designing EM-LFR, we closely followed the *LFR* benchmark such that we can expect it to produce graphs following the same distribution as the original *LFR* benchmark. To confirm this experimentally, we generated graphs with identical parameters using the original *LFR* implementation and EM-LFR. For disjoint clusters we also compare it with the implementation of *NetworKit* [316].

¹⁴Roughly 1.5 h are spent in the end-game of the Global Rewiring (at that point less than one edge out of 10^6 is invalid). In this situation, an algorithm using random I/Os may yield a speedup. Alternatively, we could simply discard the insignificant fraction of remaining invalid edges.

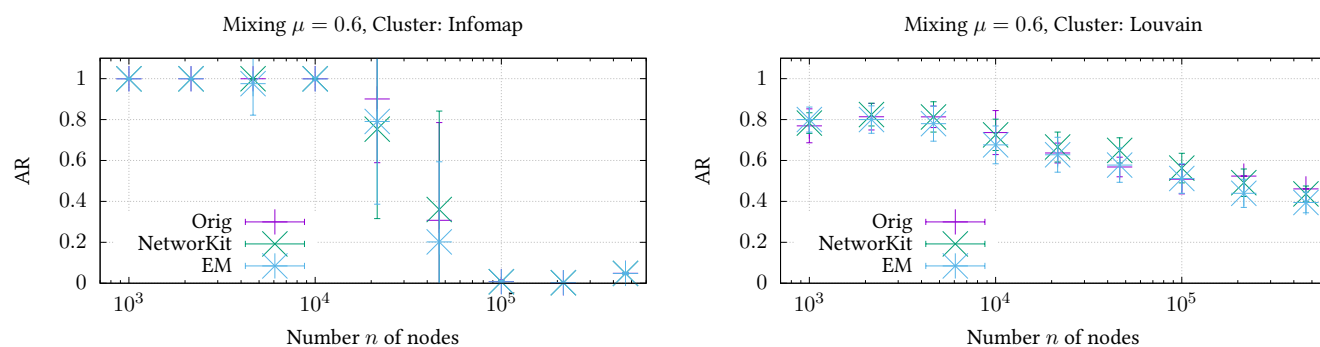


Figure 4.12: Adjusted rand of INFOMAP or LOUVAIN and ground truth at $\mu=0.6$ with disjoint clusters, $s_{\min}=10$, $s_{\max}=n/20$.

INFOMAP and LOUVAIN

coverage

For disjoint clusters, we evaluate the results of the INFOMAP [287] and the LOUVAIN [53] algorithm. The LOUVAIN algorithm optimizes the famous modularity measure [261] while INFOMAP optimizes the map equation [287]. Both are formalizations of the intuitive principle that clusters should be internally dense but externally sparse. Modularity is directly based on this principle. Its value is based on the fraction of edges inside clusters, the so-called *coverage*. However, just optimizing coverage would mean that a single cluster with all nodes is optimal. As a remedy, the expected coverage of the clustering in a graph with the same nodes and degrees, but edges distributed randomly according to the *Configuration Model*, is subtracted from the actual coverage. The map equation, on the other hand, optimizes the expected length of the description of a random walk. In the non-hierarchical version we employ here, this expected length is calculated for a two-level code with global code words for clusters and then local code words for the nodes inside every cluster. The basic idea is that in a good clustering, random walks tend to stay within a cluster and thus such a clustering leads to shorter code words in expectation.

The INFOMAP and the LOUVAIN algorithm are quite similar in their basic structure. They start with a clustering where every node is in its own cluster. Then they apply two principles alternately: local moving and contraction. The idea of local moving is to move a node into a cluster of one of its adjacent nodes if this improves the clustering quality. This is repeatedly applied to all nodes in a random order until no improvement is possible anymore. In the contraction phase, the nodes of each cluster are contracted into a single node while combining duplicate edges. The INFOMAP algorithm extends this basic scheme by introducing additional local moving phases on parts of the graph where clusters can be split again to improve the quality.

Higher modularity and lower map equation values indicate better clusterings. However, sometimes higher modularity values can also be achieved by merging small but actually clearly distinct clusters. This effect is called resolution limit [132]. The map equation has a resolution limit, too, but in practice it is orders of magnitudes smaller [190]. The LOUVAIN algorithm as well as INFOMAP were found to achieve high-quality results on *LFR* benchmark graphs while being fast [209]. In particular the LOUVAIN method is also among the most frequently used community detection algorithms [133, 120].

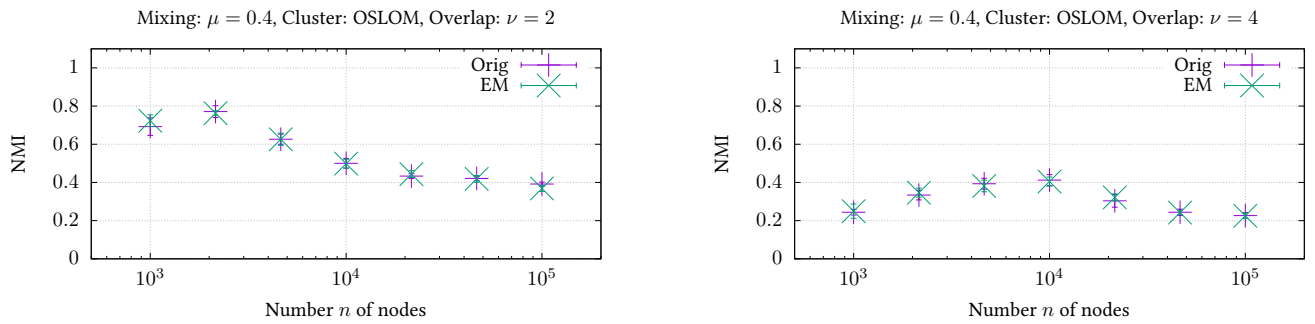


Figure 4.13: NMI of OSLOM and ground truth at $\mu = 0.4$ with 2/4 overlapping clusters per node.

For overlapping clusters, we evaluate the results of OSLOM [211]. OSLOM aims to find clusters that are statistically significant. Given a cluster C and a node u , it analyzes whether u has statistically significantly many connections to nodes in C relative to a Configuration Model graph. For a single cluster, OSLOM considers both adding and removing nodes based on this criteria.

OSLOM

To cluster a whole graph, clusters are expanded starting from single nodes and then evaluated by testing if repeatedly adding and removing nodes leads to an empty cluster. Repeatedly encountered clusters are considered significant. The algorithm stops when it starts detecting similar clusters over and over again. OSLOM is one of the best-performing algorithms for overlapping community detection [77, 133]. We compare the clusterings of the algorithms to *LFR*'s ground truth using the adjusted rand measure [180] for disjoint clusters and NMI [124] for both disjoint and overlapping clusters.

Adjusted Rand Measure and NMI

Further, we examine the average local clustering coefficient. As it measures the fraction of closed triangles, it shows the presence of locally denser areas as expected in communities [189]. We report these measures for graphs ranging from 10^3 to 10^6 nodes and present a selection of results in figures 4.12 to 4.14; all of them can be found in Section 4.B (Appendix). There are only small differences within the range of random noise between the graphs generated by EM-LFR and the other two implementations. Note that due to the computational costs above 10^5 edges, there is only one sample for the original implementation causing the outliers in Figure 4.12.

Similar to the results in [120], we also observe that the performance of clustering algorithms drops significantly as the graph's size grows. For LOUVAIN, this is partially due to the resolution limit that prevents the detection of small communities in huge graphs. Due to the different powerlaw exponents, the average community size grows much faster than the average degree as the size of the graphs is increased. Therefore, in particular the larger clusters become sparser and thus more difficult to detect with increasing graph size. On the other hand, small clusters become easier to detect as the graph size grows because outgoing edges are distributed among more nodes and are thus easier to distinguish from intra-cluster edges. This might explain why the performance of OSLOM first improves as the graph size grows. Apart from that, currently used heuristics might also just be unsuited for large graphs with nodes of very different

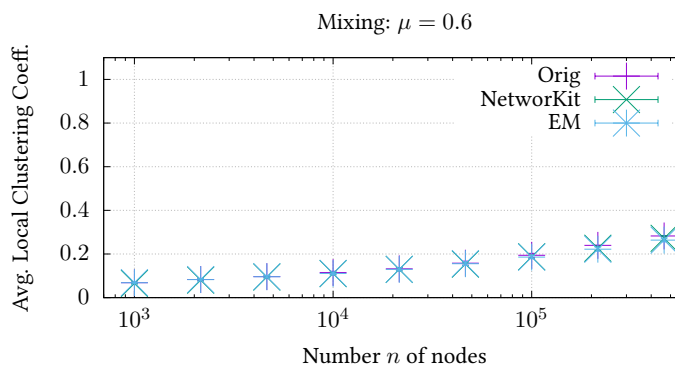


Figure 4.14: Average local clustering coefficient with mixing of $\mu = 0.6$ and disjoint clusters.

degrees. Results on *LFR* graphs with one million nodes in [169] show that both *LOUVAIN* and *INFOMAP* are unable to detect the ground truth on *LFR* graphs with higher values of μ even though the ground truth has a better modularity or map equation score than the found clustering. Such behavior clearly demonstrates the necessity of *EM-LFR* for being able to study this phenomenon on even larger graphs and develop algorithms that are able to handle such instances.

The quality of the community assignments used by *LFR* and *EM-LFR* is assessed in terms of the modularity $\mathcal{Q}_G(C)$ scores [261] achieved by the generated graph G and ground truth C . In general $\mathcal{Q}_G(C)$ takes values in $[-1, 1]$, but for large n and bounded community sizes, the modularity of a *LFR* graph approaches $\mathcal{Q} \rightarrow 1 - \mu$ as the coverage corresponds to $1 - \mu$ while the expected coverage approaches 0. For each configuration $n \in \{10^3, \dots, 10^6\}$ and $\mu \in \{0.2, 0.4, 0.6\}$, we generate $S \geq 10$ networks for each generator and compute their mean modularity score. In all cases, the relative differences between the two generators is below 10^{-2} and for small μ typically another order of magnitude smaller.

4.11 Outlook and Conclusion

We propose the first I/O-efficient graph generator for the *LFR* benchmark and the *FDSM*, which is the most challenging step involved that dominates the running time: *EM-HH* materializes a graph based on a prescribed degree distribution without I/O for virtually all realistic parameters. Including the generation of a powerlaw degree sequence and the writing of the output to disk, our implementation generates $1.8 \cdot 10^8$ edges per second for graphs exceeding main memory. *EM-ES* randomizes graphs with m edges based on k edge switches using $\mathcal{O}(k/m \cdot \text{sort}(m))$ I/Os for $k = \Omega(m)$.

We demonstrate that *EM-ES* is faster than the internal memory implementation [332] even for large instances still fitting in main memory and scales well beyond the limited main memory. Compared to the distributed approach by [43] on a cluster with 128 CPUs, *EM-ES* exhibits a slowdown of only 8.3 on one CPU and hence poses a viable and cost-efficient alternative. Our *EM-LFR* implementation is orders of magnitude faster than the

original *LFR* implementation for large instances and scales well to graphs exceeding main memory while the generated graphs are equivalent. Graphs with more than $1 \cdot 10^{10}$ edges can be generated in 17 h. We further give evidence that commonly accepted parameters to derive the length of the edge switching Markov chain remain valid for graph sizes approaching the external memory domain and that EM-CM/ES can be used to accelerate the process.

This provides the basis for the development and evaluation of clustering algorithms for graphs that exceed main memory. The necessity for such an evaluation has already been demonstrated by first results in [169] that show that the behavior of algorithms on large graphs is not necessarily the same as on small graphs even when cluster sizes do not change. Comparison measures such as NMI or the adjusted rand index typically do not consider the graph structure, therefore they can usually still be computed in internal memory even for graphs that exceed main memory. However, for graphs where even the number of nodes exceeds the size of the internal memory, there is the need to develop memory-efficient algorithms also for comparing clusterings.

Acknowledgment

We thank Hannes Seiwert and Mark Ortmann for valuable discussions on EM-HH.

Appendix 4.A Summary of Definitions

Table 4.1: Definitions used in this paper.

Symbol	Description
$[k]$	$[k] := \{1, \dots, k\}$ for $k \in \mathbb{N}_+$ (Sec. 4.2)
$[u, v]$	Undirected edge with implication $u \leq v$ (Sec. 4.2)
$\langle X \rangle$	The mean $\langle X \rangle := \sum_{i=1}^n x_i/n$
$\langle X^2 \rangle$	The second moment $\langle X^2 \rangle := \sum_{i=1}^n x_i^2/n$
B	Number of items in a block transferred between IM and EM (Sec. 4.2.2)
d_{\min}, d_{\max}	Min/max degree of nodes in LFR benchmark (Sec. 4.3)
d_v^{in}	$d_v^{\text{in}} = (1-\mu) \cdot d_v$, intra-community degree of node v (Sec. 4.3)
\mathcal{D}	$\mathcal{D} = (d_1, \dots, d_n)$ with $d_i \leq d_{i+1} \forall i$. Degree sequence of a graph (Sec. 4.4)
$D(\mathcal{D})$	$D(\mathcal{D}) = \{d_i : 1 \leq i \leq n\} $ where $\mathcal{D} = (d_1, \dots, d_n)$ (Sec. 4.4)
n	Number of vertices in a graph (Sec. 4.2)
m	Number of edges in a graph (Sec. 4.2)
μ	Mixing parameter in LFR benchmark, i.e. ratio of neighbors that shall be in other communities (Sec. 4.3)
M	Number of items fitting into internal memory (Sec. 4.2.2)
PLD $([a, b], \gamma)$	Powerlaw distribution with exponent $-\gamma$ on the interval $[a, b]$ (Sec. 4.2)
s_{\min}, s_{\max}	Min/max size of communities in LFR benchmark (Sec. 4.3)
scan(n)	scan(n) = $\Theta(n/B)$ I/Os, scan complexity (Sec. 4.2.2)
sort(n)	sort(n) = $\Theta((n/B) \cdot \log_{M/B}(n/B))$ I/Os, sort complexity (Sec. 4.2.2)

Table 4.2: Parameters of overlapping LFR. The typical values are based on the suggestions by [208].

Parameter	Meaning
n	Number of nodes to be produced
PLD $([d_{\min}, d_{\max}], \gamma)$	Degree distribution of nodes, typically $\gamma = 2$
$0 \leq O \leq n, \nu \geq 1$	O random nodes belong to ν communities; remainder has one membership
PLD $([s_{\min}, s_{\max}], \beta)$	Size distribution of communities, typically $\beta=1$
$0 < \mu < 1$	Mixing parameter: fraction of neighbors of every node u that shall not share a community with u

Appendix 4.B Comparing LFR Implementations

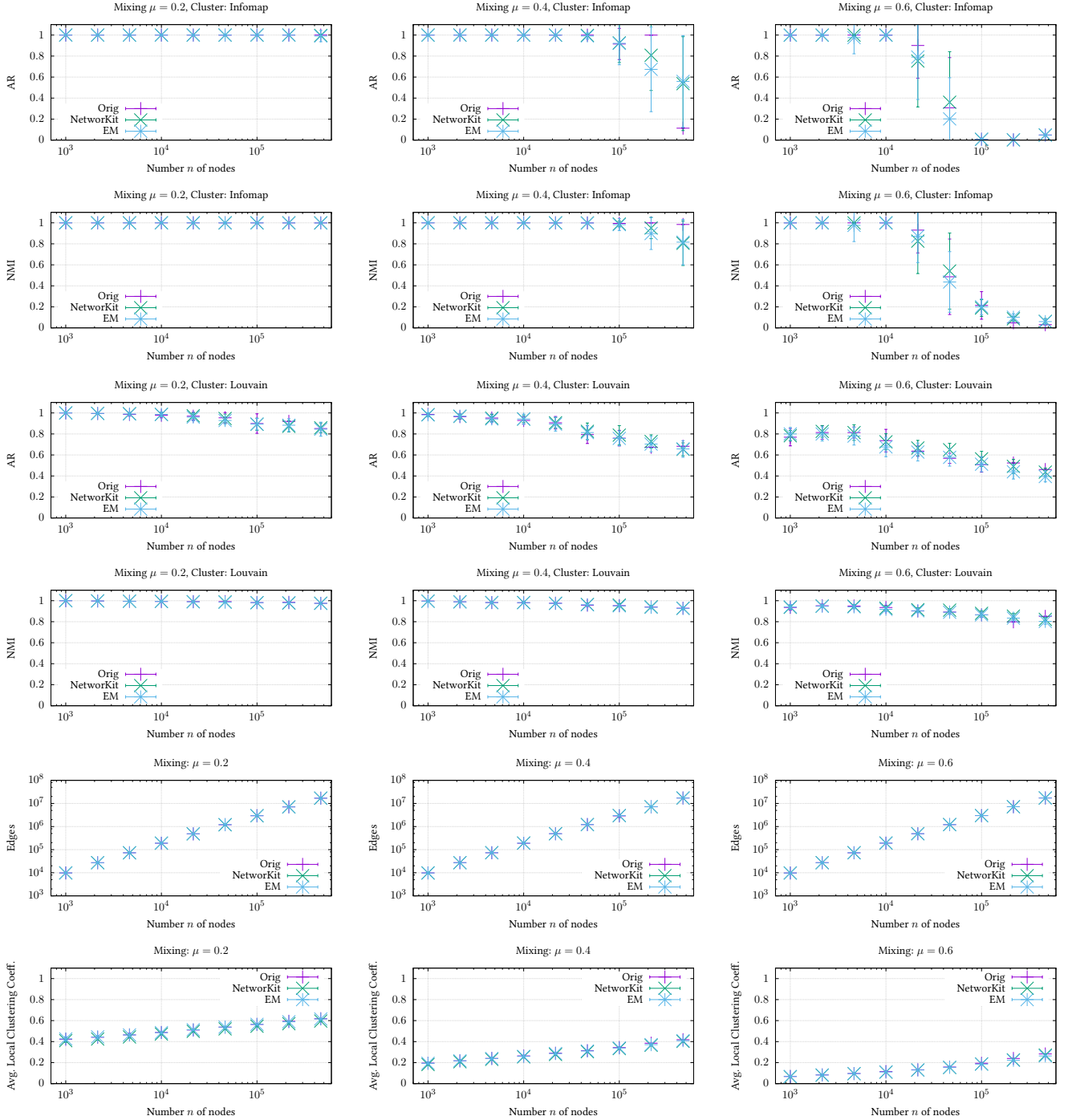
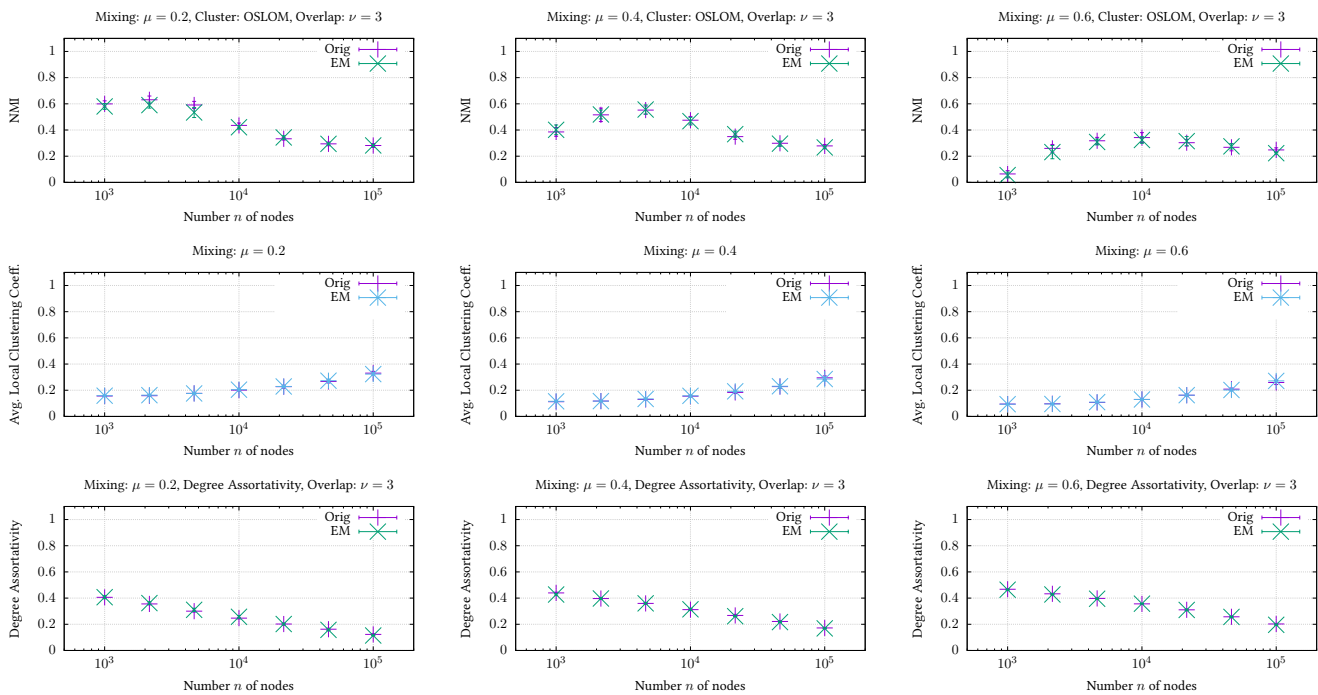
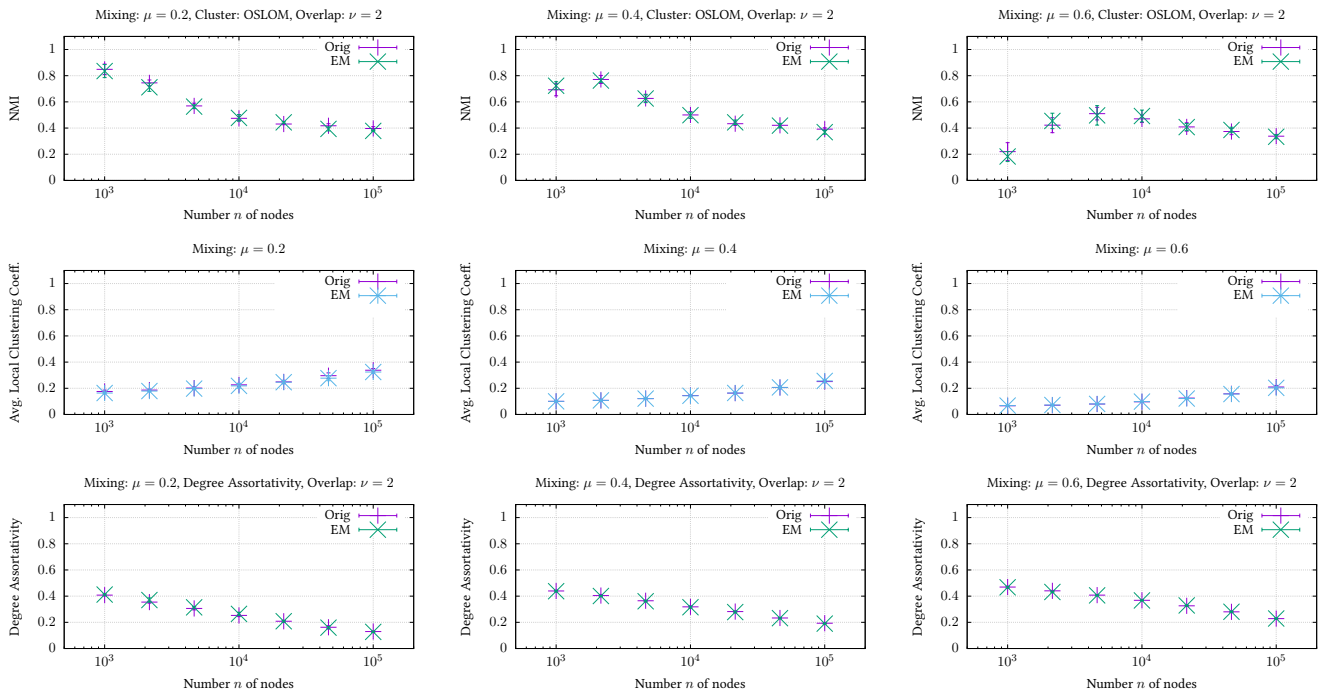


Figure 4.15: Comparison of the original *LFR* implementation, the *NetworkKit* implementation and our EM solution for values of $10^3 \leq n \leq 10^6$, $\mu \in \{0.2, 0.4, 0.6\}$, $\gamma=2$, $\beta=1$, $d_{\min}=10$, $d_{\max}=n/20$, $s_{\min}=10$, $s_{\max}=n/20$. Clustering is performed using INFOMAP and LOUVAIN and compared to the ground truth emitted by the generator using AdjustedRandMeasure (AR) and Normalized Mutual Information (NMI); $S \geq 8$. Due to the computational costs, graphs with $n \geq 10^5$ have a reduced multiplicity. In case of the original implementation it may be based on a single run which accounts for the few outliers.

Massive Graphs Following the LFR Benchmark



Comparing LFR Implementations

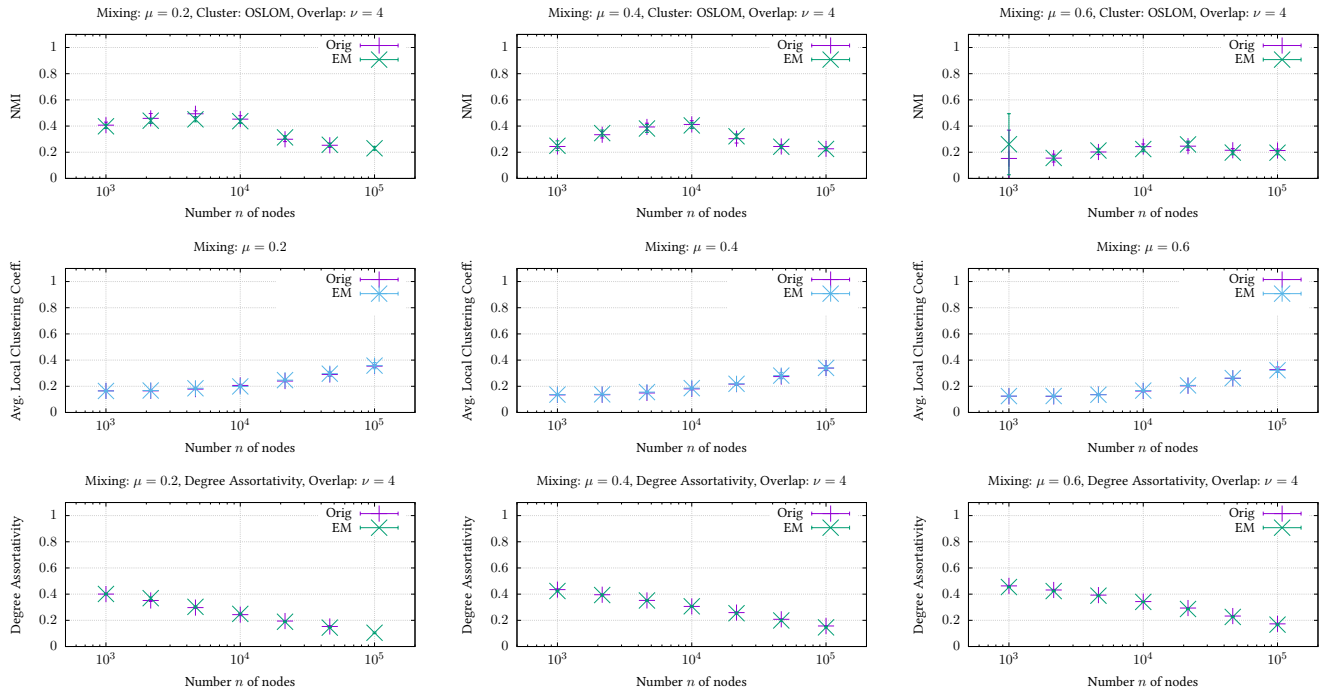


Figure 4.16: Comparison of the original *LFR* implementation and our EM solution for values values of $10^3 \leq n \leq 10^6$, $\mu \in \{0.2, 0.4, 0.6\}$, $\nu \in \{2, 3, 4\}$, $O = n$, $\gamma = 2$, $\beta = 1$, $d_{\min} = 10$, $d_{\max} = n/20$, $s_{\min} = 10\nu$, $s_{\max} = \nu \cdot n/20$. Clustering is performed using OSLOM and compared to the ground truth emitted by the generator using a generalized Normalized Mutual Information (NMI); $S \geq 5$.

Appendix 4.C Comparing EM-ES and EM-CM/ES

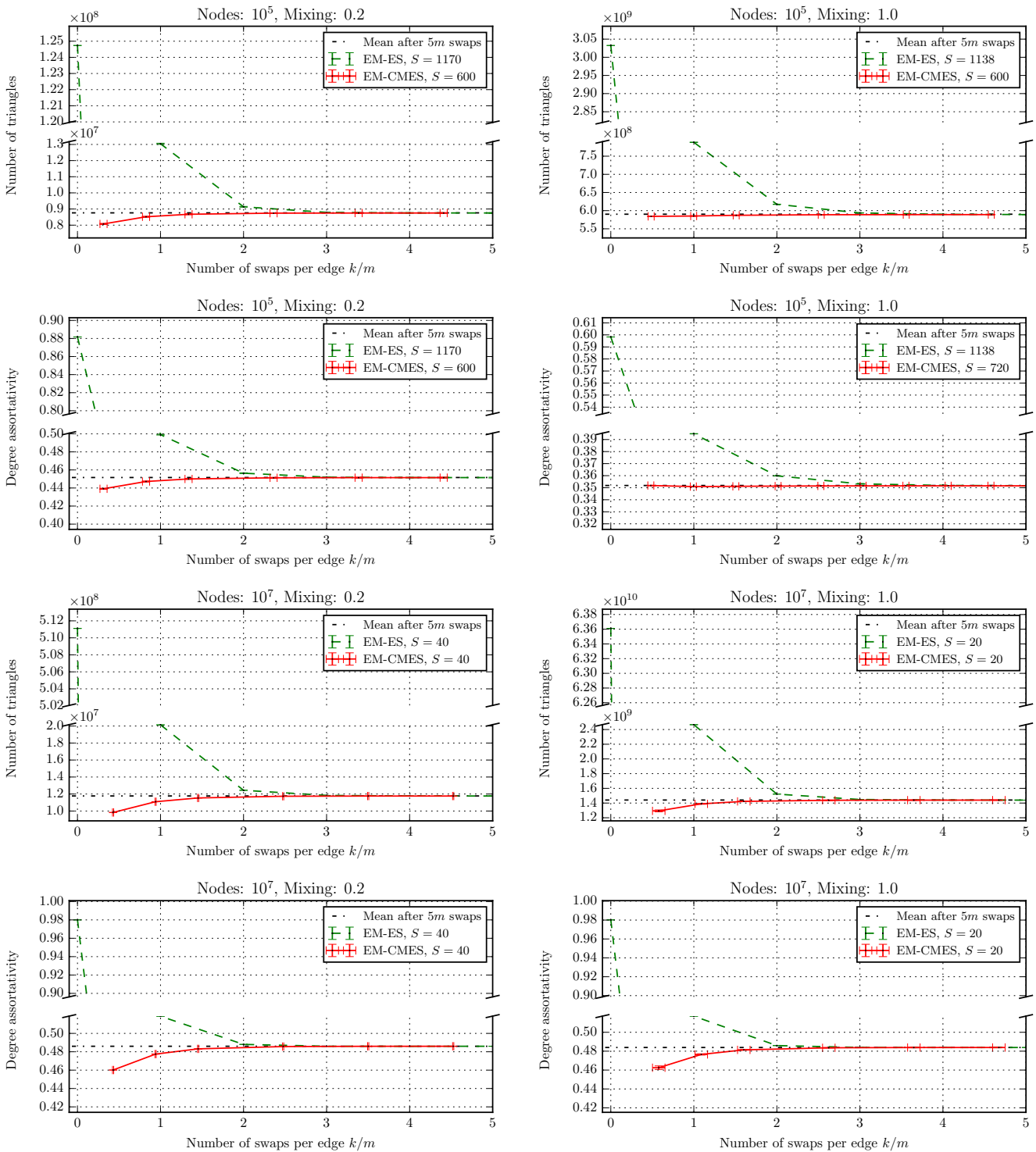


Figure 4.17: Triangle count and degree assortativity of a graph ensemble obtained by applying random swaps/the Configuration Model to a common seed graph. Refer to section 4.10.7 for experimental details.

Parallel and I/O-efficient Randomization of Massive Networks using Global Curveball Trades

5

joint work with C.J. Carstens, M. Hamann, U. Meyer, H. Tran, and D. Wagner

Graph randomization is a crucial task in the analysis and synthesis of networks. It is typically implemented as an *edge switching* process (*ES*) repeatedly swapping the nodes of random edge pairs while maintaining the degrees involved [151].

Curveball is a novel approach that instead considers the whole neighborhoods of randomly drawn node pairs. Its Markov chain converges to a uniform distribution, and experiments suggest that it requires less steps than the established *ES* [81]. Since trades however are more expensive, we study Curveball's practical runtime by introducing the first efficient Curveball algorithms: the I/O-efficient *EM-CB* for simple undirected graphs and its internal memory pendant *IM-CB*.

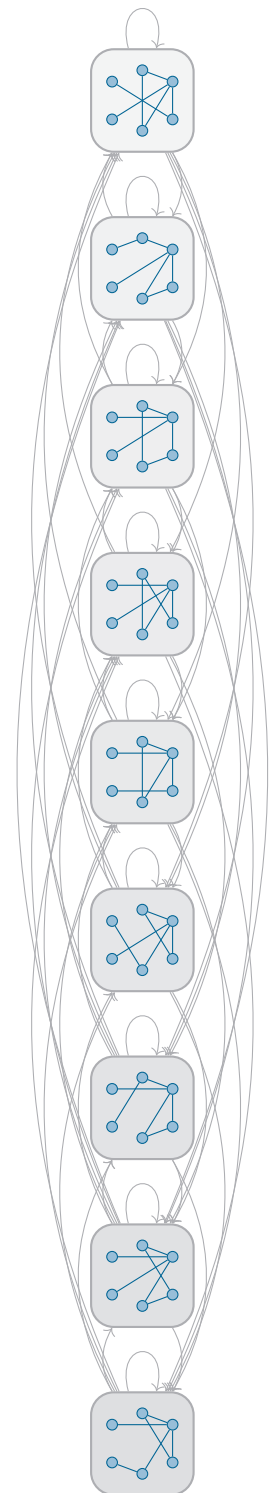
Further, we investigate *global trades* [81] processing every node in a graph during a single super step, and show that undirected global trades converge to a uniform distribution and perform superior in practice. We then discuss *EM-GCB* and *EM-PGCB* for global trades and give experimental evidence that *EM-PGCB* achieves the quality of the state-of-the-art *ES* algorithm *EM-ES* [168] nearly one order of magnitude faster.

This chapter is based on the peer-reviewed conference article [82]:

- [82] C. J. Carstens, M. Hamann, U. Meyer, M. Penschuck, H. Tran, and D. Wagner. Parallel and I/O-efficient randomisation of massive networks using Global Curveball trades. In Y. Azar, H. Bast, and G. Herman, editors, *European Symp. on Algorithms ESA*, volume 112 of *LIPICs*, pages 11:1–11:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.ESA.2018.11 .

My contribution

Hung Tran and I are main authors of this paper. Together, we contributed most of the algorithms and their implementations.



Markov chain
of Curveball for
(1, 2, 4, 2, 2, 1)

5.1 Introduction

In the analysis of complex networks, such as social networks, the underlying graphs are commonly compared to random graph models to understand their structure [186, 262, 320]. While simple models like Erdős-Rényi graphs [121] are easy to generate and analyze, they are too different from commonly observed powerlaw degree sequences [262, 258, 320]. Thus, random graphs with the same degree sequence as the given graph are frequently used [97, 186, 297]. In practice, many of these graphs are simple graphs, i.e. graphs without self-loops and multiple edges. In order to obtain reliable results in these cases, the graphs sampled need to be simple since non-simple models can lead to significantly different results [296, 297]. The randomization of a given graph is commonly implemented as an *Edge Switching* [97, 246].

Nowadays, massive graphs that cannot be processed in the RAM of a single computer, require new analysis algorithms to handle these huge datasets. In turn, large benchmark graphs are required to evaluate the algorithms' scalability — in terms of speed and quality. *LFR* is a standard benchmark for evaluating clustering algorithms which repeatedly generates highly biased graphs that are then randomized [208, 210]. [168] presents the external memory *LFR* generator EM-LFR and its I/O-efficient edge switching EM-ES. Although EM-ES is faster than previous results even for graphs fitting into RAM, it dominates EM-LFR's running time. Alternative sampling via the Configuration Model [247] was studied to reduce the initial bias and the number of *ES* steps necessary [168]. Still, graph randomization remains a major bottleneck during the generation of these huge graphs.

The Curveball algorithm has been originally proposed for randomizing binary matrices while preserving row and column sums [321, 331] and has been adopted for graphs [80, 81]: instead of switching a pair of edges as in *ES*, Curveball trades the neighbors of two nodes in each step. Carstens et al. further propose the concept of a *global trade*, a super step composed of single trades targeting every node¹ in a graph once [81]. The authors show that global trades in bipartite or directed graphs converge to a uniform distribution, and give experimental evidence that global trades require fewer Markov-chain steps than single trades. However, while fewer steps are needed, the trades themselves are computationally more expensive. Since we are not aware of previous efficient Curveball algorithms and implementations, we investigate this trade-off here.

5.1.1 Our Contributions

We present the first efficient algorithms for Curveball: the (sequential) internal memory and external memory algorithms IM-CB² and EM-CB for the Simple Undirected Curveball algorithm (see Section 5.4). Experiments in Section 5.5, indicate that they are faster than the established edge switching approaches in practice.

¹For an odd number n of nodes, a single node is left out

²We prefix internal memory algorithms with IM and I/O-efficient algorithms with EM. The suffices CB, GCB, and PGCB denote Curveball, CB. with global trades, and parallel CB. with global trades respectively.

In Section 5.3, we show that random global trades lead to uniform samples of simple, undirected graphs and demonstrate experimentally in Section 5.5 that they converge even faster than the corresponding number of uniform single trades. Exploiting structural properties of global trades, we simplify EM-CB yielding EM-GCB and the parallel I/O-efficient EM-PGCB which achieves EM-ES's quality nearly one order of magnitude faster in practice (see Section 5.5).

5.2 Preliminaries and Notation

We define the short-hand $[k] := \{1, \dots, k\}$ for $k \in \mathbb{N}_{>0}$, and write $[x_i]_{i=a}^b$ for an ordered sequence $[x_a, x_{a+1}, \dots, x_b]$.

Graphs and Degree Sequences

A graph $G = (V, E)$ has $n = |V|$ sequentially numbered nodes $V = \{v_1, \dots, v_n\}$ and $m = |E|$ edges. Unless stated differently, graphs are assumed to be undirected and unweighted. To obtain a unique representation of an *undirected* edge $\{u, v\} \in E$, we use *ordered* edges $[u, v] \in E$ implying $u \leq v$; in contrast to a directed edge, the ordering is used algorithmically but does not carry any meaning. A graph is called *simple* if it contains neither multi-edges nor self-loops, i.e. $E \subseteq \{\{u, v\} \mid u, v \in V \text{ with } u \neq v\}$. For node $u \in V$ define the *neighborhood* $\mathcal{A}_u := \{v : \{u, v\} \in E\}$ and *degree* $\deg(u) := |\mathcal{A}_u|$. Let $d_{\max} := \max_v \{\deg(v)\}$ be the maximal degree of a graph. A vector $\mathcal{D} = [d_i]_{i=1}^n$ is a degree sequence of graph G iff $\forall v_i \in V : \deg(v_i) = d_i$.

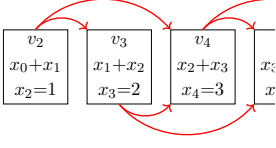
Randomization and Distributions

PLD $([a, b], \gamma)$ refers to an integer Powerlaw Distribution with exponent $-\gamma \in \mathbb{R}$ for $\gamma \geq 1$ and values from the interval $[a, b]$; let X be an integer random variable drawn from PLD $([a, b], \gamma)$ then $\mathbb{P}[X=k] \propto k^{-\gamma}$ (proportional to) if $a \leq k < b$ and $\mathbb{P}[X=k] = 0$ otherwise. A statement depending on some number $x > 0$ is said to hold *with high probability* if it is satisfied with probability at least $1 - 1/x^c$ for some constant $c \geq 1$. Let S be a finite set, $x \in S$ and let σ be permutation on S , we define $\text{rank}_\sigma(x)$ as the number of elements positioned in front of x by σ .

5.2.1 External Memory Model

In contrast to classic models of computation, such as the unit-cost random-access machine, modern computers contain deep memory hierarchies ranging from fast registers, over caches and main memory to solid-state drives (SSDs) and hard disks. Algorithms unaware of these properties may face significant performance penalties.

We use the commonly accepted *External Memory Model* by Aggarwal and Vitter [7] to reason about the influence of data locality in memory hierarchies. It features two memory types, namely fast internal memory (IM or RAM) holding up to M data items, and a slow disk of unbounded size. The input and output of an algorithm are stored in external memory (EM while computation is only possible on values in IM. An algorithm's



Algorithm 6: Compute Fibonacci numbers using *Time Forward Processing*

```

1 PQ.PUSH( (KEY = 2, VAL = 0), (KEY = 2, VAL = 1)) // Send base cases  $x_0$  &  $x_1$  to  $v_2$ 
2 for  $i \leftarrow 2, \dots, n$  do
3   SUM  $\leftarrow$  0
4   while PQ.MIN.KEY ==  $i$  do
5     SUM  $\leftarrow$  SUM + PQ.REMOVEMIN().VAL // Receive all messages for  $x_i$ 
6   PRINT( $x_i =$  SUM)
7   PQ.PUSH( (KEY =  $i+1$ , VAL = SUM), (KEY =  $i+2$ , VAL = SUM))
```

performance is measured in the number of I/Os required. Each I/O transfers a block of $B = \Omega(\sqrt{M})$ consecutive items between memory levels. Reading or writing n contiguous items is referred to as *scanning* and requires $\text{scan}(n) := \Theta(n/B)$ I/Os. Sorting n consecutive items triggers $\text{sort}(n) := \Theta((n/B) \cdot \log_{M/B}(n/B))$ I/Os. For all realistic values of n , B and M , $\text{scan}(n) < \text{sort}(n) \ll n$. Sorting complexity constitutes a lower bound for most intuitively non-trivial EM tasks [242]. EM queues use amortized $\mathcal{O}(1/B)$ I/Os per operation and require $\mathcal{O}(B)$ main memory [230]. An external priority queue (PQ) requires $\mathcal{O}(\text{sort}(n))$ I/Os to push and pop n items [23, 22].

5.2.2 TFP: Time Forward Processing

Time Forward Processing (TFP) is a generic technique to manage data dependencies of external memory algorithms [230]. Consider an algorithm computing values x_1, \dots, x_n in which the calculation of x_i requires previously computed values. One typically models these dependencies using a directed acyclic graph $G=(V, E)$. Every node $v_i \in V$ corresponds to the computation of x_i and an edge $(v_i, v_j) \in E$ indicates that the value x_i is necessary to compute x_j . For instance consider the Fibonacci sequence $x_0 = 0$, $x_1 = 1$, $x_i = x_{i-1} + x_{i-2} \forall i \geq 2$ in which each node v_i with $i \geq 2$ depends on exactly its two predecessors (see Algorithm 6). Here, a linear scan for increasing i suffices to solve the dependencies.

In general, an algorithm needs to traverse G according to some topological order \prec_T of nodes V and also has to ensure that each v_j can access values from all v_i with $(v_i, v_j) \in E$. The *TFP* technique achieves this as follows: as soon as x_i has been calculated, messages of the form $\langle v_j, x_i \rangle$ are sent to all successors $(v_i, v_j) \in E$. These messages are kept in a minimum priority queue sorting the items by their recipients according to \prec_T . By construction, the algorithm only starts the computation v_i once all predecessors $v_j \prec_T v_i$ are completed. Since these predecessors already removed their messages from the PQ, items addressed to v_i (if any) are currently the smallest elements in the data structure and can be dequeued. Using a suited EM PQ [23, 22], *TFP* incurs $\mathcal{O}(\text{sort}(k))$ I/Os, where k is the number of messages sent.

5.3 Randomization Schemes

Here, we summarize the randomization schemes *ES* [246] and Curveball for simple undirected graphs [80], and then discuss the notion of global trades. Since these algorithms iteratively modify random parts of a graph, they can be analyzed as finite Markov chains. It is well known that any finite, irreducible, aperiodic, and symmetric Markov chain converges to the uniform distribution on its state space (e.g. [217]). Its *mixing time* indicates the number of steps necessary to reach the stationary distribution.

5.3.1 Edge Switching

Edge Switching is a state-of-the-art randomization method with a wide range of applications, e.g. the generation of graphs [168, 210], or the randomization of biological datasets [185]. In each step, *ES* chooses two edges $e_1 = [u_1, v_1]$, $e_2 = [u_2, v_2]$ and a direction $d \in \{0, 1\}$ uniformly at random and rewires them into $\{u_1, u_2\}, \{v_1, v_2\}$ if $d=0$ and $\{u_1, v_2\}, \{v_1, u_2\}$ otherwise. If a step yields a non-simple graph, it is skipped. *ES*'s Markov chain is irreducible [118], aperiodic and symmetric [151] and hence converges to the uniform distribution on the space of simple graphs with fixed degree sequence. While analytic bounds on the mixing time [156, 157] are impractical, usually a number of steps linear in the number of edges is used in practice [280].

5.3.2 Simple Undirected Curveball Algorithm

Curveball is a novel randomization method. In each step, two nodes trade their neighborhoods, possibly yielding faster mixing times [80, 321, 331].

Definition 5.1 (Simple Undirected Trade). Let $G = (V, E)$ be a simple graph, A be its adjacency list representation, and A_u be the set of neighbors of node u . A trade $t = (i, j, \sigma)$ from A to adjacency list B is defined by two nodes i and j , and a permutation $\sigma: D_{ij} \rightarrow D_{ij}$ where $A_{i-j} := A_i \setminus (A_j \cup \{j\})$ and $D_{ij} := A_{i-j} \cup A_{j-i}$. As shown in Figure 5.1, performing t on G results in

$$\begin{aligned} B_i &= (A_i \setminus A_{i-j}) \cup \{x \mid x \in D_{ij}, \text{rank}_\sigma(x) \leq |A_{i-j}|\} \quad \text{and} \\ B_j &= (A_j \setminus A_{j-i}) \cup \{x \mid x \in D_{ij}, \text{rank}_\sigma(x) > |A_{i-j}|\}. \end{aligned}$$

Since edges are undirected, symmetry has to be preserved: for all $u \in A_i \setminus B_i$ the label j in adjacency list B_u is changed to i and analogously for $A_j \setminus B_j$. \blacktriangleleft

Simple Undirected Curveball randomizes a graph by repeatedly selecting a pair of nodes $\{i, j\}$ and a permutation σ on the disjoint neighbors uniformly at random. Its Markov chain is irreducible, aperiodic and symmetric. Therefore, it converges to the uniform distribution [81].

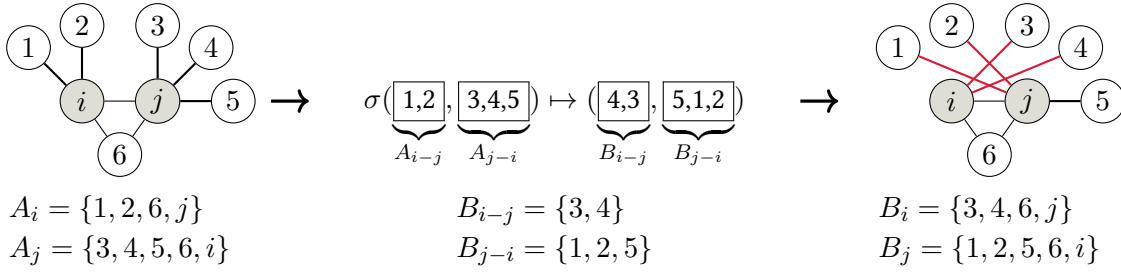


Figure 5.1: The trade (i, j, σ) between nodes i and j only considers edges to the disjoint neighbors $\{1, \dots, 5\}$. For the reassigned disjoint neighbors we use the short-hand $B_{i-j} := \{x \mid x \in D_{ij}, \text{rank}_\sigma(x) \leq |A_{i-j}|\}$ and $B_{j-i} := \{x \mid x \in D_{ij}, \text{rank}_\sigma(x) > |A_{i-j}|\}$. The triangle $(i, j, 6)$ is omitted as trading any of its edges would either introduce parallel edges, self-loops, or result in no change at all. Then, the given σ exchanges four edges.

5.3.3 Undirected Global Trades

Trade sequences typically consist of pairs in which each constituent is drawn uniformly at random. While it is a well known fact³ that $\Theta(n \log n)$ trades are required in expectation until each node is included at least once, there is no apparent reason why this should be beneficial; in fact, experiments in Section 5.5 suggest the contrary.

Carstens et al. propose the notion of *global trades* for directed or bipartite graphs as a 2-partition of all nodes implicitly forming $n/2$ node pairs to be traded in a single step [81]. This concept fails for undirected graphs where in general the two directions (u, v) and (v, u) of an edge $\{u, v\}$ cannot be processed independently in a single step. We hence extend global trades to undirected graphs by interpreting them as a sequence of $n/2$ simple trades, which together target each node exactly once (we assume n to be even; if this is not the case we add an isolated node⁴). Dependencies are then resolved by the order of this sequence.

Definition 5.2 (Undirected Global Trade). Let $G = (V, E)$ be a simple undirected graph and $\pi: V \rightarrow V$ be a permutation on the set of nodes. A *global trade* $T = (t_1, \dots, t_\ell)$ for $\ell = \lfloor n/2 \rfloor$ is a sequence of trades $t_i = \{\pi(v_{2i-1}), \pi(v_{2i}), \sigma_i\}$. By applying T to G we mean that the trades t_1, \dots, t_ℓ are applied successively starting with G . ◀

Theorem 5.3 allows us to use global trades as a substitute for a sequence of single trades, as global trades preserve the stationary distribution of Curveball's Markov chain. The proof extends [81], which shows convergence of global trades in bipartite or directed graphs, to undirected graphs and uses similar techniques.

Theorem 5.3. Let $G = (V, E)$ be an arbitrary simple undirected graph, and let Ω_G be the set of all simple undirected graphs that have the same degree sequence as G . Curveball with global trades started at G converges to the uniform distribution on Ω_G . ◀

Proof. In order to prove the claim, we have to show irreducibility and aperiodicity of the Markov chain as well as symmetry of the transition probabilities.

³For instance studied as the coupon collector problem.

⁴This is equivalent to randomly excluding a single node from a global trade

For the first two properties it suffices to show that whenever there exists a single trade from state A to B , there also exists a global trade from A to B (see [79] for a similar argument).⁵

Observe that there is a non-zero probability that a single trade does not change the graph, e.g. by selecting σ_i as the identity. Hence there is a non-zero probability that ...

- ... a global trade does not alter the graph at all. This corresponds to a self-loop at each state of the Markov chain and hence guarantees aperiodicity.
- ... all but one single trade of a global trade do not alter the graph. In this case, a global trade degenerates to a single trade and the irreducibility shown in [79] carries over.

It remains to show that the transition probabilities are symmetric. Let \mathcal{T}_{AB}^g be the set of global trades that transform state A to state B . Then the transition probability between A and B equals the sum of probabilities of selecting a trade sequence from \mathcal{T}_{AB}^g . That is $P_{AB} = \sum_{T \in \mathcal{T}_{AB}^g} \mathbf{P}_A(T)$ where $\mathbf{P}_A(T)$ denotes the probability of selecting global trade T in state A .

The probability $\mathbf{P}_A(t)$ of selecting a single trade $t = (i, j, \sigma)$ from state A to state B equals the probability $\mathbf{P}_B(\tilde{t})$ of selecting the reverse trade $\tilde{t} = (i, j, \sigma^{-1})$ from state B to A [81]. We now define the reverse global trade of $T = (t_1, \dots, t_\ell)$ as $\tilde{T} = (\tilde{t}_\ell, \dots, \tilde{t}_1)$. It is straight-forward to check that this gives a bijection between the sets \mathcal{T}_{AB}^g and \mathcal{T}_{BA}^g .

It remains to show that the middle equality holds in

$$P_{AB} = \sum_{T \in \mathcal{T}_{AB}^g} \mathbf{P}_A(T) \stackrel{!}{=} \sum_{\tilde{T} \in \mathcal{T}_{BA}^g} \mathbf{P}_B(\tilde{T}) = P_{BA}.$$

Let $T = (t_1, \dots, t_\ell)$ be a global trade from state A to state B as implied by π and $A = A_1, \dots, A_{\ell+1} = B$ be the intermediate states. We denote the reversal of T and π as \tilde{T} and $\tilde{\pi}$ respectively and obtain

$$P_A(T) = \mathbb{P}[\pi] \mathbf{P}_{A_1}(t_1) \dots \mathbf{P}_{A_\ell}(t_\ell) = \mathbb{P}[\tilde{\pi}] \mathbf{P}_B(\tilde{t}_\ell) \dots \mathbf{P}_{A_2}(\tilde{t}_1) = P_B(\tilde{T}).$$

Clearly $\mathbb{P}[\pi] = \mathbb{P}[\tilde{\pi}]$ as we are picking permutations uniformly at random. The second equality follows from $\mathbf{P}_A(t) = \mathbf{P}_B(\tilde{t})$ for a single trade between A and B . \square

5.4 Novel Curveball Algorithms for Undirected Graphs

In this section we present the related algorithms EM-CB, IM-CB, EM-GCB and EM-PGCB. They receive a simple graph G and a *trade sequence* $T = [\{u_i, v_i\}]_{i=1}^\ell$ as input and compute the result of carrying out the trade sequence T (see Section 5.3.2) in order.

EM-CB and IM-CB are sequential solutions suited to process arbitrary trade sequences T . For our analysis, we assume T 's constituents to be drawn uniformly at

⁵Since each global trade can be emulated by its $n/2$ decomposed single trades, the reverse is true for a hop of $n/2$ single trade steps. Due to dependencies however the transition probabilities generally do not match, see $V = \{1, 2, 3, 4\}$ and $E = \{[1, 2], [3, 4]\}$ for a simple counterexample.

random (as expected in typical applications). Both algorithms share a common design, but differ in the data structures used. EM-CB is an I/O-efficient algorithm while IM-CB is optimized for small graphs by using unstructured accesses to RAM. In contrast, EM-GCB and EM-PGCB process global trades only. This restricted input model allows to represent the trade sequence T implicitly by hash functions which further accelerates trading.

At core, all algorithms perform trades in a similar fashion: In order to carry out the i -th trade $\{u_i, v_i\}$, they retrieve the neighborhoods \mathcal{A}_{u_i} and \mathcal{A}_{v_i} , shuffle⁶ them, and then update the graph. Once the neighborhoods are known, trading itself is simple. We compute the set of disjoint neighbors $D = (\mathcal{A}_{u_i} \cup \mathcal{A}_{v_i}) \setminus (\mathcal{A}_{u_i} \cap \mathcal{A}_{v_i})$ and then draw $|\mathcal{A}_{u_i} \cap D|$ nodes from D for u_i uniformly at random while the remaining nodes go to v_i . If \mathcal{A}_{u_i} and \mathcal{A}_{v_i} are sorted this requires only $\mathcal{O}(|\mathcal{A}_{u_i}| + |\mathcal{A}_{v_i}|)$ work and scan $(|\mathcal{A}_{u_i}| + |\mathcal{A}_{v_i}|)$ I/Os (see also proof of Lemma 5.6 if the neighborhoods fit into RAM). Hence we focus on the harder task of obtaining and updating the adjacency information.

5.4.1 EM-CB: A Sequential I/O-efficient Curveball Algorithm

EM-CB (Algorithm 7) is an I/O-efficient Curveball algorithm to randomize undirected graphs. This basic algorithm already contains crucial design principles which we further explore with IM-CB, EM-GCB and EM-PGCB in sections 5.4.2 and 5.4.4 respectively.

The algorithm encounters the following challenges. After an undirected trade $\{u, v\}$ is carried out, it does not suffice to only update the neighborhoods \mathcal{A}_u and \mathcal{A}_v : consider the case that edge $\{u, x\}$ changes into $\{v, x\}$. Then the switch also affects the neighborhood of \mathcal{A}_x . Here, we call u and v *active* nodes while x is a *passive* neighbor.

In the EM setting another challenge arises for graphs exceeding main memory; it is prohibitively expensive to directly access the edge list since this unstructured pattern triggers $\Omega(1)$ I/Os for each edge processed with high probability.

EM-CB approaches these issues by abandoning a classical static graph data structure containing two redundant copies of each edge. Following the *TFP* principle⁶, we rather interpret all trades as a sequence of points over time that are able to receive messages. Initially, we send each edge to the earliest trade one of its endpoints is active in.⁷ This way, the first trade receives one message from each neighbor of the active nodes and hence can reconstruct \mathcal{A}_{u_1} and \mathcal{A}_{v_1} . After shuffling and reassigning the disjoint neighbors, EM-CB sends each resulting edge to the trade which requires it next. If no such trade exists, the edge can be finalized by committing it to the output.

The algorithm hence requires for each (actively or passively) traded node u , the index of the next trade in which u is actively processed. We call this the *successor* of u and define it to be ∞ if no such trade exists. The dependency information is obtained in a preprocessing step; given $T = [\{u_i, v_i\}]_{i=1}^{\ell}$, we first compute for each node u the monotonically increasing index list $\mathcal{S}(u)$ of trades in which u is actively processed, i.e. $\mathcal{S}(u) := [i \mid u \in t_i \text{ for } i \in [\ell]] \circ [\infty]$.

⁶In contrast to Definition 5.2, we do not consider the permutation σ of disjoint neighbors as part of the input, but let the algorithm choose one randomly for each trade.

⁷If an edge connects two nodes that are both actively traded we implicitly perform an arbitrary tie-break.

Algorithm 7: EM-CB

Data: Trade sequence T , simple graph $G = (V, E)$ by edge list E
// Preprocessing: Compute Dependencies

- 1 **foreach** trade $t_i = (u, v) \in T$ for increasing i **do**
- 2 Send messages $\langle u, t_i \rangle$ and $\langle v, t_i \rangle$ to Sorter `SORTERTTOV`
- 3 Sort `SORTERTTOV` lexicographically *// All trades of a node are next to each other*
- 4 **foreach** node $u \in V$ **do**
- 5 Receive $\mathcal{S}(u) = [t_1, \dots, t_k]$ from k messages addressed to u in `SORTERTTOV`
- 6 Set $t_{k+1} \leftarrow \infty$ *// $t_1 = \infty$ iff u is never active*
- 7 Send $\langle t_i, u, t_{i+1} \rangle$ to `SORTERDEPCHAIN` for $i \in [k]$
- 8 **foreach** directed edge $(u, v) \in E$ **do**
- 9 **if** $u < v$ **then**
- 10 Send message $\langle v, u, t_1 \rangle$ via `PQVTOV`
- 11 **else**
- 12 Receive t_1^v from unique message received via `PQVTOV`
- 13 **if** $t_1 \leq t_1^v$ **then** Send message $\langle t_1, u, v, t_1^v \rangle$ via `PQTTOV`
- else** Send message $\langle t_1^v, v, u, t_1 \rangle$ via `PQTTOV`
- 14 Sort `SORTERDEPCHAIN`
- // Main phase – Currently at least the first trade has all information it needs*
- 15 **foreach** trade $t_i = (u, v) \in T$ for increasing i **do**
- 16 Receive successors $\tau(u)$ and $\tau(v)$ via `SORTERDEPCHAIN`
- 17 Receive neighbors $\mathcal{A}_G(u)$, $\mathcal{A}_G(v)$ and their successors $\tau(\cdot)$ from `PQTTOV`
- 18 Randomly reassign disjoint neighbors, yielding new neighbors $\mathcal{A}'_G(u)$ and $\mathcal{A}'_G(v)$.
- 19 **foreach** $(a, b) \in (\{u\} \times \mathcal{A}'_G(u)) \cup (\{v\} \times \mathcal{A}'_G(v))$ **do**
- 20 **if** $\tau_a = \infty$ and $\tau_b = \infty$ **then** Output final edge $\{a, b\}$
- else if** $\tau_a \leq \tau_b$ **then** Send message $\langle \tau_a, a, b, \tau_b \rangle$ via `PQTTOV`
- else** Send message $\langle \tau_b, b, a, \tau_a \rangle$ via `PQTTOV`

Example 5.4. Let $G = (V, E)$ be a simple graph with $V = \{v_1, v_2, v_3, v_4\}$ and trade sequence $T = [t_1: \{v_1, v_2\}, t_2: \{v_3, v_4\}, t_3: \{v_1, v_3\}, t_4: \{v_2, v_4\}, t_5: \{v_1, v_4\}]$. Then, the successors \mathcal{S} follow as $\mathcal{S}(v_1) = [1, 3, 5, \infty]$, $\mathcal{S}(v_2) = [1, 4, \infty]$, $\mathcal{S}(v_3) = [2, 3, \infty]$, $\mathcal{S}(v_4) = [2, 4, 5, \infty]$. ◀

This information is then spread via two channels:

- After preprocessing, EM-CB scans \mathcal{S} and T conjointly and sends $\langle t_i, u_i, t_i^u \rangle$ and $\langle t_i, v_i, t_i^v \rangle$ to each trade t_i . The messages carry the successors t_i^u and t_i^v of the trade's active nodes.
- When sending an edge as described before, we augment it with the successor of the passive node. Initially, this information is obtained by scanning the edge list E and \mathcal{S} conjointly. Later, it can be inductively computed since each trade receives the successors of all nodes involved.

Lemma 5.5. For an arbitrary trade sequence T of length ℓ , EM-CB has a worst-case I/O complexity of $\mathcal{O}\left[\text{sort}(\ell) + \text{sort}(n) + \text{scan}(m) + \ell d_{\max}/B \log_{M/B}(m/B)\right]$. For r global trades, the worst-case I/O complexity is $\mathcal{O}(r[\text{sort}(n) + \text{sort}(m)])$. ◀

Proof. Refer to Section 5.A (Appendix) for the proof. ◻

5.4.2 IM-CB: An Internal Memory Version of EM-CB

While EM-CB is well suited if memory access is a bottleneck, we also consider the modified version IM-CB. As shown in Section 5.5, IM-CB is typically faster for small graph instances. IM-CB uses the same algorithmic ideas as EM-CB but replaces its priority queues and sorters⁸ by unstructured I/O into main memory (see Algorithm 8 (Appendix) for details):

- Instead of sending neighborhood information in a *TFP* fashion, we now rely on a classical adjacency vector data structure \mathcal{A}_G (an array of arrays). Similarly to EM-CB, we only keep one directed representation of an undirected edge. As an invariant, an edge is always placed in the neighborhood of the incident node traded before the other. To speed up these insertions, IM-CB maintains unordered neighborhood buffers.
- IM-CB does not forward successor information, but rather stores \mathcal{S} in a contiguous block of memory. The algorithm additionally maintains the vector $\mathcal{S}_{\text{idx}}[1 \dots n]$ where the i -th entry points to the current successor of node v_i . Once this trade is reached, the pointer is incremented giving the next successor.

Lemma 5.6. For a random trade sequence T of length ℓ , IM-CB has an expected running time of $\mathcal{O}(n + \ell + m + \ell m/n)$. In the case of r many global trades (each consisting of $n/2$ normal trades) the running time is given by $\mathcal{O}(n + rm)$. ◀

Proof. Refer to Section 5.B (Appendix) for the proof. ◻

5.4.3 EM-GCB: An I/O-efficient Global Curveball Algorithm

EM-GCB builds on EM-CB and exploits the regular structure of global trades to simplify and accelerate the dependency tracking. As discussed in Section 5.3.3, a global trade can be encoded as a permutation $\pi: [n] \rightarrow [n]$ by interpreting adjacent ranks as trade pairs, i.e. $T_\pi = [\{v_{\pi(2i-1)}, v_{\pi(2i)}\}]_{i=1}^{n/2}$. In this setting, a sequence of global trades is given by r permutations $[\pi_j]_{j=1}^r$. The model simplifies dependencies as it is not necessary to explicitly gather \mathcal{S} and communicate successors.

⁸The term *sorter* refers to a data structure with two modes of operation: items are first pushed into the write-only sorter in an arbitrary order by some algorithm. After an explicit switch, the filled data structure becomes read-only and the elements are provided as a lexicographically non-decreasing stream. It can be rewound at any time. While a sorter is functionally equivalent to sorting an EM vector, the restricted access model reduces constant factors in the implementation's runtime and I/O complexity [37].

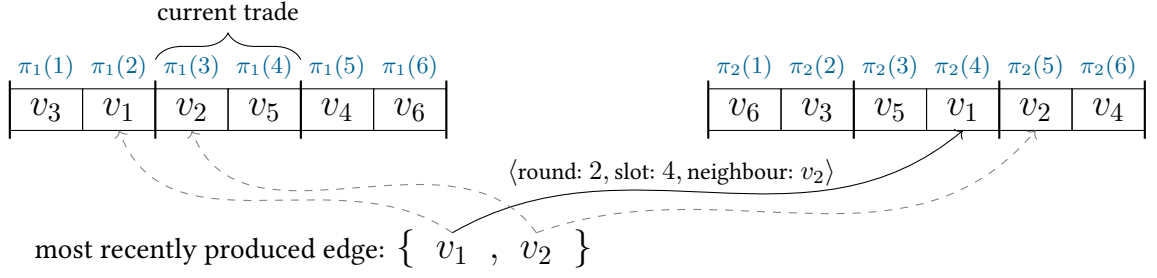


Figure 5.2: During the trade $j=1, i_1=3, i_2=4$ the edge $\{v_1, v_2\}$ is produced; the arrows indicate positions considered as successors. Since v_1 and v_2 are already processed in round $j=1$, π_2 is used to compute the successor. Then, the message is sent to v_1 in round 2 as v_1 is processed before v_2 .

As illustrated in Figure 5.2, we also change the addressing scheme of messages. While EM-CB sends messages to specific nodes in specific trades, EM-GCB exploits that each node v_i is actively traded only once in each round j and hence can be addressed by its position $\pi_j(i)$. Successors can then be computed in an ad-hoc fashion; let a trade of adjacent positions $i_1 < i_2$ of the j -th global trade produce (among others) the edge $\{v_x, v_y\}$. The successor of v_x (and analogously the one of v_y) is $\mathcal{S}_{j,i_2}[v_x] = (j, \pi_j(x))$ if v_x is processed later in round j (i.e. $\pi_j(x)/2 > i_2$) and otherwise $\mathcal{S}_{j,i_2}[v_x] = (j+1, \pi_{j+1}(x))$. Here we imply an untraded additional function $\pi_{r+1}(x) = x$ which avoids corner cases and generates an ordered edge list as a result of the r -th global trade.

To reduce the computational cost of the successor computation, EM-GCB supports fast injective functions $f: X \rightarrow Y$ where $[n] \subseteq X$ and $[n] \subseteq Y$. In contrast to the original permutations, their relevant image $\{f(x) \mid x \in [n]\}$ may contain gaps which are simply skipped by EM-GCB. This requires minor changes in the addressing scheme (see Section 5.C (Appendix)).

In practice, we use functions from the family of linear congruential maps H_p where p is the smallest prime number $p \geq n$:

$$H_p := \{h_{a,b} \mid 1 \leq a < p \text{ and } 0 \leq b < p\} \quad (5.1)$$

$$h_{a,b}(x) \equiv (ax + b) \pmod{p}, \quad (5.2)$$

As detailed in Section 5.D (Appendix) random choices from H_p are well suited for EM-GCB since they are 2-universal⁹ and contain only $\mathcal{O}(\log(n))$ gaps. They are also bijections with an easily computable inverse $h_{a,b}^{-1}$ that allows EM-GCB to determine the active node $h_{a,b}^{-1}(i)$ traded at position i ; this operation is only performed once for each traded position. EM-GCB can also support non-invertible functions using messages $\langle h(i), i \rangle$ that are generated for $1 \leq i \leq n$ and delivered using *TFP*.

5.4.4 EM-PGCB: An I/O-efficient Parallel Global Curveball Algorithm

EM-PGCB adds parallelism to EM-GCB by concurrently executing multiple sequential trades. As in Figure 5.3, we split a global trade into *microchunks* each containing a similar

⁹i.e. given one node in a single trade, the other is uniformly chosen among the remaining nodes.

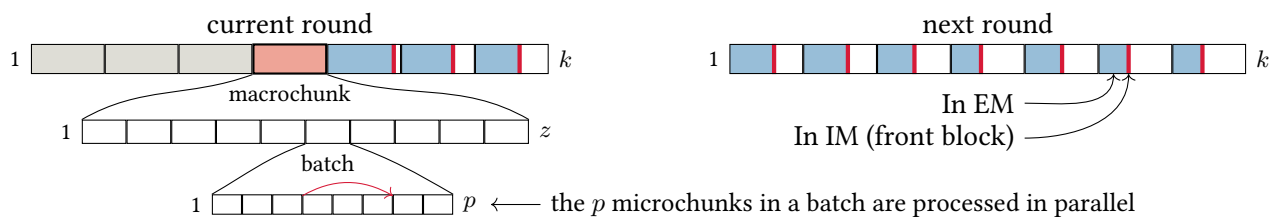


Figure 5.3: EM-PGCB splits each global trade into k *macrochunks* and maintains an external memory queue for each. Before processing a macrochunk, the buffer is loaded into IM and sorted, and further subdivided into z batches each consisting of p microchunks. A type (ii) message is visualized by the red intra-batch arrow.

number of node pairs and then execute a *batch* of p such subdivisions in parallel. The batch's size is a compromise between intra-batch dependencies (messages are awaited from another processor) and overhead caused by synchronizing threads at the batch's end (see Section 5.E (Appendix)).

EM-PGCB processes each microchunk similarly as in EM-CB but differentiates between messages that are sent (i) within a microchunk, (ii) between microchunks of the same batch (iii) and microchunks processed later. Each class is transported using an optimized data structure (see below).

Only type (ii) messages introduce dependencies between parallel path of execution. They are resolved as follows: when a processor retrieves the messages of its next trade, it checks whether all required data is available by comparing the number of messages to the active nodes' degrees. If data is missing the trade is skipped and later executed by the processor that adds the last missing neighbor.

For graphs with $m = \mathcal{O}(M^2/B)$ edges¹⁰, we optimize the communication structure for type (iii) messages. Observe that EM-PGCB sends messages only to the current and the subsequent round. We partition a round into k *macrochunks* each consisting of $\Theta(n/k)$ contiguous trades. An external memory queue is used for each macrochunk to buffer messages sent to it; in total, this requires $\Theta(kB)$ internal memory. Before processing a macrochunk, all its messages are loaded into IM, subsequently sorted and arranged such that missing messages can be directly placed to the position they are required in. This can also be overlapped with the processing of the previous macrochunk. As thoroughly discussed in Section 5.E (Appendix), the number k of macrochunks should be as small as possible to reduce overheads, but sufficiently large such that all messages of a macrochunk fit into main memory (see Section 5.F).

Theorem 5.7. EM-PGCB requires $\mathcal{O}(r[\text{sort}(n) + \text{sort}(m)])$ I/Os for r global trades. ◀

Proof. Observe that we can analyze each of the r rounds individually. A constant amount of auxiliary data is needed per node to provision gaps for missing data, to detect whether a trade can be executed and (if required) to invert the permutation. These $\Theta(n)$ messages require $\text{sort}(n)$ I/Os to be delivered. Using a PQ, the analysis of EM-CB (Lemma 5.5) carries over, requiring $\text{sort}(m)$ I/Os for a global trade. ◻

¹⁰Even with as little as 1 GiB of internal memory, several billion edges are supported.

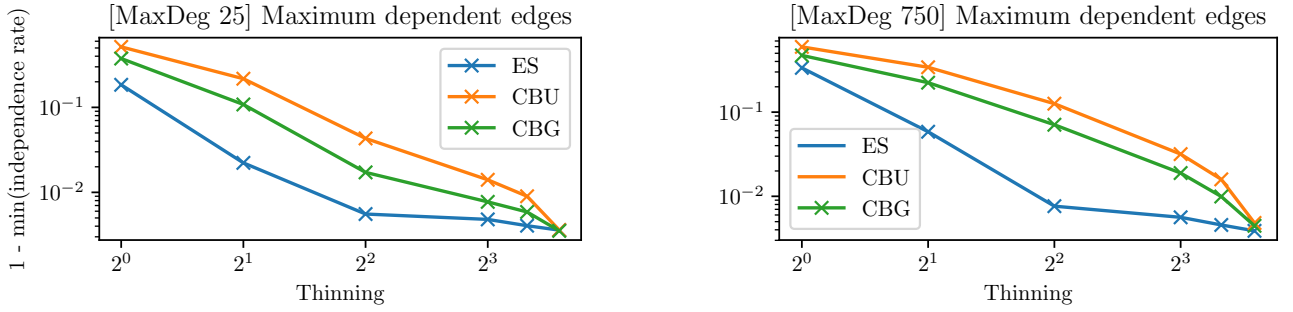


Figure 5.4: Fraction of edges still correlated as a function of the thinning parameter k for graphs with $n = 2 \cdot 10^3$ nodes and degree distribution PLD $((a, b), \gamma)$ with $\gamma = 2$, $a = 5$, and $b \in \{25, 750\}$. The (not thinned) long Markov chains of edge switching (ES), Curveball with uniform trades (CBU) and Curveball with global trades (CBG) contain 6000 super steps each.

5.5 Experimental Evaluation

In this section we evaluate the quality of the proposed algorithms and analyze the runtime of our C++ implementations.¹¹ EM-CB, IM-CB, EM-GCB are designed as modules of *NetworKit* [316]; due to their superior performance, only the latter two were added to the library and are available since release 4.6. EM-PGCB’s implementation is developed separately and facilitates external memory data structures and algorithms of STXXL [102].

Intuitively, graphs with skewed degree distributions are hard instances for Curveball since it shuffles and reassigns the disjoint neighbors of two trading nodes. Hence, limited progress is achieved if a high-degree node trades with a low-degree node. Since our experiments support this hypothesis, we focus on graphs with powerlaw degree distributions as difficult but highly relevant graph instances. Our experiments use two parameter sets:

- (*lin*) – The maximal possible degree scales linearly in the number n of nodes. The degree distribution PLD $((a, b), \gamma)$ is chosen as $a = 10$, $b = n/20$ and $\gamma = 2$.
- (*const*) – The extremal degrees are kept constant. In this case the parameters are chosen as $a = 50$, $b = 10000$ and $\gamma = 2$.

We select these configurations to be comparable with [168] where both parameter sets are used to evaluate EM-ES. The first setting (*lin*) considers the increasing average degree of real-world networks as they grow. The second setting (*const*) approximates the degree distribution of the Facebook network in May 2011 [168]. Runtimes are measured on the following off-the-shelf machine: Intel Xeon E5-2630 v3 (8 cores at 2.40GHz), 64GB RAM, 2× Samsung 850 PRO SATA SSD (1 TB), Ubuntu Linux 16.04, GCC 7.2.

5.5.1 Mixing of Edge-Switching, Curveball and Global Curveball

We are not aware of any practical theoretical bounds on the mixing time of Markov chains of *Curveball*, *Global Curveball* or *Edge Switching*. Hence, we quantitatively study

¹¹Code used for the presented benchmarks can be found at our fork <https://github.com/hthetran/networkit> (IM-CB and EM-CB) and <https://github.com/massive-graphs/extmem-lfr> (EM-PGCB).

the progress made by Curveball trades compared to edge switching and approximate the mixing time of the underlying Markov chains by a method developed in [281]. This criterion is a more sensitive proxy to the mixing time than previously used alternatives, such as the local clustering coefficient, triangle count and degree assortativity [168].

Intuitively, one determines the number of Markov chain steps required until the correlation to the initial state decays. Starting from an initial graph G_0 , the Markov chain is executed for a large number of steps, yielding a sequence $(G_t)_{t \geq 0}$ of graphs evolving over time. For each occurring edge e , we compute a Boolean vector $(Z_{e,t})_{t \geq 0}$ where a 1 at position t indicates that e exists in graph G_t . We then derive the k -thinned series $(Z_{e,t}^k)_{t \geq 0}$ only containing every k -th entry of the original vector $(Z_{e,t})_{t \geq 0}$ and use k as a proxy for the mixing time.

To determine if k Markov chain steps suffice for edge e to lose the correlation to the initial graph, the empirical transition probabilities of the k -thinned series $(Z_{e,t}^k)_{t \geq 0}$ are fitted to both an independent and a Markov model respectively. If the independent model is a better fit, we deem edge e to be independent. The results presented here consider only small graphs due to the high computational cost involved. However, additional experiments suggest that the results hold for graphs at least one order of magnitude larger.

We compare a sequence of uniform (single) trades, global trades and edge switching and visually align the results of these schemes by defining a *super step*. Depending on the algorithm a super step corresponds to either a single global trade, $n/2$ uniform trades or m edge swaps. Comparing $n/2$ uniform trades with a global trade seems sensible since a global trade consists of exactly $n/2$ single trades, furthermore randomizing with $n/2$ single trades considers the state of $2m$ edges which is also true for m edge swaps. It accounts for the fact that a single Curveball Markov chain step may execute multiple neighbor switches, thus easily outperforming *ES* in a step-by-step comparison.

Figure 5.4 contains a selection of results obtained for small powerlaw graph instances using this method (see Section 5.G.1 (Appendix) for the complete dataset). Progress is measured by the fraction of edges that are still classified as correlated, i.e. the faster a method approaches zero the better the randomization. We omit an in-depth discussion of uniform trades and rather focus on global trades which consistently outperform the former (cf. Section 5.3.2).

In all settings *ES* shows the fastest decay. The gap towards global trades grows temporarily as the maximal degree is increased which is consistent with our initial claim that skewed degree distributions are challenging for Curveball. The effect is however limited and in all cases performing 4 global trades for each edge switching super step gives better results. This is a pessimistic interpretation since typically $10m$ to $100m$ edge switches are used to randomize graphs in practice; in this domain global trades perform similarly well and 20 global trades consistently give at least the quality of $10m$ edge switches.

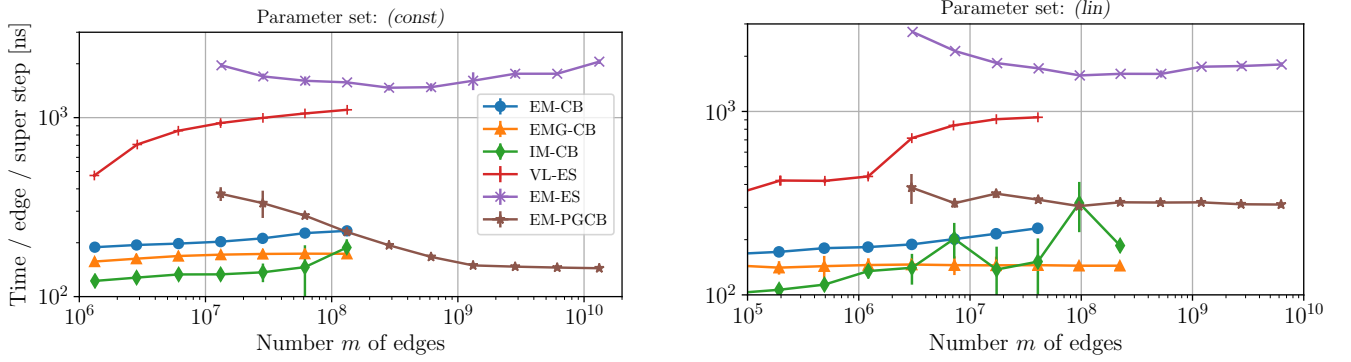


Figure 5.5: Runtime per edge and super step (global trade or m edge swaps) of the proposed algorithms IM-CB, EM-CB and EM-PGCB compared to state-of-the-art IM edge switching VL-ES and EM edge switching EM-ES. Each data point is the median of $S \geq 5$ runs over 10 super steps each. The left plot contains the *(const)*-parameter set, the right one *(lin)*. Observe that the super steps of different algorithms advance the randomization process at different speeds (see discussion).

5.5.2 Runtime Performance Benchmarks

We measure the runtime of the algorithms proposed in Section 5.4 and compare them to two state-of-the-art edge switching schemes (using the authors’ C++ implementations):

- VL-ES is a sequential IM algorithm with a hashing-based data structure optimized for efficient neighborhood queries and updates [332]. To achieve comparability, we removed connectivity tests, fixed memory management issues, and adopted the number of swaps.
- EM-ES is an EM edge switching algorithm and part of EM-LFR’s toolchain [168].

We carry out experiments using the *(const)* and *(lin)* parameter sets, and limit the problem sizes for internal memory algorithms to avoid exhaustion of the main memory. For each data point we carry out 10 super steps (i.e. 10 global trades or $10m$ edge swaps) on a graph generated with Havel-Hakimi from a random powerlaw degree distribution.

Figure 5.5 presents the wall-time per edge and super step including precomputation¹² required by the algorithms but excluding the initial graph generation process. The plots include (mostly small) errorbars corresponding to the unbiased estimation of the standard deviation of S repetitions per data point (with different random seeds).

The number k of macrochunks does not significantly affect EM-PGCB’s performance for small graphs due to comparably high synchronization cost. In contrast, adjusting k for larger graphs can noticeably increase the performance of EM-PGCB. We thus experimentally determined the value $k = 32$ for both *(const)* and *(lin)* with $n = 10^7$ nodes and use that value for all other instances.

All Curveball algorithms outperform their direct competitors significantly – even if we pessimistically executed two global trades for each edge switching super step (see Section 5.5.1). For large instances of *(const)* EM-PGCB carries out a super step 14.3 times faster than EM-ES and 5.8 times faster for *(lin)*. EM-PGCB also shows a superior scaling

¹²For VL-ES we report only the swapping process and the generation of the internal data structures.

behavior with an increasing speedup for larger graphs. Similarly, IM-CB processes super steps up to 6.3 times faster than VL-ES on (*const*) and 5.1 times on (*lin*).

On our test machine, the implementation of IM-CB outperforms EM-CB in the internal memory regime; EM-GCB is faster for large graphs. As indicated in Figure 5.9 (Section 5.G.2 (Appendix)), this changes qualitatively for machines with slower main memory and smaller cache; on such systems the unstructured I/O of IM-CB and VL-ES is more significant rendering EM-CB and EM-GCB the better choice with a speedup factor exceeding 8 compared to VL-ES.

5.6 Conclusion and Outlook

We applied *global* Curveball trades to undirected graphs simplifying the algorithmic treatment of dependencies and showed that the underlying Markov chain converges to a uniform distribution. Experimental results show that global trades yield an improved quality compared to a sequence of uniform trades of the same size.

We presented IM-CB and EM-CB, the first efficient algorithms for Simple Undirected Curveball algorithms; they are optimized for internal and external memory respectively. Our I/O-efficient parallel algorithm EM-PGCB exploits the properties of global trades and executes a super step 14.3 times faster than the state-of-the-art edge switching algorithm EM-ES; for IM-CB we demonstrate speedups of up to 6.3 (in a conservative comparison the speedups should be halved to account for the differences in mixing times of the underlying Markov chains). The implementations of all three algorithms are freely available and are in the process of being incorporated into EM-LFR and considered for *NetworKit*.

Acknowledgments

We thank the anonymous reviewers for their many insightful comments and suggestions.

Appendix 5.A EM-CB

Lemma 5.5. For an arbitrary trade sequence T of length ℓ , EM-CB has a worst-case I/O complexity of $\mathcal{O}\left[\text{sort}(\ell) + \text{sort}(n) + \text{scan}(m) + \ell d_{\max}/B \log_{M/B}(m/B)\right]$. For r global trades, the worst-case I/O complexity is $\mathcal{O}(r[\text{sort}(n) + \text{sort}(m)])$. ◀

Proof. As in Algorithm 7, EM-CB scans over T and E during preprocessing, and thereby triggers $\mathcal{O}(\text{scan}(\ell) + \text{scan}(m))$ I/Os. It also involves sorters SORTERTTOV and SORTERDEPCHAIN as well as priority queues PQVTOV and PQTTOV transporting $\mathcal{O}(\ell)$, $\mathcal{O}(\ell)$, $\mathcal{O}(n)$ and $\mathcal{O}(n)$ messages respectively. Hence preprocessing incurs $\mathcal{O}(\text{sort}(\ell) + \text{sort}(n) + \text{scan}(m))$ I/Os.

During the i -th trade $\mathcal{O}(\text{deg}(u_i) + \text{deg}(v_i))$ messages are retrieved shuffled and redistributed causing $\mathcal{O}[\text{sort}(\text{deg}(u_i) + \text{deg}(v_i))]$ I/Os. The bound can be improved to $\mathcal{O}\left((\text{deg}(u_i) + \text{deg}(v_i))/B \log_{M/B}(m/B)\right)$ by observing that $\mathcal{O}(m)$ items are stored in the PQ at any time. For a worst-case analysis we set $\text{deg}(u_i) = \text{deg}(v_i) = d_{\max}$ yielding the first claim.

Preprocessing of r global trades can be performed in r chunks of $n/2$ trades each. By arguments similar to the previous analysis, this yields an I/O complexity of $\mathcal{O}(r \text{sort}(n) + r \text{scan}(m))$. For the main phase, the above analysis tightens to $\mathcal{O}(r \text{sort}(m))$ using the fact that a single global trade targets each edge at most twice. ◻

Appendix 5.B IM-CB

Lemma 5.6. For a random trade sequence T of length ℓ , IM-CB has an expected running time of $\mathcal{O}(n + \ell + m + \ell m/n)$. In the case of r many global trades (each consisting of $n/2$ normal trades) the running time is given by $\mathcal{O}(n + rm)$. ◀

Proof. As detailed in Algorithm 8, the computation of $\mathcal{S}[\cdot]$ and its auxiliary structures involves scanning over T and V resulting in $\mathcal{O}(n + \ell)$ operations. Inserting all edges into \mathcal{A}_G requires another $\mathcal{O}(n + m)$ steps.

The i -th trade takes $\mathcal{O}(\text{deg}(v_i) + \text{deg}(u_i))$ time to retrieve the input edges and distribute the new states. To compute the disjoint neighbors, we insert \mathcal{A}_{u_i} into a hash set and subsequently issue one existence query for each neighbor in \mathcal{A}_{v_i} ; this takes expected time $\mathcal{O}(\text{deg}(v_i) + \text{deg}(u_i))$. Since T 's constituents are drawn uniformly at random, we estimate the neighborhood sizes as $\mathbb{E}[\text{deg}(u_i)] = \mathbb{E}[\text{deg}(v_i)] = m/n$ yielding the first claim. In case of r global trades, T consists of r groups with $n/2$ trades targeting all nodes each. Hence, trading requires time $r \sum_i (\text{deg}(u_i) + \text{deg}(v_i)) = r \sum_{v \in V} \text{deg}(v) = \mathcal{O}(rm)$. ◻

Appendix 5.C EM-GCB

Recall that a global trade can be encoded by a permutation $\pi: V \rightarrow V$ on the nodes or node indices (see Section 5.3.2). Consequently, generating a uniform random permuta-

Algorithm 8: IM-CB as detailed in Section 5.4.2.

Data: Trade sequence T , simple graph G

- 1 $\mathcal{S}_{\text{IDX}}[1 \dots n+1] \leftarrow 0$ // Compute \mathcal{S} : First count how often a node is active, ...
- 2 **foreach** $\{u, v\} \in T$ **do**
- 3 $\mathcal{S}_{\text{IDX}}[u] \leftarrow \mathcal{S}_{\text{IDX}}[u] + 1$
- 4 $\mathcal{S}_{\text{IDX}}[v] \leftarrow \mathcal{S}_{\text{IDX}}[v] + 1$
- 5 $\mathcal{S}_{\text{BEGIN}}[i] \leftarrow 1 + \sum_{j=1}^{i-1} \mathcal{S}_{\text{IDX}}[j] \quad \forall 1 \leq i \leq n+1$ // Exclusive prefix sum with stop marker
- 6 copy $\mathcal{S}_{\text{IDX}} \leftarrow \mathcal{S}_{\text{BEGIN}}$
- 7 Allocate $\mathcal{S}[1 \dots 2\ell]$
- 8 **foreach** $t_i = \{u_i, v_i\} \in T$ for increasing i **do**
- 9 $\mathcal{S}[\mathcal{S}_{\text{IDX}}[u_i]] \leftarrow i$ // Compute \mathcal{S} : ... when it is active
- 10 $\mathcal{S}_{\text{IDX}}[u_i] \leftarrow \mathcal{S}_{\text{IDX}}[u_i] + 1$
- 11 $\mathcal{S}[\mathcal{S}_{\text{IDX}}[v_i]] \leftarrow i$
- 12 $\mathcal{S}_{\text{IDX}}[v_i] \leftarrow \mathcal{S}_{\text{IDX}}[v_i] + 1$
- 13 reset $\mathcal{S}_{\text{IDX}} \leftarrow \mathcal{S}_{\text{BEGIN}}$
- 14 $\tau_{v_i} := \text{if } (\mathcal{S}_{\text{IDX}}[i] == \mathcal{S}_{\text{BEGIN}}[i+1]) \text{ then } \infty \text{ else } \mathcal{S}[\mathcal{S}_{\text{IDX}}[i]]$ // Short for read successor
- // Fill \mathcal{A}_G
- 15 $\mathcal{A}_{\text{BEGIN}}[i] \leftarrow 1 + \sum_{j=1}^{i-1} \deg(v_j) \quad \forall 1 \leq i \leq n+1$ // Prefix sum with stop marker
- 16 copy $\mathcal{A}_{\text{IDX}} \leftarrow \mathcal{A}_{\text{BEGIN}}$
- 17 Allocate $\mathcal{A}_G[1 \dots 2m]$
- 18 **foreach** $\{a, b\} \in E$ **do**
- 19 **if** $\tau_a \leq \tau_b$ **then** push b into $\mathcal{A}_G(a)$: $\mathcal{A}_G[\mathcal{A}_{\text{IDX}}[a]] \leftarrow b$; $\mathcal{A}_{\text{IDX}}[a] \leftarrow \mathcal{A}_{\text{IDX}}[a] + 1$
- else** push a into $\mathcal{A}_G(b)$: $\mathcal{A}_G[\mathcal{A}_{\text{IDX}}[b]] \leftarrow a$; $\mathcal{A}_{\text{IDX}}[b] \leftarrow \mathcal{A}_{\text{IDX}}[b] + 1$
- // Trade
- 20 **foreach** trade $t_i = (u, v) \in T$ for increasing i **do**
- 21 Gather neighbors $\mathcal{A}_G(u), \mathcal{A}_G(v)$ from \mathcal{A}_G using $\mathcal{A}_{\text{BEGIN}}$
- 22 Reset $\mathcal{A}_{\text{IDX}}[u] \leftarrow \mathcal{A}_{\text{BEGIN}}[u], \mathcal{A}_{\text{IDX}}[v] \leftarrow \mathcal{A}_{\text{BEGIN}}[v]$
- 23 Advance $\mathcal{S}_{\text{IDX}}[u]$ and $\mathcal{S}_{\text{IDX}}[v]$, s.t. τ_u and τ_v gets next trades
- 24 Randomly reassign disjoint neighbors, yielding new neighbors \mathcal{A}_u and \mathcal{A}_v .
- 25 **foreach** $(a, b) \in (\{u\} \times \mathcal{A}'_G(u)) \cup (\{v\} \times \mathcal{A}'_G(v))$ **do**
- 26 // Push node edge into \mathcal{A}_G ; same as line 18
- if** $\tau_a < \tau_b$ **then** Push b in $\mathcal{A}_G(a)$
- else** Push a in $\mathcal{A}_G(b)$

tion on $[n]$ yields a uniform random global trade. Injective hash functions have several computational advantages and can substitute the random permutation:

Definition 5.8 (Relaxed global trade). Let $h: [n] \rightarrow \mathbb{N}$ be an injective hash function and $[a_i]_{i=1}^n$ be the image $[h(i)]_{i=1}^n$ in sorted order. Further let $T_h = [t_i]_{i=1}^{n/2}$ where t_i trades the nodes with indices $h^{-1}(a_{2i-1})$ and $h^{-1}(a_{2i})$. Hence h implies the global trade T_h analogously to a permutation. ◀

In this setting, similar to using permutations, a sequence T of global trades is given by r hash functions $T = [h_i]_{i=1}^r$. Again, EM-GCB uses the fact that each node v_i is actively traded only once in each round j and can then be addressed by $h_j(i)$ (instead of previously $\pi_j(i)$).

Appendix 5.D Linear Congruential Maps

We use linear congruential maps as fast injective hash functions to model global trades for EM-PGCB. In this section, some of their useful properties are shown. We use the notation $\mathbb{Z}_p = \{0, 1, \dots, p-1\}$ and $\mathbb{Z}_p^* = \{1, \dots, p-1\}$ for p prime and implicitly use $0 \equiv p \pmod p$. Additionally for a map $h: X \rightarrow Y$ we denote the image of h as $\mathbf{im}(h) = \{h(x) : x \in X\}$.

Definition 5.9 (2-universal hashing). Let H be an ensemble of maps from X to Y and h be uniformly drawn from H . For finite X and Y we call the ensemble H 2-universal if for any two distinct $x_1, x_2 \in X$ and any two $y_1, y_2 \in Y$ and uniform random $h \in H$

$$\mathbb{P}[h(x_1) = y_1 \wedge h(x_2) = y_2] = |Y|^{-2}. \quad \blacktriangleleft$$

Proposition 5.10. A linear congruential map $h_{a,b}: \mathbb{Z}_p \rightarrow \mathbb{Z}_p, x \mapsto ax + b \pmod p$ for $a \neq 0$ and p prime is a bijection. ◀

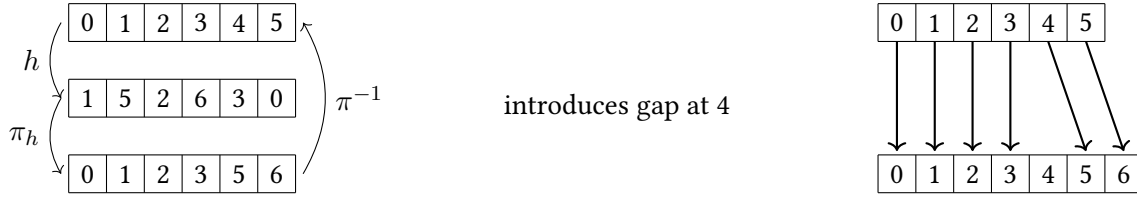
Proof. The translation $\tau_b(x) = x + b \pmod p$ and multiplication $\chi_a(x) = ax \pmod p$ is injective for all $a \in \mathbb{Z}_p^*$ and $b \in \mathbb{Z}_p$. Then, the composition $h_{a,b} = (\chi_a \circ \tau_b)$ is also injective and the inverse is given by $h_{a,b}^{-1}(y) = a^{-1}(y - b) \pmod p$. ◻

Lemma 5.11. The ensemble $H = \{h_{a,b} : a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$ is 2-universal. ◀

Proof. see Proposition 7 of [85]. ◻

The input size will most likely not be prime but linear congruential maps can still be used as injective maps since by the prime number theorem the next larger prime to a number n is on average $\mathcal{O}(\ln(n))$ larger. Additionally, since $[n]$ is a subset of \mathbb{Z}_p the 2-universality also already applies to distinct keys $x_1, x_2 \in [n]$. The small difference in n and p brings an additional feature we exploit while sending type (ii) messages (see Proposition 5.16): given a lower and upper bound on a hashed value with their respective ranks, one can estimate the rank of an element lying between those bounds.

Definition 5.12 (Sorted rank-map). Let $n \in \mathbb{N}$. Further, let $h: [n] \rightarrow \mathbb{N}$ be an injective map restricted to $[n]$ and π_h be the permutation that sorts $[h(i)]_{i=1}^n$ ascendingly. Denote



introduces gap at 4

Figure 5.6: The sorted rank-map for $n = 6$ and $h: [n] \rightarrow \mathbb{Z}_7, x \mapsto 4x + 1$. For the set $\{0, 1, 2, 3\}$ the sorted rank-map π is just the identity. In contrast for $x \in \{4, 5\}$ the value x is mapped to $\pi(x) = x + 1$.

with $\pi = (h \circ \pi_h): [n] \rightarrow \mathbf{im}(h)$ the *sorted rank-map*. It is clear that π is bijective, and π^{-1} remaps a mapped value to its rank in $\mathbf{im}(h)$, see Figure 5.6. ◀

Remark 5.13. The sorted rank-map π can only shift the original values and is thus monotonically increasing, see Figure 5.6. The shift in value is given by $\pi(x) - x$ and is monotonically increasing, too. By applying π we introduce gaps in the set \mathbb{Z}_p from $[n]$, refer to Figure 5.6. ◀

Proposition 5.14. Let $n \in \mathbb{N}$ and $p \geq n$ be a prime number. Further, let $h: [n] \rightarrow \mathbb{Z}_p$ be a linear congruential map and π be its sorted rank-map. If we want to compute the rank of $y \in \mathbf{im}(h)$ and know $x, x' \in [n]$ where $h(x) \leq y \leq h(x')$ then we can bound the rank $\pi^{-1}(y)$ of y by using the shifts of x and x' : $y - (\pi(x') - x') \leq \pi^{-1}(y) \leq y - (\pi(x) - x)$. ◀

Proof. The sorted rank-map π is by definition monotone increasing, see also Figure 5.6. It follows that $\pi(x) = x + k$, $\pi(x') = x' + k'$ and $k \leq k'$ for some $k, k' \in \mathbb{N}$. By monotonicity $\pi(\pi^{-1}(y)) = \pi^{-1}(y) + s$ for $s \in \{k, \dots, k'\}$, resulting in inequalities

$$\pi^{-1}(y) + k \leq y \leq \pi^{-1}(y) + k'.$$

By subtracting k and k' on both sides, the claim follows. ◻

With Proposition 5.14 we can reduce the number of candidates to search in. This is especially useful, when working on a smaller contiguous part of the data (EM-PGCB, Section 5.4.4).

Example 5.15. Let n and h be given from Figure 5.6. It is clear that the hashed-values are given by $\mathbf{im}(h) = \{0, 1, 2, 3, 5, 6\}$. Suppose the rank of 2 in $\mathbf{im}(h)$ has to be computed given the outer values e.g. that $\pi(0) = 0$ and $\pi(5) = 6$. Then by Proposition 5.14

$$\begin{aligned} 2 - (\pi(5) - 5) &\leq \pi^{-1}(2) \leq 2 - (\pi(0) - 0), \\ 1 &\leq \pi^{-1}(2) \leq 2. \end{aligned}$$

Thus, the rank of 2 in $\mathbf{im}(h)$ is either 1 or 2. ◀

Appendix 5.E EM-PGCB

EM-PGCB achieves parallelism by performing multiple trades concurrently. In contrast to EM-GCB, rather than only retrieving the first two necessary adjacency rows for the

single next trade, a whole chunk of data is loaded and maintained in IM-CB's adjacency list to store neighbors for a subset of nodes. The adjacency list is further used as a way to transport messages within a loaded macrochunk. Observe that at most $2m$ messages are sent in a global trade round since only neighborhood information is forwarded.

The idea is to split the messages into chunks of size $\mathcal{M} = cM$ where $c \in (0, 1)$ which can be processed in IM. For this, EM-PGCB loads and processes all messages targeted to the next n/k nodes for a constant k and performs the corresponding trades concurrently. This subdivides the messages and its processing into k *macrochunks*. If a macrochunk is too large, it cannot be fully kept in IM resulting in unstructured I/O in the trading process. The choice of k should therefore additionally consider the variance. An analysis on the size of the macrochunks is given in Section 5.F.

5.E.1 Data Structure for Message Transportation

Recall in Section 5.4.4 that each macrochunk is subdivided into many *microchunks* and processed in batches. During the trading process EM-PGCB has to differentiate between messages that are sent (i) within a microchunk, (ii) between microchunks of the same batch (iii) and microchunks processed later. To support both type (i) and type (ii) messages we organize the messages of the current macrochunk in an adjacency vector data structure similar to IM-CB. Instead of forwarding these messages in a *TFP* fashion, EM-PGCB inserts them directly into the adjacency data structure. We rebuild the data structure for each macrochunk requiring the degrees of the n/k loaded nodes to leave gaps if messages are missing. In a preprocessing step we provide EM-PGCB with this information by inserting messages $\langle h_r(v), \deg(v), v \rangle$ into a separate priority queue. Initializing the adjacency vector can now be done by loading the degrees for the next n/k targets and reserving for each target $h_r(v)$ the necessary $\deg(v)$ slots. Messages $\langle r, h_r(v), x \rangle$ targeted to the node v can then be inserted in an unstructured fashion in IM. This can be done in parallel for all targets in the macrochunk: first the retrieved messages are sorted in parallel and then accessed concurrently after determining delimiters by a parallel prefix sum over the message counts.

For a trade $t = \{u_i, v_i\}$ of targets $h_r(u_i)$ and $h_r(v_i)$ the assigned processor can determine if the t is tradable by checking whether $\deg(u_i)$ and $\deg(v_i)$ match the number of available messages. After performing the trade, we forward the updated adjacency information. Assume that the edge $\{u_i, x\}$ has to be sent to a later trade in the same global trade.

1. If x is traded within the processed microchunk there is no synchronization required and u_i can be inserted into the row corresponding to target $h_r(x)$.
2. If x is traded within the currently processed batch the processor has to insert u_i into the row corresponding to target $h_r(x)$ with synchronization. This yields a data dependency in the parallel execution. We can infer if the trade for x belongs to the current batch by comparing $h_r(x)$ to the maximum target of the batch.
3. If x is traded in a later microchunk, it either belongs to the same macrochunk or

a later one (of the same global trade). For the former EM-PGCB proceeds similar to type (ii) without processing foreign trades. In the latter case EM-PGCB inserts a message $\langle r, h_r(x), u_i \rangle$ into the priority queue.

Addressing the adjacency row of a target $h_r(u)$ can be done by computing the rank of $h_r(u)$ in the retrieved n/k targets. Since the separate priority queue provides all loaded targets by messages $\langle h_r(u), \deg(u), u \rangle$, we can perform a binary search and obtain the rank in time $\mathcal{O}(\log(n/k))$.

For linear congruential maps (Section 5.D) we can do better:

Proposition 5.16. Let h be a linear congruential map. Then, heuristically computing the row (rank) corresponding to $h(u)$ requires $\mathcal{O}(\log \log n)$ time. ◀

Proof. The next larger prime p to n is heuristically $\ln(n)$ larger than n . After loading all messages $\langle h(u), \deg(u), u \rangle$ for the current macrochunk the smallest and largest hashed value of the current macrochunk are known. By subtracting both values by the already processed number of targets and using Proposition 5.14 the search space can be reduced to $\mathcal{O}(\log n)$ elements. Application of a binary search on the remaining elements yields the claim. ◻

As already mentioned, if a trade has not received all its required messages, the assigned processor cannot perform the trade yet and therefore skips it. This can only happen within a batch when type (ii) messages occur. In Section 5.F we argue that this happens rarely. The processor that inserts the last message for that particular trade will perform it instead.

5.E.2 Improvements for Type (iii) Messages

Messages inserted into the priority queue need to contain the round-id to process global trades separately. Observe however that in a sequence of global trades, messages are only send to the current and subsequent round. We therefore modify our data structure, omitting the round from *every* message reducing the memory footprint significantly. Recall that, as an optimization for $m = \mathcal{O}(M^2/B)$ edges, EM-PGCB uses external memory queues for each of the k macrochunks of both global trade rounds.

A previously generated message $\langle r, h_r(u), x \rangle$ is now inserted into the corresponding queue containing messages for $h_r(u)$. Again, in a preprocessing step EM-PGCB determines for each queue its target range. For this, the separate priority queue containing messages $\langle h_r(u), \deg(u), u \rangle$ is read while extracting every (n/k) -th target (retrieving every element results in a sequence of sorted messages). This enables the computation of the correct queue for $h_r(u)$ with a binary search in time $\mathcal{O}(\log(k))$. Naturally since both the current and subsequent round are relevant, EM-PGCB employs k external memory queues for each. If a global trade is finished, the k EM queues of the currently processed and finished round can be reused for the next global trade. EM-PGCB's pseudo code can be found in Algorithm 9.

Algorithm 9: EM-PGCB as detailed in Section 5.4.4 and Section 5.E.

Data: Trade sequence $T = [h_i]_{i=1}^r$, simple graph $G = (V, E)$ as edge list E
Result: Randomized graph G'
// Initialization: provide auxiliary info and initialise with edges

- 1 **foreach** node $u \in V$ **do**
- 2 | Send $\langle h_1(u), \deg(u), u \rangle$ via `AUXINFOTOTARGET` *// Send node and degree to target*
- 3 Sort `AUXINFOTOTARGET` lexicographically
- 4 Scan `AUXINFOTOTARGET` and determine bounds for the k queues
- 5 **foreach** edge $e = [u, v]$ in E **do**
- 6 | Insert e according to h_1 into one of the corresponding queues

// Execution: Process rounds and macrochunks

- 7 **for** round $R = 1, \dots, r$ **do**
- 8 | **for** macrochunk $K = 1, \dots, k$ **do**
- 9 | Retrieve auxiliary data $\langle h_R(u), \deg(u), u \rangle$ from `AUXINFOTOTARGET`
- 10 | Load and sort messages of the K -th queue
- 11 | Insert the messages into the adjacency list \mathcal{A}_G in parallel
- 12 | **for** batch $\mathcal{B} = 1, \dots, z$ **do**
- 13 | **pardo** the i -th processor works on the i -th microchunk of batch \mathcal{B}
- 14 | **for** a trade $t = \{u, v\}$ **do**
- 15 | Retrieve A_u and A_v from \mathcal{A}_G
- 16 | With $\deg(u)$ and $\deg(v)$ determine whether tradable
- 17 | **if** tradable **then**
- 18 | Compute A'_u and A'_v
- 19 | Forward each resulting edge
- 20 | worksteal if inserted message fills all necessary data
- 20 | **else** Skip
- 21 | **if** $R < r$ **then**
- 22 | Clear `AUXINFOTOTARGET` and refill for h_{R+1} (repeat steps 3 to 5)

Appendix 5.F Analysis of EM-PGCB

5.F.1 Macrochunk Size

As already mentioned, the number of incoming messages may exceed the size of the internal memory M , since we partition the nodes into chunks which then may receive a different number of messages. Therefore some analysis on the size of the maximum macrochunk is necessary. Denote with $\mathcal{N}(\mu, \sigma^2)$ the distribution of a Gaussian r.v. with mean μ and variance σ^2 . A macrochunk holds the sum of n/k many iid degrees and is thus approximately Gaussian with mean $2m/k$ and variance $n/k \cdot \text{Var}(D)$ where D is distributed to the underlying degree distribution. This approximation gets better for larger values of n/k and is thus a suitable approximation for large graphs. Denote with S_1, \dots, S_k the sizes of all k macrochunks.

When determining a suitable choice of k , it is necessary to consider both the mean and the variance of the maximum macrochunk $\max_{1 \leq i \leq k} S_i$. The largest macrochunk may receive many high-degree nodes exceeding the size of the internal memory M . We thus bound its number in Corollaries 5.18 and 5.20.

Lemma 5.17. Let $Y = \max_{1 \leq i \leq k} X_i$, where the X_i are iid r.v. distributed as $\mathcal{N}(0, \sigma^2)$. Then, $\mathbb{E}[Y] \leq \sigma \sqrt{2 \log(k)}$. ◀

Proof. The following chain of inequalities holds

$$e^{t\mathbb{E}[Y]} \leq \mathbb{E}[e^{tY}] = \mathbb{E}\left[\max_{1 \leq i \leq k} e^{tX_i}\right] \leq \sum_{i=1}^k \mathbb{E}[e^{tX_i}] = ke^{t^2\sigma^2/2},$$

where in order Jensen's inequality¹³ monotonicity and non-negativity of the exponential function as well as the definition of the moment generating function of a Gaussian r.v. have been applied. Taking the natural logarithm and dividing by t on both sides (ruling out $t \neq 0$) yields $\mathbb{E}[Y] \leq \frac{\log(k)}{t} + \frac{t\sigma^2}{2}$, which is minimized by $t = \sqrt{2 \log(k)}/\sigma$. The above proof is a special case in a proof of [231]. ◻

Corollary 5.18. Let $Y = \max_{1 \leq i \leq k} S_i$. By approximating S_i with a Gaussian r.v. N_i with $\mu = \mathbb{E}[S_i]$ and $\sigma^2 = \text{Var}(S_i)$, one gets an approximate upper bound on Y :

$$\mathbb{E}[Y] \approx \mathbb{E}\left[\max_{1 \leq i \leq k} N_i\right] \leq \mathbb{E}[S_1] + \sqrt{2 \log(k) \text{Var}(S_1)} = \mathbb{E}[S_1] + \sqrt{\frac{n \log(k)}{2k} \text{Var}(D)}.$$

Proof. Since $\max_{1 \leq i \leq k} N_i$ is centered around μ , it is identically distributed to $\mu + \max_{1 \leq i \leq k} N'_i$ where N'_i has the same variance but is centered around 0. By applying Lemma 5.17 to $\max_{1 \leq i \leq k} N'_i$ the claim follows, since $\mathbb{E}[\max_{1 \leq i \leq k} N_i] = \mu + \mathbb{E}[\max_{1 \leq i \leq k} N'_i]$. ◻

Lemma 5.19. Let X_1, \dots, X_k be iid and $Y = \max_{1 \leq i \leq k} X_i$. Then, $\text{Var}(Y) \leq k \text{Var}(X_1)$. ◀

Proof. For Z, Z' iid. $\mathbb{E}[(Z - Z')^2] = 2 \text{Var}(Z)$ holds, since

$$\mathbb{E}[Z^2 - 2ZZ' + Z'^2] = 2\mathbb{E}[Z^2] - 2\mathbb{E}[Z]^2.$$

Now, let $Y' = \max_{1 \leq i \leq k} X'_i$ be an independent copy of Y and $r > 0$. First, the inequality $\mathbb{P}[|Y - Y'|^2 > r] \leq \sum_{i=1}^k \mathbb{P}[|X_i - X'_i|^2 > r]$ is shown. We show the implication that if $|Y - Y'|^2 > r$ there exists an index i such that $|X_i - X'_i|^2 > r$.

If $|Y - Y'|^2 > r$ holds, then w.l.o.g. let $Y = X_i$ and $Y' = X'_j$ and $Y > Y'$, such that $|X_i - X'_j|^2 > r$. By maximality the following inequality chain $X_i > X'_j \geq X'_i$ implies $|X_i - X'_i| > r$ and consequently $\mathbb{P}[|Y - Y'|^2 > r] \leq \mathbb{P}[\exists i: |X_i - X'_i| > r]$.

A union bound yields $\mathbb{P}[|Y - Y'|^2 > r] \leq \sum_{i=1}^k \mathbb{P}[|X_i - X'_i|^2 > r]$. Integrating r from 0 to ∞ yields $2 \text{Var}(Y) = \mathbb{E}[(Y - Y')^2] \leq k\mathbb{E}[(X_1 - X'_1)^2] = 2k \text{Var}(X_1)$, which concludes the proof. ◻

¹³Let f be convex. For a non-negative λ_i with $\sum_{i=1}^n \lambda_i = 1$ it follows $f(\sum_{i=1}^n \lambda_i x_i) \leq \sum_{i=1}^n \lambda_i f(x_i)$.

Corollary 5.20. Let $Y = \max_{1 \leq i \leq k} S_i$. Then, $\text{Var}(Y) \leq k \text{Var}(S_1) = n \text{Var}(D)$. ◀

Proof. This is a special case of Lemma 5.19. ◻

The probability mass of a Gaussian r.v. is concentrated around its mean, e.g. the tails vanish very quickly, see Proposition 5.21. This heuristically additionally holds true for the maximum macrochunk size (Lemma 5.22).

Proposition 5.21. Let X be a standard Gaussian r.v. and $f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$ be its probability density function. Let $t > 0$ then it holds

$$\mathbb{P}[X > t] \leq \exp(-t^2/2)/\sqrt{2\pi}/t = \mathcal{O}\left(\frac{e^{-t^2/2}}{t}\right). \quad \blacktriangleleft$$

Proof. The value of $\mathbb{P}[X > t]$ equals $\int_t^\infty \frac{1}{\sqrt{2\pi}} e^{-x^2/2} dx$. Since the integrating variable ranges from $[t, \infty)$ then $\frac{x}{t} \geq 1$ s.t. $\mathbb{P}[X > t] \leq \int_t^\infty \frac{x}{t} \frac{1}{\sqrt{2\pi}} e^{-x^2/2} dx = \frac{1}{t} \frac{e^{-t^2/2}}{\sqrt{2\pi}}$. ◻

Lemma 5.22. Let $Y = \max_{1 \leq i \leq k} N_i$ where N_i are iid standard Gaussian random variables. Then $\mathbb{P}[Y > t] = \mathcal{O}\left(k \exp(-t^2/2)/t\right)$. ◀

Proof. The claim follows by the following calculation:

$$\mathbb{P}[Y > t] = \mathbb{P}\left[\max_{1 \leq i \leq k} N_i > t\right] = \mathbb{P}[\exists i \text{ s.t. } N_i > t] \leq \sum_{i=1}^k \mathbb{P}[N_i > t] = \mathcal{O}\left(k \cdot \frac{e^{-t^2/2}}{t}\right).$$

If for any random variable $N_i > t$, then already $\max_{1 \leq i \leq k} N_i > t$, inversely if $\max_{1 \leq i \leq k} N_i > t$ then there exists a N_i such that $N_i > t$, which shows the first equality. After applying the union bound and Proposition 5.21 the claim follows. ◻

5.F.2 Heuristic on Intra-Batch Dependencies

In EM-PGCB, if information on an edge $\{u, w\}$ has to be inserted into the same batch a dependency arises. We will now argue that this happens not too often when the number of batches z is chosen sufficiently large.

Lemma 5.23. Let \mathcal{B} be the set of targets for a batch. Assuming uniform neighbors, the number of dependencies from \mathcal{B} to \mathcal{B} heuristically is $\binom{p}{2} \frac{2m}{k^2 z^2 p^2}$. ◀

Proof. By construction $|\mathcal{B}| = \frac{n}{kz}$ since \mathcal{B} is part of an equal subdivision of a macrochunk. Each individual microchunk consists of $\frac{n}{kzp}$ many targets for the same reason. The i -th microchunk therefore has $(p-i) \frac{n}{kzp}$ many critical targets. On average each microchunk generates $\text{avg deg} \frac{n}{kzp} = \frac{2m}{kzp}$ many messages that need to be forwarded. For an edge produced by the i -th microchunk assume uniformity on the neighbors A , then V_i is the number of critical messages where $V_i = \sum_{i=1}^n 1_{i \in A} 1_{i \in h^{-1}(\mathcal{B})}$. Its expectation is

$$\mathbb{E}[V_i] = \sum_{i=1}^n \mathbb{P}[i \in A] \mathbb{P}[i \in h^{-1}(\mathcal{B})] = n \frac{\text{deg}_{\text{avg}} \frac{n(p-i)}{kzp}}{n} = \text{deg}_{\text{avg}} \frac{p-i}{kzp}.$$

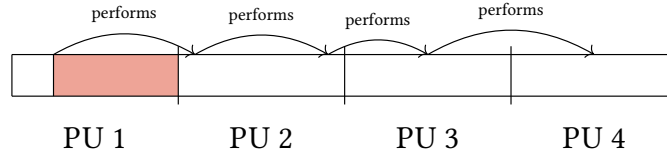


Figure 5.7: Work stealing of PU 1. The arrows represent a long work stealing chain of trades. The red marked area represents still untouched trades of the first microchunk that will get processed *after* the long chain by the first PU.

Now let the total number of messages from the i -th microchunk to \mathcal{B} be H_i . Since each microchunk holds $\frac{n}{kzp}$ many nodes, H_i is given by

$$\mathbb{E}[H_i] = \frac{n}{kzp} \mathbb{E}[V_i] = \frac{2m(p-i)}{k^2 z^2 p^2}.$$

By summing over all p microchunks, e.g. $\sum_{i=1}^p \mathbb{E}[H_i]$ the claim follows. \square

Example 5.24. Consider Lemma 5.23 where $m = 12 \times 10^9$, $k = 32$, $z = 2^{11}$ and $p = 16$. The average number of messages in the batch is given by $m/kz \geq 1.8 \times 10^5$. And Lemma 5.23 predicts a count of less than 4 critical messages on average in a batch. \blacktriangleleft

Theoretically by Lemma 5.23 the number of critical messages is very small if z is set to be sufficiently large. Therefore waiting and stalling for missing messages is inefficient and should be avoided. EM-PGCB thus skips a trade when it cannot be performed and is later executed by the processor that adds the last missing neighbor. However, since a work stealing processor spends time on a trade that is possibly assigned to another microchunk, it is not working on its own. Therefore messages coming from that particular microchunk are generated later down the line. This may be especially bad when a PU performs a chain of trades that it was not originally assigned to as illustrated in Figure 5.7. Since work stealing can only be done in a *TFP* fashion, the chain length therefore is geometrically distributed (in fact, the probability declines in each step since less targets are critical) and is thus whp. of order $\mathcal{O}(1)$ by Proposition 5.25.

Proposition 5.25. Let X be geometrically distributed with parameter $(1 - 1/z^2)$ for $z > 1$. Then, $\mathbb{P}[X > t] = \frac{1}{z^{2t}} = e^{-2 \ln(z)t}$. \blacktriangleleft

Proof. The claim follows by $\mathbb{P}[X > t] = 1/z^{2t}$ and setting $t = \mathcal{O}(1)$. \square

Appendix 5.G Additional Experimental Results

5.G.1 Swaps Performed by Curveball and Global Curveball

In Figure 5.8 we counted the number of neighborhood swaps in $n/2$ uniform trades and a single global trade and obtain the fraction of performed swaps to all possible swaps. These experiments are performed on a series of 10-regular graphs and powerlaw graphs with increasing maximum degree. Both algorithms perform a similar count of swaps and suggest no systematic difference. As expected, for regular graphs the fraction of performed swaps goes to $1/2$ for an increasing number of nodes, since with increasing n the number of common neighbors goes to zero. On the other hand the fraction of performed swaps decreases for powerlaw graphs with a higher maximum degree.

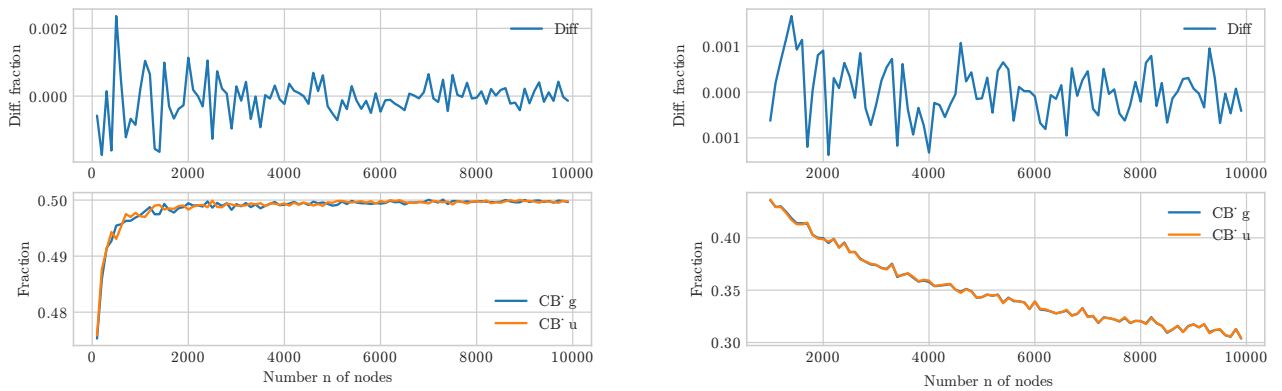


Figure 5.8: The average fraction of performed neighborhood swaps of $n/2$ uniform trades and a single global trade. **Left:** 10-regular graphs for increasing n . **Right:** powerlaw graphs realized from PLD $([10, n/20], 2)$ for increasing n by the HAVEL-HAKIMI algorithm.

5.G.2 Autocorrelation Time of Curveball and Edge Switching

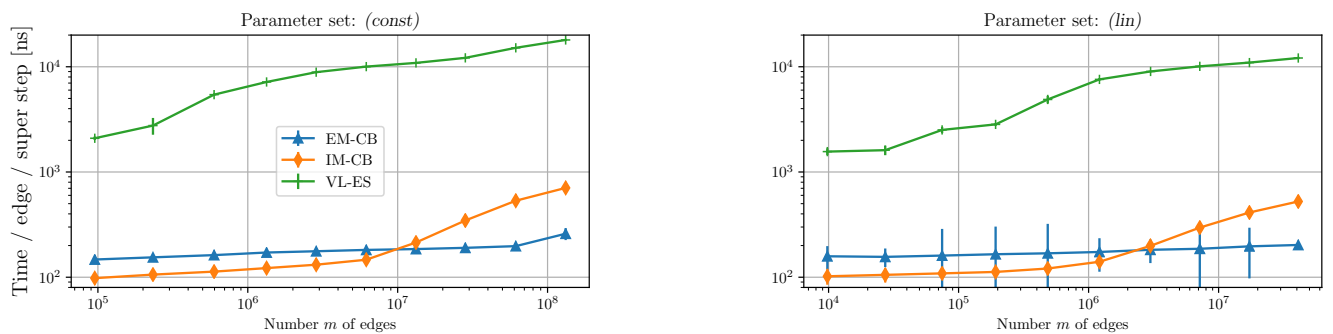


Figure 5.9: Runtime per edge and super step of IM-CB and EM-CB compared to state-of-the-art IM edge switching VL-ES. Each data point is the median of $S \geq 5$ runs over 10 super steps each. The left plot contains the (const)-parameter set, the right one (linear). Machine: Intel i7-6700HQ CPU (4 cores), 64 GB RAM, Ubuntu Linux 17.10.

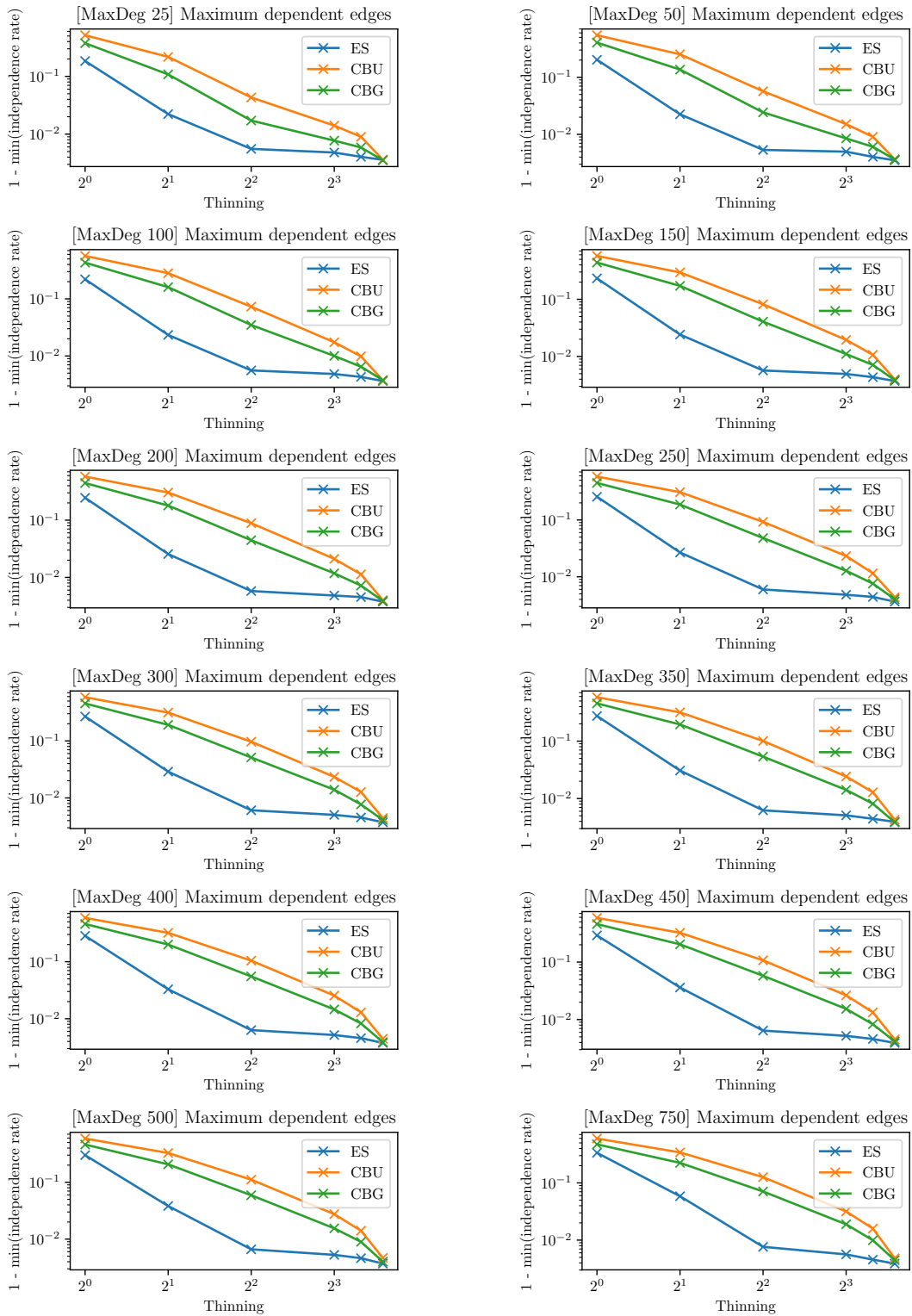


Figure 5.10: Fraction of edges still correlated as function of the thinning parameter k for graphs with $n = 2 \cdot 10^3$ nodes and degree distribution $\text{PLD}([a, b], \gamma)$ with $\gamma = 2$, $a = 5$, and several different values for b . The (not thinned) long Markov chains contain 6000 super steps each.

Generating practical random hyperbolic graphs in near-linear time and with sub-linear memory

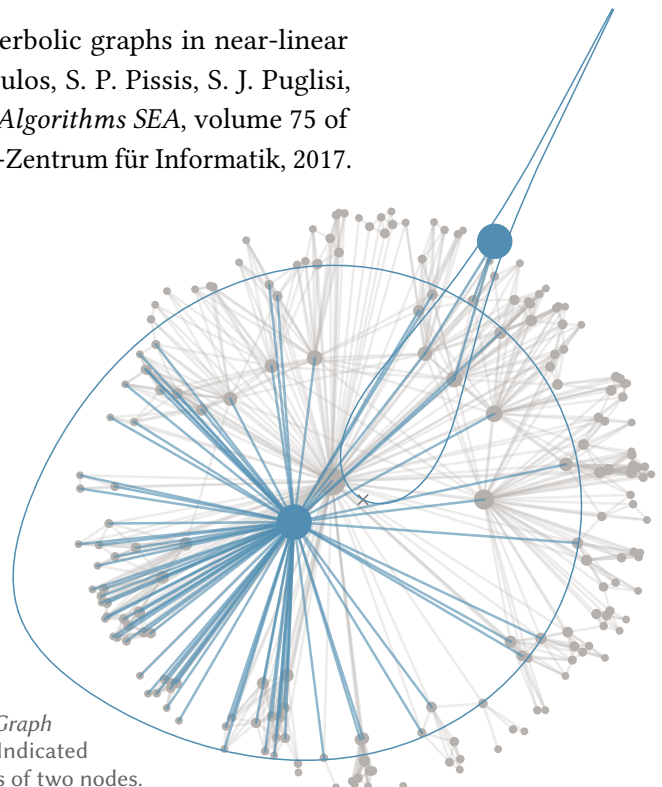
6

Random graph models, originally conceived to study the structure of networks and the emergence of their properties [59], have become an indispensable tool for experimental algorithmics. Amongst them, hyperbolic random graphs form a well-accepted family, yielding realistic complex networks while being both mathematically and algorithmically tractable. We introduce two generators MEMGEN and HYPERGEN for the $G_{\alpha,C}(n)$ -model, which distributes n random points within a hyperbolic plane and produces $m = n\bar{d}/2$ undirected edges for all point pairs close by; the expected average degree \bar{d} and exponent $2\alpha + 1$ of the powerlaw degree distribution are controlled by $\alpha > 1/2$ and C . Both algorithms emit a stream of edges which they do not have to store. MEMGEN keeps $\mathcal{O}(n)$ items in internal memory and has a time complexity of $\mathcal{O}(n \log \log n + m)$, which is optimal for networks with an average degree of $\bar{d} = \Omega(\log \log n)$. For realistic values of $\bar{d} = o(n / \log^{1/\alpha}(n))$, HYPERGEN reduces the memory footprint to $\mathcal{O}([n^{1-\alpha}\bar{d}^\alpha + \log n] \log n)$.

In an experimental evaluation, we compare HYPERGEN with four generators among which it is consistently the fastest. For small $\bar{d} = 10$ we measure a speedup of 4.0 compared to the fastest publicly available generator increasing to 29.6 for $\bar{d} = 1000$. On commodity hardware, HYPERGEN produces $3.7 \cdot 10^8$ edges per second for graphs with $10^6 \leq m \leq 10^{12}$ and $\alpha = 1$, utilizing less than 600 MB of RAM. We demonstrate nearly linear scalability on an Intel Xeon Phi.

This chapter is based on the peer-reviewed conference article [271]:

- [271] M. Penschuck. Generating practical random hyperbolic graphs in near-linear time and with sub-linear memory. In C. S. Iliopoulos, S. P. Pissis, S. J. Puglisi, and R. Raman, editors, *Int. Symp. on Experimental Algorithms SEA*, volume 75 of *LIPICs*, pages 26:1–26:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.SEA.2017.26 .



Threshold Random Hyperbolic Graph with $n = 200$, $\bar{d} = 10$, $\alpha = 1$. Indicated are the neighborhood distances of two nodes.

6.1 Introduction

Even though most practical algorithms aim for a good performance on real-world data, artificial benchmarks are crucial for their development. Suited real-world datasets are typically scarce, do not scale, may exhibit noise or have uncontrollable properties. Tunable synthetic instances based on random models alleviate these issues. They are indispensable for systematic experiments allowing to quantify an algorithm’s performance as a function of controllable parameters.

Selecting the right model depends on the use case: many real-world networks (e.g., communication or social networks) exhibit basic features, such as a small diameter, a powerlaw degree distribution, and a non-vanishing local cluster coefficient [14, 28, 238, 255]. Among suited models, geometric random networks seem most natural. They explain the high local clustering of social networks¹ by embedding the nodes into a geometric space. Then the distance between any two nodes determines the probability of an edge between them. While Euclidean space is appropriate for spatial networks (e.g., [160]), it distorts complex networks such as the internet graph for which hyperbolic embedding performs well [56, 303].

Clustering:
 [Section 1.2.5](#)

Threshold RHG:
 [Section 6.1.3](#)

The task of actually generating instances of hyperbolic random graphs has been approached recently yielding generators that are either fast in practice [224] or optimal in theory [68]. We target the generation of large instances whose set of nodes² does not fit into memory. Space requirements are crucial especially in the context of co-processors with small dedicated memory. Another application of such a generator is the experimental evaluation of streaming [146, 235] or external memory algorithms [7, 242]. Since our algorithm is typically faster than the time it takes to write data to disk, one can connect it to the algorithm under testing without a round trip to secondary storage. In such a case, the generator should leave the majority of memory to the main application in order to allow fast context switches.

6.1.1 Our Contribution

We introduce two related generators MEMGEN (Section 6.2) and HYPERGEN (Section 6.3) for the $G_{\alpha,C}(n)$ -model (Section 6.1.3) for large instances with n nodes and $m = \bar{d}n/2$ edges where \bar{d} is the expected average degree. Both generators target a streaming setting and are compatible with the external memory model for practical instances. MEMGEN requires $\mathcal{O}(n)$ internal memory and has a time complexity of $\mathcal{O}(n \log \log n + m)$, which is optimal for networks with an average degree of $\bar{d} = \Omega(\log \log n)$. For realistic values of $\bar{d} = o(n/\log^{1/\alpha}(n))$, HYPERGEN reduces the memory footprint to $\mathcal{O}([n^{1-\alpha}\bar{d}^\alpha + \log n] \log n)$, where $\alpha > 1/2$ controls the exponent $2\alpha + 1$ of the result’s powerlaw degree distribution. In an experimental evaluation (Section 6.5), HYPERGEN consistently the previously fastest generators we are aware of.

¹i.e., a high triangle count; roughly speaking two friends of a person are likely to acquaint too.

²Implementations typically use at least 80 byte/node (e.g., Section 6.5.2).

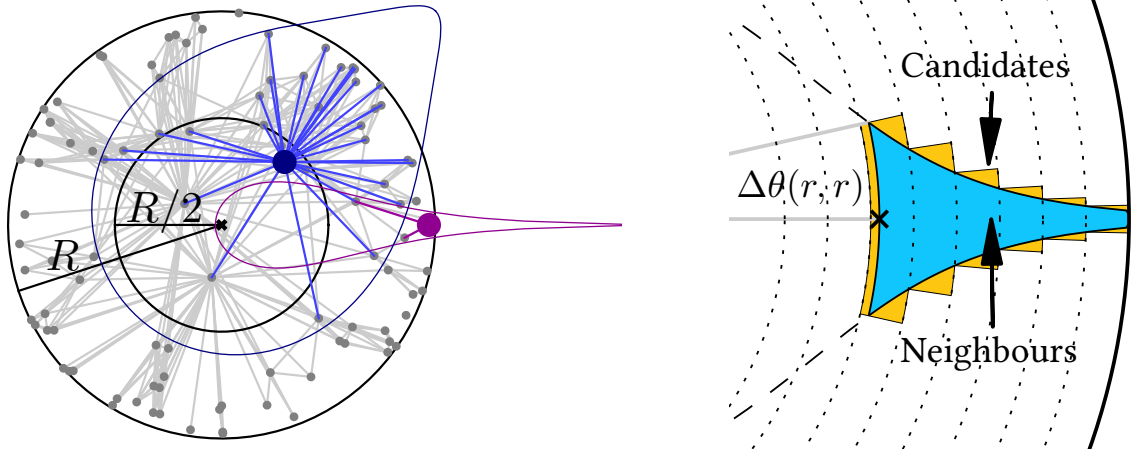


Figure 6.1: **Left:** The $G_{\alpha,C}(n)$ model with $n = 150$, $\alpha = 1$, $C = -2$. The area enclosed by each colored lobe corresponds to all points in distance at most R around its highlighted center. **Right:** Band model introduced by NKBAND (not to scale). The partial blue lobe indicates the area in which candidates can be found. The step-wise overestimation for candidate selection is shown in yellow.

In the quest for a smaller memory footprint, we increase the data locality leading to an easily parallelisable algorithm. While we only explore shared memory parallelism, HYPERGEN works in a distributed setting with constant communication.

6.1.2 Notation

Define $[k] := \{1, \dots, k\}$ for $k \in \mathbb{N}_{>0}$. A graph $G = (V, E)$ has $n = |V|$ sequentially numbered nodes $V = \{v_i\}_{i \in [n]}$ and $m = |E|$ edges. Unless stated differently, graphs are undirected, unweighted, and have an average degree of $\bar{d} = 2m/n$. Let $N_G(v) \subseteq V$ be the neighborhood of node v in graph G , i.e., the set of adjacent nodes.

We mainly consider points (r, θ) in polar coordinates where r is the radius (i.e., distance from the origin) and θ is the polar angle or azimuth. A point with radius r_1 is said to be *above* a point with radius r_2 if $r_1 > r_2$ and *below* if $r_1 < r_2$. Let $B_y(z)$ the ball of radius y and center at radius z , i.e., the set of all points with distance at most y from point³ z [159]. We apply standard set operations to balls where \setminus , \cup , and \cap denote set differences, union and intersection accordingly.

Let $\mu(X)$ denote the probability mass of X . We denote a Binomial distribution over n items with probability p as $\mathcal{B}(n, p)$.

Also refer to Section 6.A (Appendix) for a summary of definitions.

6.1.3 The Hyperbolic Random Graph Model $G_{\alpha,C}(n)$

We consider the well-accepted $G_{\alpha,C}(n)$ threshold model [159]. It follows the initial zero-temperature model of Krioukov et al. [200], but removes a redundant curvature parameter by fixing $\zeta = 1$.

$G_{\alpha,C}(n)$: Threshold RHG

³We omit the azimuth of z as it is irrelevant in our analysis due to polar symmetry.

Definition 6.1 (Gugelmann et al. [159]). Let $\alpha > 1/2$, $n \in \mathbb{N}_{>0}$, and $C > -2 \log n$. The random graph $G_{\alpha, C}(n) = (V, E)$ has the following properties (Section 6.1.3):

$R := 2 \log n + C$ radius
of hyperbolic disk

- Each node $v_i \in V = \{v_1, \dots, v_n\}$ is modeled by a random point $p_i = (r_i, \theta_i)$ in the hyperbolic disk.⁴ Its angular coordinate θ_i is drawn uniformly from $[0, 2\pi]$ while its radius $0 \leq r_i < R$ with $R := 2 \log n + C$ has density

$$\rho(r) = \frac{\alpha \sinh(\alpha r)}{\cosh(\alpha R) - 1}. \quad (6.1)$$

- The distance $d(p_i, p_j)$ between the two points p_i and p_j is given by

$$\cosh(d(p_i, p_j)) = \cosh(r_i) \cosh(r_j) - \sinh(r_i) \sinh(r_j) \cos(\theta_i - \theta_j). \quad (6.2)$$

Two nodes $v_i, v_j \in V$ with $i \neq j$ are adjacent iff they are close $d(p_i, p_j) < R$ by. ◀

$\alpha > 1/2$: dispersion
parameter

 $\gamma = 2\alpha + 1$: powerlaw
exponent of degree
distribution

Roughly speaking, the smaller α the more likely are points with small radial components, which are expected to have a high number of neighbors. The parameter hence controls the skewness of the resulting powerlaw degree distribution with an exponent of $\gamma = 2\alpha + 1 > 2$. [200] We assume $\alpha = \mathcal{O}(1)$ since real networks typically exhibit $2 \leq \gamma \leq 4$ (e.g., [109, 202]). Further, while with high probability there exists a giant component of linear size for $\alpha < 1$, networks with $\alpha > 1$ have components of sub-linear size [54]. The parameter C controls the average degree \bar{d} of the graph which is governed as follows: [159]

$$\mathbb{E}[\bar{d}] = \frac{2}{\pi} \left(\frac{\alpha}{\alpha - 1/2} \right)^2 e^{-C/2} (1 + o(1)) \quad (6.3)$$

6.1.4 Hyperbolic Graph Generators

A naïve generator for hyperbolic graphs checks all $\binom{n}{2}$ pairwise distances and emits an edge for each pair of points close enough. On the one hand, such an approach can be implemented with constant memory overhead based on a pseudorandom hash function mapping node ids to coordinates. On the other, it incurs a sequential runtime of $\Theta(n^2)$ and is hence prohibitively expensive for large n . Similarly, while it can be fully parallelized yielding a $\mathcal{O}(1)$ time computation on a EREW-PRAM [187] with $p = \Theta(n^2)$ processors, such a solution requires $\Theta(n^2)$ work and is infeasibly inefficient.

All sub-quadratic algorithms we are aware of rather rely on a two-step approach: For each node $v \in V$, the generators firstly identify a set of candidates $C(v) \subseteq V$ by some geometric means (see below). Edges are then generated by computing only the distances between v and $C(v)$. In order to avoid false negatives, all neighbors $N(v)$ have to be a subset of $C(v)$. Techniques include:

- Looz et al. [223] project all points into the Poincaré disk model which allows neighborhood queries based on Euclidean disks. Candidates are selected using a polar quad-tree. The authors bound the generator's runtime to $\mathcal{O}((n^{3/2} + m) \log n)$ with high probability.

⁴We treat a node v_i and its corresponding point p_i as equivalent and use the terms interchangeably. Similarly, the symbols r_i and θ_i always refer to the radius and azimuth of point p_i .


- Later, Looz et al. improve the runtime significantly by dropping the angular separation of the quad-tree [224]. As sketched in Section 6.1.3, their generator (which is the basis of our work and to which we refer as `NkBAND`⁵) decomposes the hyperbolic plane into $k = \Theta(\log n)$ bands, each covering the radial range $[b_j, b_{j+1})$ where $b_j = (1 - \beta^{j-1})R / (1 - \beta^k)$ for $j \in [k+1]$ and a tuning parameter $\beta \approx 0.9$. In a preprocessing step, the points are randomly scattered over the plane by inserting each point (r, θ) into the appropriate band j , where $b_j \leq r < b_{j+1}$. The points are then sorted by angular coordinates independently for each band.


In order to query the neighbor candidates of a point $p = (r, \theta)$ stored in band i , the algorithm iterates over all bands $i \leq j \leq k$. For each band j , it computes the angular range $A_j = [\theta - \Delta\theta(r, b_j), \theta + \Delta\theta(r, b_j)]$ where the maximal angular distance $\Delta\theta(r, b_j)$ between p and any hypothetical point in band j is given by

$$\Delta\theta(r, b) := \begin{cases} \pi & \text{if } r + b < R \\ \arccos \left[\frac{\cosh(r) \cosh(b) - \cosh(R)}{\sinh(r) \sinh(b)} \right] & \text{otherwise} \end{cases} \quad (6.4)$$

The points within A_j constitute all candidates from band j . As points are sorted by their angles, the bounds of A_j can be identified using two binary searches in time $\mathcal{O}(\log n)$. The authors experimentally find a runtime of $\mathcal{O}(n \log n + m)$.

- Bringmann et al. [68] propose the *Geometric Inhomogenous Random Graphs* model (*GIRG*) and show that $G_{\alpha, C}(n)$ is a special case of *GIRG* which can be generated with their sampling algorithm in expected time $\mathcal{O}(n + m)$. Their sampling method for hyperbolic graphs is similar to the quad-tree approach in the sense that it partitions the space uniformly along the angular axis and exponentially in the radial direction. The resulting cells roughly correspond to leaves in a quad-tree. However, the algorithm does not execute fine-grained neighborhood queries for each node; it rather tests all point pairs of two related cells in a pessimistic and data-oblivious fashion. Despite its expected linear runtime, the algorithm seems to suffer from high constants (Section 6.5). Bläsius et al. provide an implementation⁶ to which we refer as *GIRGS* [49].
- Very recently and independently from this work, S. Lamm proposed the distributed and communication-agnostic generator *RHG* (*PRELIM*) with a partitioning scheme similar to [68], although with different radial limits [206]. Each band is split into disjoint buckets of equal angular size. Their number is chosen such that each cell is expected to contain k points, where $k \approx 4$ is a tuning parameter. *RHG* (*PRELIM*) allows all processing units to compute the points within any bucket independently, eliminating the need of communication. The author shows an expected sequential runtime of $\mathcal{O}(n + m)$, bounds the generation time of the distributed grid structure to $\mathcal{O}(P \log n + n/P)$, where P is the number of processors, and empirically finds a time complexity of $\mathcal{O}\left(\frac{n+m}{P} + P \log n\right)$ for the parallel algorithm.

these constants were reduced later in [HYPERGIRGS](#):  Chapter 8

*Development of [HYPERGEN](#) and [RHG](#) overlapped. The later published [RHG](#) ( Chapter 7) uses improvements proposed here unavailable at time of writing. We hence denote the earlier variant of [RHG](#) as [RHG](#) (*PRELIM*).*

⁵A reference implementation is included in *NetworKit* [316]. <https://networkkit.github.io/>

⁶<https://bitbucket.org/HaiZhong/hyperbolic-embedder/overview>

Algorithm 10: MEMGEN

```

1  $\Delta\theta(a, b) := \pi$  if  $a + b < R$  else  $\arccos\left[\frac{\cosh(a)\cosh(b) - \cosh(R)}{\sinh(a)\sinh(b)}\right]$ 
2  $\text{NOBANDS} \leftarrow \max(2, \lceil \beta R \rceil)$ 
3  $\text{LIMITS} \leftarrow [0, R/2, c + R/2, 2c + R/2, \dots, R - c, R]$  with  $c = R/2/(\text{NOBANDS} - 1)$ 
4 for  $i \in [1, \dots, n]$  do
5    $r \leftarrow$  random radius from  $[0, R]$  with density  $\rho(r) = \alpha \sinh(\alpha r)/(\cosh(\alpha R) - 1)$ 
6    $b \leftarrow$  search band s.t.  $\text{LIMITS}[b] \leq r < \text{LIMITS}[b + 1]$ 
7    $\theta \leftarrow$  next non-decreasing uniformly random polar angle
8   // In case  $\theta + 2\Delta\theta(r, r) > 2\pi$  special treatment is necessary – see text
9    $\text{BANDS}[b].\text{ADDPPOINT}(\text{POINT}(i, (r, (\theta + \Delta\theta(r, r)) \bmod 2\pi)))$ 
10   $b \leftarrow \max(2, b)$ 
11   $\text{req} \leftarrow \text{REQUEST}(i, [r, r + 2\Delta\theta(r, \max(r, \text{LIMITS}[b + 1]))], (r, \theta + \Delta\theta(r, r)))$ 
12   $\text{BANDS}[b].\text{ADDEREQUEST}(\text{req})$ 
13  // Main Phase: Generation of Edges
14  foreach  $u, v \in \text{BANDS}[1].\text{POINTS}$  with  $u < v$  do
15    emit edge  $\{u.\text{ID}, v.\text{ID}\}$ 
16   $\text{REQSTOABOVE} \leftarrow []$ 
17  for  $b \in [2, \dots, \text{NOBANDS}]$  do
18    sort  $\text{BANDS}[b].\text{points}$  by angle
19     $\text{REQSFROMBELOW} \leftarrow \text{sorted}(\text{REQSTOABOVE})$ 
20    initialise empty  $\text{REQSTOABOVE}, \text{CANDIDATES}$ 
21    foreach  $pt \in \text{BANDS}[b].\text{POINTS}$  do
22      remove all requests from  $\text{CANDIDATES}$  ending before  $pt.\theta$ 
23      foreach  $\text{REQ} \in (\text{BANDS}[b].\text{REQS} \cup \text{REQSFROMBELOW})$  with
24         $\text{REQ}.\text{RANGEBEGIN} \geq pt.\theta$  do
25        insert  $\text{REQ}$  into  $\text{CANDIDATES}$  if not existing
26        insert  $\text{REQ}$  into  $\text{REQSTOABOVE}$  with updated range
27      foreach  $\text{REQ} \in \text{CANDIDATES}$  do
28        if  $(\text{REQ}.r, \text{REQ}.\text{ID}) \leq_{\text{lexico}} (pt.r, pt.\text{ID}) \wedge \text{DIST}(pt, \text{REQ}) \leq R$  then
29          emit edge  $\{pt.\text{ID}, \text{req}.\text{ID}\}$ 

```

6.2 MemGen: a Fast Algorithm with Linear Memory Usage

To simplify the description of HYPERGEN and present its main design, we start with a sequential version MEMGEN (Algorithm 10) requiring $\mathcal{O}(n)$ memory. Most arguments regarding the runtime of MEMGEN later translate into the space complexity of HYPERGEN.

Geometrically, MEMGEN employs a band partitioning similar to the one introduced by NKBAND and illustrated in Section 6.1.3. However, we alter their contents and access patterns, and use different radial band limits: all bands except the lowest one have a constant height $x = R/2k$, where $k + 1$ is the number of bands and $x = \Theta(1)$ a tuning parameter (typically $x \in [1, 2]$). Band $1 \leq i \leq k + 1$ covers a radial range of $[l_i, l_{i+1})$ with $l_1 = 0$ and $l_i = [1 + (i - 2)/k] \cdot R/2$ for some $k = \Theta(R)$. It is not necessary to further divide the lowest band since all points with radius $r \leq R/2$ are forming a clique (see Section 6.1.3) and can be handled without vicinity tests.

Band b stores all points contained. For each point p within b or below it, the band additionally maintains a so-called *request* $\text{req}_b(p)$, storing the coordinates of p itself as well as the angular range in which neighbors of p can lie in band b . Such requests effectively reduce random accesses during the candidate selection and carry precomputed values repeatedly required for the distance calculations (see Section 6.4).

In fact, the algorithm chooses a request-centric view and randomly draws the beginnings of each request range, computes its radius-dependent length, and then places a point at its center.⁷ We draw the polar components as sorted random numbers using the online technique detailed in [39] requiring constant time per element. The generation process may yield requests with a range $[a, b]$ with $b > 2\pi$. To take the azimuthal 2π -period of the hyperbolic disk into account, we split such queries into two separate ranges $[0, b - 2\pi]$ and $[a, 2\pi]$ respectively and mark the latter as a copy. Analogously, points with $\theta > 2\pi$ are remapped to $\theta - 2\pi$. After the generation phase, the points are sorted by their polar coordinate.

In the main phase, we iterate over the bands starting from the center for which we simply emit the clique of all nodes contained. For all higher bands, we scan the points and requests in lock-step and keep a separate list of candidates $C(\cdot)$. Since both streams are sorted, we can efficiently update $C(v)$ when moving from one point to the next.

Each time we reach a new unmarked request $\text{req}_j(p)$, we propagate it to the next higher band $j + 1$ by adding $\text{req}_{j+1}(p)$ to the appropriate insertion buffer. Here, it may be again necessary to split a request due to the 2π -periodicity. Further observe that the range of a request may shrink during the propagation. As a consequence, the insertion buffer has to be sorted when switching to band $j + 1$ (cf. Section 6.2.2) before it can be merged with the requests generated in the preprocessing phase. In a last step, we compute the distance between a point and all candidates in order to emit the edges.

The linear time generators we are aware of use discrete buckets along the angular axis to avoid sorting [68, 206]. However, preliminary experiments with MEMGEN suggested that a more involved candidate selection process is faster in practice (especially in the context of vectorization) and does incur only small theoretical penalties (see Theorem 6.7). Thus, we maintain a data structure which keeps active candidates in a continuous array to facilitate vectorization efficiently (see Section 6.4 for details). The array has an arbitrary order allowing to implement deletions as moves of the array's back. The data structure is further augmented with a search tree to find the position of a candidate using its point id as key. We also keep a priority queue with range-ends to quickly find and remove obsolete candidates.

6.2.1 Candidate Selection is at Worst a Constant Approximation

In this section, we establish all necessary facts to show that the candidate selection incurs a non-substantial overhead. In Lemma 6.2, we will see that most points issue only a constant number of requests.

⁷While this is an arbitrary choice for MEMGEN, it will become a crucial ingredient for HYPERGEN.

Subsequently, we derive a high-probability bound on the number of candidates processed for any node in two steps: Observe that a node has to process all requests from nodes below. Lemma 6.3 bounds their number in terms of n and average degree \bar{d} . Further, Lemma 6.4 states that MEMGEN overestimates the probability mass during candidate selection by at most a constant factor. Therefore, the bound on the number of neighbors from below carries over to the number of candidates processed.

Lemma 6.2. The expected number of bands $\mathbb{E}[B_i]$ a random node v_i sends requests to is $\mathbb{E}[B_i] = 1 + [1 - e^{-\alpha R/2}] / [e^{\alpha/2k} - 1] = \mathcal{O}(1)$ where $k + 1 = \Theta(R)$ is the number of bands used by MEMGEN. \blacktriangleleft

Proof. Each point with radius r sends requests to its own band j with $b_j \leq r < b_{j+1}$ as well as to all above. Consequently, the probability of a random point p_i contributing to band j is governed by the mass function $\mu(B_{b_{j+1}}(0))$ as given by Eq. (6.14). Using indicator variables for the reception of a request by band j , we obtain the claimed expectation value:

$$\begin{aligned} \mathbb{E}[B_i] &= \sum_{j=0}^k \mu(B_{b_{j+1}}(0)) = \sum_{j=0}^k e^{\alpha[\frac{R}{2}(1+j/k)-R]} = e^{-\alpha R/2} \sum_{j=0}^k \left(e^{\alpha \frac{R}{2k}}\right)^j \\ &= 1 + \frac{1 - e^{-\alpha R/2}}{e^{\alpha/2k} - 1} \quad \square \end{aligned}$$

Lemma 6.3. Let N_j be the number of neighbors the point $p_j = (r_j, \theta_j)$ has from below, i.e., neighbors with smaller radius. With high probability, there exist $\mathcal{O}(n/\log^2 n)$ points with $N_j = \mathcal{O}(n^{1-\alpha} \bar{d}^\alpha \log(n))$ while the remainder of points with $r_j > R/2$ has $N_j = \mathcal{O}(n^{1-2\alpha} \log^{2\alpha}(n) \bar{d}^{2\alpha})$ neighbors. \blacktriangleleft

Proof. Let X_1, \dots, X_n be indicator variables with $X_i = 1$ if p and p_i are adjacent. Due to radial symmetry we directly obtain the expectation value of X_i conditioned on the radius p_i :

$$\mathbb{E}[X_i \mid r_i = x] = \mathbb{P}[X_i=1 \mid r_i = x] = \begin{cases} 1 & \text{if } x < R - r \\ \Delta\theta(x, r)/\pi & \text{otherwise} \end{cases} \quad (6.5)$$

We remove the conditional using the Law of Total Expectation and Eqs. (6.13) and (6.14):

$$\mathbb{E}[X_i] = \int_0^{R-r} \rho(x) dx + \frac{1}{\pi} \int_{R-r}^r \rho(x) \Delta\theta(x, R) dx \quad (6.6)$$

$$= [e^{-\alpha r} - e^{-\alpha R}] (1 + o(1)) + \frac{1}{\pi} \frac{\alpha}{\alpha - \frac{1}{2}} e^{-\alpha r} \left[e^{(\alpha - \frac{1}{2})(2r-R)} - 1 \right] (1 \pm \mathcal{O}(e^{-r})) \quad (6.7)$$

Fix the radius $r_T = R - \frac{2}{\alpha} \log \log n$ with $R/2 < r_T$ (without loss of generality) and consider three radial regimes:

- We ignore all points of the central clique (i.e., $r \leq R/2$)

- Observe that with high probability there exist $\mathcal{O}(n/\log^2(n))$ points below r_T . Exploiting the monotonicity of Eq. (6.7) in r , we maximize it by setting $r = R/2$, which cancels out the second term. Linearity of the expectation value, substitution of $R = 2\log(n) + C$, and Eq. (6.3) yield $\mathbb{E}[\sum_i X_i] = \mathcal{O}[n(\bar{d}/n)^\alpha]$. Then, Chernoff's bound gives $\sum_i X_i = \mathcal{O}(n^{1-\alpha}\bar{d}^\alpha \log(n))$ with high probability.
- For all points above r_T , set $r = r_T$ yielding $\sum_i X_i = \mathcal{O}(n^{1-2\alpha} \log^{2\alpha}(n)\bar{d}^{2\alpha})$ with high probability analogously. \square

Lemma 6.4. Consider a query point with radius r and a band with boundaries $[a, b]$. MEMGEN's candidate selection overestimates the probability mass of the actual query range by a factor of $OE(b-a, \alpha)$ where $OE(x, \alpha) := \frac{\alpha-1/2}{\alpha} \frac{1-e^{\alpha x}}{1-e^{(\alpha-1/2)x}}$. \blacktriangleleft

Proof. If $r < R - b$, the requesting point covers the band completely which renders the candidate selection process optimal. We now consider $r \geq R - a$ and omit the fringe case of $R - b < r < R - a$ which follows analogously (and by continuity between the two other cases). Then, the probability mass μ_Q of the intersection of the actual query circle $B_R(r)$ with the band $B_b(0) \setminus B_a(0)$ is given by

$$\begin{aligned}
 \mu_Q &:= \mu[(B_b(0) \setminus B_a(0)) \cap B_R(r)] \\
 &= \mu[(B_R(0) \cap B_R(r)) \setminus B_a(0)] - \mu[(B_R(0) \cap B_R(r)) \setminus B_b(0)] \\
 &\stackrel{(6.15)}{=} \frac{2\alpha e^{-\frac{r}{2}}}{\pi(\alpha - \frac{1}{2})} \left[\left(1 + \frac{\alpha - \frac{1}{2}}{\alpha + \frac{1}{2}} e^{-2\alpha b}\right) e^{(\alpha - \frac{1}{2})(b-R)} + \left(1 + \frac{\alpha - \frac{1}{2}}{\alpha + \frac{1}{2}} e^{-2\alpha a}\right) e^{(\alpha - \frac{1}{2})(a-R)} \right] (1 + \varepsilon) \\
 &= \frac{2}{\pi} e^{-\frac{r}{2} - (\alpha - \frac{1}{2})R} \left[\frac{\alpha}{\alpha - \frac{1}{2}} \left(e^{(\alpha - \frac{1}{2})b} - e^{(\alpha - \frac{1}{2})a} \right) + \mathcal{O}\left(e^{-(\alpha - \frac{1}{2})a}\right) \right],
 \end{aligned}$$

where ε substitutes the error term expanded in the last line (see Section 6.1.3, blue cover of a band). MEMGEN overestimates the actual query range at the border and covers the mass μ_H (see Section 6.1.3, yellow cover of a band):

$$\begin{aligned}
 \mu_H &:= \frac{1}{\pi} \Delta\theta(r, a) \int_a^b \rho(y) dy = \frac{1}{\pi} \cdot 2e^{\frac{R-a-r}{2}} (1 + \mathcal{O}(e^{R-a-r})) \cdot \frac{\cosh(\alpha b) - \cosh(\alpha a)}{\cosh(\alpha R) - 1} \\
 &= \frac{2}{\pi} e^{\frac{R-a-r}{2}} (1 + \mathcal{O}(e^{R-a-r})) \cdot \left[e^{\alpha(b-R)} - e^{\alpha(a-R)} \right] (1 \pm o(1)) \\
 &= \frac{2}{\pi} e^{-\frac{r}{2} - (\alpha - \frac{1}{2})R} \left[e^{\alpha b - a/2} - e^{(\alpha - \frac{1}{2})a} \right] \cdot \left(1 \pm \mathcal{O}\left(e^{(1-\alpha)(R-a)-r}\right) \right)
 \end{aligned}$$

The claim follows by the division of both mass functions μ_H/μ_Q . \square

Corollary 6.5. Given a constant band height, i.e., $b - a = \mathcal{O}(1)$, Lemma 6.4 implies a constant overestimation for any $\alpha > 1/2$. In case of $b - a = 1$, we have $OE(1, \alpha) \leq \sqrt{e} \approx 1.64 \forall \alpha > 1/2$. \blacktriangleleft

6.2.2 Nearly Sorted Points/Request Allow for Faster Sorting

MEMGEN's scheme to update the candidate list requires the input streams of requests and points to be increasing in their angular coordinate. Since we are not aware of a technique that directly yields both in an ordered fashion, we have to sort them. Using naïve methods this would amount to $\mathcal{O}(n \log(n))$ time (cf. Lemma 6.2). Since the number $m = n\bar{d}/2$ of edges generated constitutes a lower bound on the time complexity of any generator, this approach is optimal for $\bar{d} = \Omega(\log n)$.

Observe, however, that the points are calculated based on ordered requests and are therefore already nearly sorted. Similarly, requests have to be sorted after being propagated from ordered streams. In both cases, and with high probability, the change of rank of each item is bounded to some $\Delta = o(n)$.⁸ Such a Δ -ordered sequence can be sorted in time $\mathcal{O}(n \log \Delta)$, e.g., using a sliding window coupled with a priority queue of size Δ .

The following Lemma gives a rough bound on the time complexity which suffices to show that MEMGEN is optimal for $\bar{d} = \Omega(\log \log(n))$ with high probability:

Lemma 6.6. Sorting all points initially and requests after their propagation requires $\mathcal{O}(n \min[\log(\bar{d} \log n), \log n])$ time. ◀

Proof. It suffices to bound the claim for requests since every point contributes at least one request and has a shorter lifetime. As stated in the introduction of the Lemma, we can rely on classical sorting in time $\mathcal{O}(n \log n)$ for the case of $\bar{d} = \Omega(\log n)$. Thus assume $\bar{d} = o(\log n)$.

The proof consists then of two steps: We pick a radius r_T , s.t. with high probability there are only $\mathcal{O}(n/\log^2(n))$ points below r_T . Since each point issues at most $\mathcal{O}(\log n)$ requests, we can classically sort their $\mathcal{O}(n/\log(n))$ tokens in time $\mathcal{O}(n)$. For the remaining points, we bound the number of overlapping requests from above and thereby also the maximal change in rank that can occur during sorting.

The number n_T of points below radius r_T is governed by the Binomial distribution $\mathcal{B}(n, B_{r_T}(0))$ with $B_{r_T}(0) = 1/\log^2(n)$. Solving for r_T yields $r_T = R - \frac{2}{\alpha} \log \log n$ and hence $n_T = \mathcal{O}[nB_{r_T}(0)]$ with high probability.

We now tend to the requests above r_T and exploit the two following facts:

- The number of bands above r_T is constant since $r_T/R \rightarrow 1$ as $n \rightarrow \infty$.
- During sorting only those requests that overlap can change their relative position. Therefore, we fix $\theta \in [0, 2\pi)$ and let n_θ be the number of requests that include θ .

To maximize n_θ , assume without loss of generality that all remaining requests lie at radius r_T . Then, n_θ is binomially distributed around its mean $n\mu$ with⁹

$$n\mu = n \frac{\Delta\theta(r_T, r_T)}{\pi} = 2ne^{-\frac{R}{2} + \frac{2}{\alpha} \log \log n} = \mathcal{O}\left(\bar{d} \log^{\frac{2}{\alpha}}(n)\right). \quad (6.8)$$

⁸Split requests and remapped points are sorted separately and merged in linear time.

⁹It can be improved to $\mathcal{O}(\bar{d} \log \log n)$ by replacing the assumption that all requests lay at r_T with an appropriate integral; we omit this non-substantial calculation in favor of simplicity.

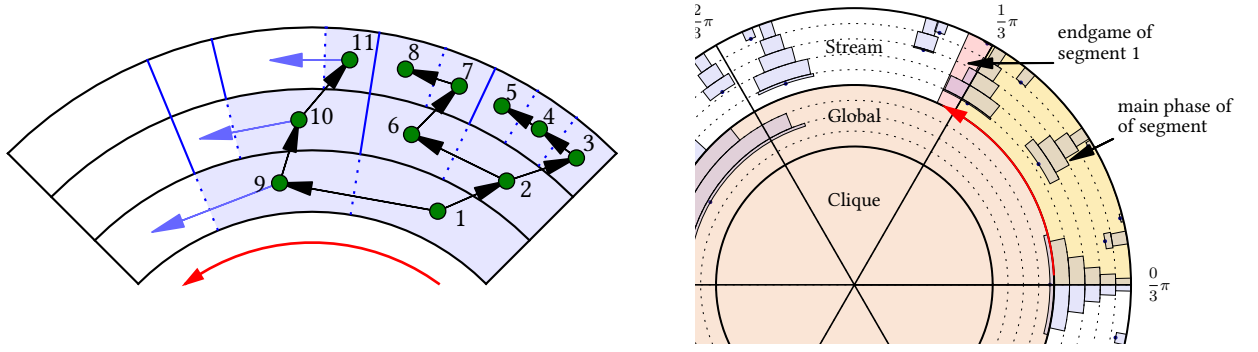


Figure 6.2: **Left:** HYPERGEN streams through each band consuming batches whose size is limited by two factors: either due to a polar limit imposed by the underlying band (solid blue line) or due to the limited number of requests a batch is allowed to have (dotted blue line). We traverse the indicated tree in depth-first order. **Right:** The hyperbolic plane is partitioned along the polar axis into p segments of equal size. Radially, there are two groups: the lower *global bands* which are preprocessed and kept in memory, and the upper *streaming bands*. In the main phase, each execution thread streams through its segment towards increasing polar angles (red arrow). Requests overlapping into the next segment are then completed in the endgame.

With high probability only $\mathcal{O}\left(\bar{d} \log^{\frac{2}{\alpha}}(n)\right)$ requests overlap due to Chernoff's inequality. We thus can sort them in time $\mathcal{O}(n \log(\bar{d} \log n))$. \square

Theorem 6.7. MEMGEN requires $\mathcal{O}(n)$ memory and time $\mathcal{O}(n \log \log n + m)$ whp.. \blacktriangleleft

Proof. The space complexity directly follows from Algorithm 10: each of the n points is stored in exactly one band, yields at most two requests, and requires $\mathcal{O}(1)$ space in the candidate list. During the main phase, there further exists only one insertions buffer at a time to which a point may contribute $\mathcal{O}(1)$ items. \triangle

We bound MEMGEN's time complexity by considering each component individually:

- The preprocessing (until line 11) requires $\mathcal{O}(1)$ time per point.
- Handling of cliques is trivially bounded by $\mathcal{O}(m)$ as every iteration emits an edge.
- The sorting steps (lines 16 and 17) require with high probability a total time of $\mathcal{O}(n \log(\bar{d} \log n)) = \mathcal{O}(n \log \bar{d} + n \log \log n)$ according to Lemma 6.6.¹⁰
- By Lemma 6.3 and Corollary 6.5, the candidate selection requires $\mathcal{O}(n \log \bar{d})$ time.
- All distance calculations require in total $\mathcal{O}(m)$ time since Corollary 6.5 bounds the fraction of computations that do not yield an edge to $\mathcal{O}(1)$. \square

6.3 HyperGen: Reducing MemGen's Memory Footprint

In the analysis of MEMGEN, we repeatedly exploited the facts that requests are generated in increasing angular order and the majority affects only a small fraction of the hyperbolic

¹⁰We consider only the first min-term: in case the second term becomes smaller, the theorem's claim is dominated by the $\mathcal{O}(m)$ where $m = n\bar{d}/2$.

plane. This is also the foundation of HYPERGEN, which strives to additionally reduce the memory requirements of the generator. In order to do so, we do not draw all points globally and insert them into their bands, but rather reverse the scheme.

HYPERGEN first computes how many points go into each band. It is then able to draw points for each band independently. Due to the radial distribution function $\rho(r)$, band i with boundaries $[l_i, l_{i+1})$ carries a probability mass of $\mu_i = \mu[B_{l_{i+1}}(0) \setminus B_{l_i}(0)]$. Consequently, the numbers $N = (n_1, \dots, n_k)$ of points per band with $n = \sum_i n_i$ are governed by a multinomial distribution with μ_i as event probabilities. We sample N and build for each band i a stream $S_i(n_i, s_i)$ that outputs exactly n_i requests with monotonously increasing angles as detailed in Section 6.2. Storing the seed value s_i used to initialize the underlying pseudorandom number generator enables HYPERGEN to replay the stream from the beginning.

Analogous to MEMGEN, each band maintains such a request stream S_i , the current candidates, and a small list of recently produced points. The generator starts with the innermost band $i = 1$ (cf. optimization in Section 6.3.1) and draws a batch of at most c requests from its stream S_i , computes the positions of their corresponding points, and finally sorts the latter by their angle. Let θ_L be the beginning of the last request generated ($\theta_L = 2\pi$ if the batch is empty). We merge the newly generated points with those remaining from the band's last batch, update the set of candidates, and match points against them as described for MEMGEN. Edges produced are pushed into the output stream.

Before we continue in the current band i , we first process all higher bands, hence limiting the amount of requests in memory. HYPERGEN propagates the recently generated requests to the band $i + 1$. Observe that the request of a point (r, θ) is always centered around θ but its range shrinks as it is moved to higher bands. As a direct consequence, the higher range is completely enclosed by the lower one and no future request produced for band i will ever start before θ_L . Therefore, we recurse to band $i + 1$ but limit processing there to points with $\theta < \theta_L$. In effect, HYPERGEN performs a depth-first traversal of the recursion tree illustrated in Figure 6.2 in which every node corresponds to a batch.

Due to the processing limit imposed on higher bands, we make sure they have the same information they would receive in MEMGEN. One subtle difference, however, concerns the fact that MEMGEN splits requests and remaps points overlapping the 2π threshold to take their angular periodicity into account. This is not possible in HYPERGEN since overlaps in outer bands are only detectable quite late in a run.

We resolve this issue by ignoring it at first, i.e., we perform the main computation phase exactly as described above. If there are still pending candidates or points after its completion, we restart the request streams to handle the so-called *endgame*. During endgame, HYPERGEN executes the same algorithm as before but only emits edges for pairs in which either the point or the request originate from the main phase. Therefore, it can be stopped as soon as all such *old* points and candidates have been processed. A single rewind suffices and thus does not affect the asymptotic runtime since a request has a length of at most 2π rad and a point can only be moved π rad in forward direction.

Theorem 6.8. For $c = \mathcal{O}(1)$ HYPERGEN requires $\mathcal{O}([n^{1-\alpha}\bar{d}^\alpha + \log n] \log n)$ memory whp., where \bar{d} is the expected average degree and n the number of nodes. ◀

Proof. Each of the $k = \Theta(\log n)$ bands requires auxiliary data structures of constant size. Regarding the data contained, it again suffices to show the result for requests (cf. proof of Lemma 6.6). The number of points N_C with radius below $R/2$ is governed by a binomial distribution $\mathcal{B}(n, \mu(B_{R/2}(0)))$. Thus, with high probability $N_C = \mathcal{O}(n^{1-\alpha}\bar{d}^\alpha + \log n)$ where the second term ensures concentration for small $(\bar{d}/n)^\alpha$. Each such point contributes requests to $k = \mathcal{O}(\log n)$ bands; multiplication yields the claim.

According to Lemma 6.3 and Corollary 6.5 and for any fixed θ , there are with high probability $\mathcal{O}(n^{1-\alpha}\bar{d}^\alpha \log n)$ points with radius $r \geq R/2$ that have at least one request including θ . By Lemma 6.2, they contribute to $\mathcal{O}(1)$ bands on average and thus are covered by the claim. ◻

Corollary 6.9. In the external memory model with $M = \Omega([1 + n^{1-\alpha}\bar{d}^\alpha] \log n)$, HYPERGEN only triggers I/Os to write out the resulting m edges in $\mathcal{O}(\text{scan}(m))$ I/Os. ◀

6.3.1 Accelerating the Endgame

A runtime/memory trade-off can be implemented to improve the runtime (especially in the context of the parallel variant). Rather than starting the streaming approach introduced above, we compute all bands with radii at most r_G and store them as in MEMGEN in the so-called *global phase*. This allows us to propagate split requests to the streaming bands which in turn allows us to stop the endgame earlier.

Observe that a request of a point (r_G, θ) has a length of at most $2\Delta\theta(r_G, r_G)$. To restrict the endgame to a fraction $1/f$ of the hyperbolic plane, we solve $2\Delta\theta(r_G, r_G) = 2\pi/f$ for r_G . The number $n_G(f)$ of points generated in the global phase, which have to be kept in internal memory, is thus binomially distributed around the mean of

$$\mathbb{E}[n_G(f)] = n\mu(B_{r_G}(0)) = n \left(\frac{\bar{d}f}{2n}\right)^\alpha \left(\frac{\alpha - \frac{1}{2}}{\alpha}\right)^{2\alpha} = \mathcal{O}(n^{1-\alpha}\bar{d}^\alpha f^\alpha). \quad (6.9)$$

6.3.2 Parallelism

Similarly to NKBAND, HYPERGEN can easily be parallelized by decomposing the hyperbolic plane into p segments of equal size along the polar axis. As shown in Figure 6.2, we use a global phase with $f \geq p$ to handle the n_G requests spanning more than one segment. We enqueue a copy of each such request into all segments it affects. For realistic settings, it suffices to execute this phase sequentially; however, parallelism can be applied as in NKBAND's implementation. The number of points in each segment (ν_1, \dots, ν_p) with $n - n_G = \sum_i \nu_i$ is then sampled from a multinomial distribution in which each event is equally likely. Based on this distribution, each band continues independently as described in the original formulation of HYPERGEN. In the endgame, each segment retrieves the seed values of its successor's pseudorandom number generators and replays its streams.

In a distributed scenario the seed values can be computed using a pseudorandom hash function mapping the segment id to a pseudorandom seed value. Further, the initial distribution as well as the fast global phase can be computed repeatedly by each compute node, yielding constant communication.

6.4 Implementation

The prototypical implementation is available at <https://github.com/manpen/hypergen/>.

6.4.1 Adjacency Tests without Trigonometric Functions

In a preliminary study we found that NKBAND's runtime is dominated by trigonometric computations during the calculation of distances between points and their neighbor candidates. We approach this issue by introducing a new precomputing scheme inspired by the usage of the Poincaré disk model in [223]. We project the random points into the unit disk causing additional work per point but simplifying all further distance computation. Thus, the speedup increases with the average degree.

Our implementation applies the transform only to the distance calculations and does not change the candidate selection process. Let $p = (r_p, \theta_p)$ and $q = (r_q, \theta_q)$ be two points in the hyperbolic space and $p' = (\text{cdm}(r_p), \theta_p)$ and $q' = (\text{cdm}(r_q), \theta_q)$ their counterparts in the Poincaré disk model, where $\text{cdm}(r) := [(1 - r^2)/(1 + r^2)]^{1/2}$.

Then p and q are adjacent if

$$R > d(p', q') = \text{acosh} \left(1 + 2 \frac{\|q - p\|^2}{(1 - \|p\|^2)(1 - \|q\|^2)} \right) \quad (6.10)$$

$$\Leftrightarrow \frac{\cosh(R) - 1}{2} > \frac{\|q - p\|^2}{(1 - \|p\|^2)(1 - \|q\|^2)} = \frac{(x_{p'} - x_{q'})^2 + (y_{p'} - y_{q'})^2}{(1 - r_{q'}^2)(1 - r_{q'}^2)} \quad (6.11)$$

$$= ((x_{p'} - x_{q'})^2 + (y_{p'} - y_{q'})^2) \cdot \gamma(r_{p'}) \cdot \gamma(r_{q'}), \quad (6.12)$$

where $x_{p'} = r_{p'} \sin(\theta_p)$ and $y_{p'} = r_{p'} \cos(\theta_p)$ are the Cartesian coordinates of point p' (analogously for q'). We reduce a distance computation to three additions and four multiplications by precomputing $x_{p'}$, $y_{p'}$ and $\gamma(r_{p'}) := 1/(1 - r_{p'}^2)$ for each point. The resulting expression can be vectorized effectively and even allows to partially fuse operations (e.g., FMA instructions).

Our implementation uses explicit vectorization¹¹ to compute distances. For graphs with small average degree, a speedup may be possible by vectorizing computations that are necessary for each point, such as the random number generation and geometric transformations.

6.4.2 Optimising NkGen for Streaming

In addition to the default implementation of NKBAND, we study a variant NKBANDOPT to which we apply the following optimizations:¹²

¹¹based on libVC – SIMD Vector Classes for C++ [198]. <https://github.com/VcDevel/Vc>

¹²NKBAND originally generates an adjacency-list-like internal memory data structure using the *NetworKit*'s GraphBuilder module. This limits the graph sizes and explains NKBAND optimization for smaller

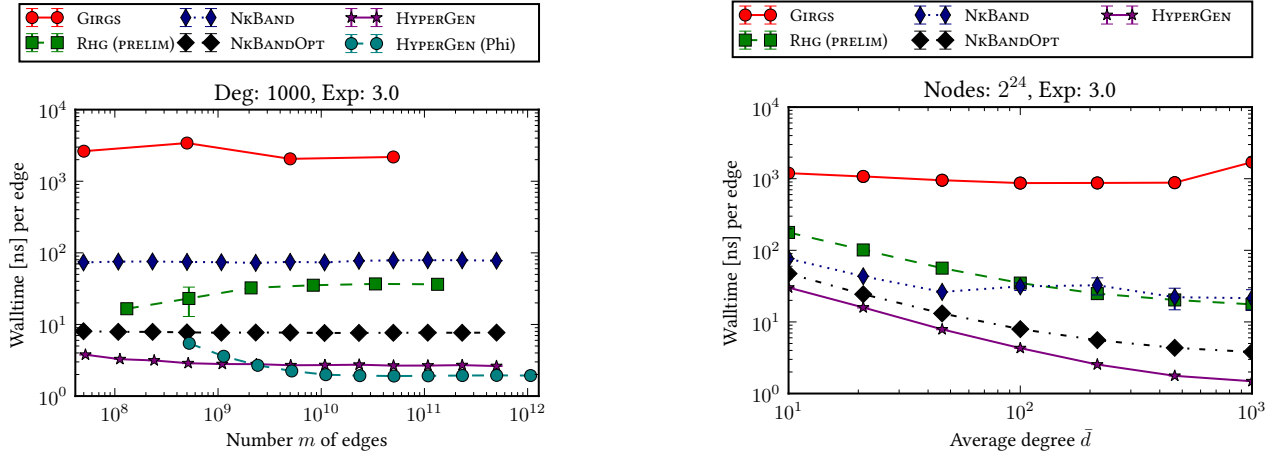


Figure 6.3: Runtime per edge generated for $\alpha = 1$ (powerlaw exponent $\gamma = 3$) as a function of n and \bar{d} . Sample size $S = 5$; with few exceptions, errors bars do not show due to highly concentrated measurements.

- It avoids recalculations similar to Section 6.4.1, but does not rely on the Poincaré transform. In NkBAND’s case all additional data has to be kept in memory amounting to roughly 32 bytes per points. We expect that this increase is only significant for very sparse graphs as *NetworkKit* keeps the whole adjacency list in RAM.
- The number of binary searches as well as their range¹³ is reduced. Further, the amount of data copied is significantly reduced which, in turn, reduces the number of (de-)allocation operations. This optimization roughly compensates the increased footprint due to the precomputations.
- We removed several checks not required for the restricted case of $G_{C,\alpha}(n)$.

HYPERGEN and NkBANDOPT are compared to NkBAND over a wide range of parameters. We observed only acceptable numerical discrepancies for large graphs affecting less than one in 10^5 edges due to different implementations of the distance computation.

6.5 Experimental Evaluation

In this section, we compare six configurations: HYPERGEN on CPU / Xeon Phi (Section 6.3), NkBAND [224], NkBANDOPT (Section 6.4.2), RHG (PRELIM) [206], and GIRGS [49]. They are implemented in C++ and built as release versions with the same compiler. As an exception, HYPERGEN requires a hardware-specific compiler, links against Intel’s TBB MALLOC_PROXY, but otherwise shares the same code with the CPU version.

To fully exploit HYPERGEN’s on-the-fly edge generation, none of the implementations writes the edge list into memory. We rather simulate a very simplistic streaming algorithm which consumes the edge stream and computes a fingerprint by summing

graphs. Further, the removal of the GraphBuilder in this work shifts the implementation’s balance and leads to the large optimization potential demonstrated. Porting the optimizations back to *NetworkKit* showed insignificant changes for typical instances which could be likely solved with an optimized GraphBuilder.

¹³By replacing $\Delta\theta(r, b_i)$ by $\Delta\theta(r, r)$ when searching candidates for (r, θ) in band i with $b_i \leq r < b_{i+1}$.

all node indices contained.¹⁴ This choice enforces that the generators have to compute and forward every edge but does not impose memory restrictions. With the exception of GIRGS, all generators support parallelism and are configured to use all available hardware threads. RHG (PRELIM) employs a multi-process design using MPI allowing several compute nodes, while HYPERGEN, NKBAND and NKBANDOPT use lightweight threads based on OpenMP. The runtime benchmarks use the following systems:

- Indicated by (Phi): Intel Xeon Phi 5120D (60 cores, 240 threads, 1.05GHz), 8 GB GDDR5 RAM Linux 2.6.38, ICC 17.0.0, Intel TBB `MALLOC_PROXY`
- Otherwise: Intel Xeon CPU E5-2630 v3 (8 cores, 16 threads, 2.40GHz) with AVX2/SSE4.2 support for 4-way double-precision vectorization, 64 GB 2133 MHz RAM, Linux 4.8.1, GCC 6.2.1, VC (8. Dec. 2016), MPICH 3.2-7

The number of repetitions per data point (with different random seeds) is denoted by S . All plots show the median of repeated measurements and error bars corresponding to the unbiased estimation of the standard deviation. Due to its large runtime GIRGS typically only includes one measurement per data point.

6.5.1 Runtime

We study the generators' runtimes for a wide range of graph sizes. For each run, we fix the number of nodes $10^5 \leq n \leq 10^9$ as well as the average target degree $\bar{d} \in \{10, 1000\}$, which we consider as lower and upper limits of realistic inputs [28, 216, 238, 255]. In order to achieve compatible results, all implementations use values of R derived with NKBAND's `GETTARGETRADIUS`-method. In case of HYPERGEN, we use two segments per thread to balance load for large average degrees. For RHG (PRELIM) we chose an expected bucket size of four which resulted in the best performance in preliminary tests.

As shown in Section 6.5 and Figure 6.6 (Appendix) and Table 6.1 (Appendix), HYPERGEN is consistently the fastest generator, followed by NKBANDOPT which outperforms NKBAND. GIRGS is always the slowest. If we assume perfect parallelizability and divide GIRGS's wall-time by the number of cores, it is on par with NKBAND for small degrees but remains up to one order of magnitude slower for $\bar{d} = 1000$. For $\bar{d} = 10$ NKBAND outperforms RHG (PRELIM), while for $\bar{d} = 1000$ and $\alpha = 1$ the opposite is true.

All generators but HYPERGEN (Phi) exhibit a near constant computation time per edge for large n . The improvements of HYPERGEN (Phi) towards larger n can be attributed to the high number of threads ($p = 240$) which incur more overhead compared to runs performed on a CPU. This overhead is amortized only for high values of n .

Based on Figure 6.6 (Appendix), we measure a speedup of 4.0 for $\bar{d} = 10$ and 29.6 for $\bar{d} = 1000$ when comparing HYPERGEN to NKBAND for $n \geq 10^8$ and $\alpha = 1$. Similar results for smaller n are included in Table 6.1 (Appendix). On (Phi), HYPERGEN is 2.3 times faster ($\bar{d} = 10$) compared to the execution on the more modern CPU-based

¹⁴We removed the appropriate memory allocations and accesses from NKBAND, RHG (PRELIM), and GIRGS, and added the streaming simulation. The patches are included in our repository.

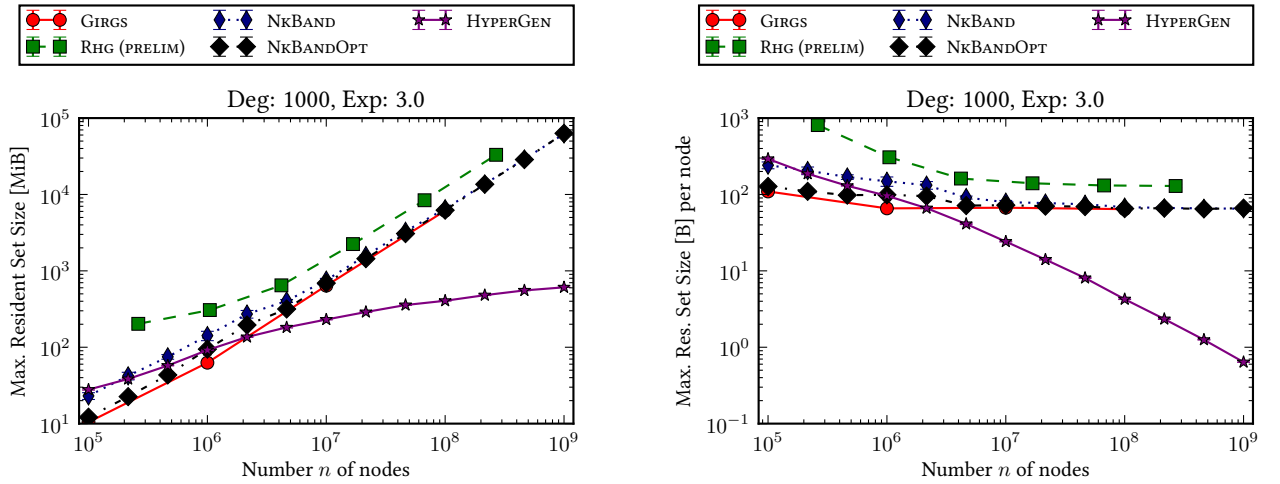


Figure 6.4: Maximal memory allocated during execution as measured by GNU/Linux `time` for $\alpha = 1$.

reference system. The speedup reduces to 1.2 for $\bar{d} = 1000$ which seems to be caused by a smaller cache per thread.

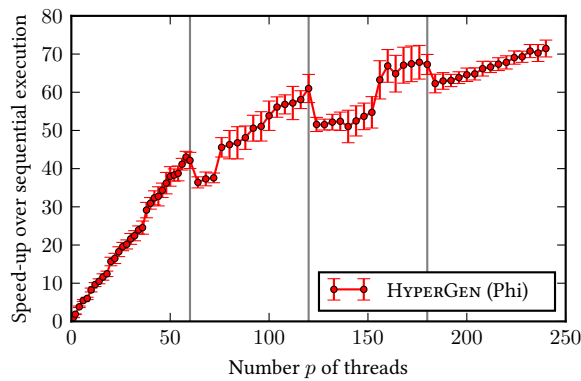
When using HYPERGEN to test a multi-pass streaming algorithm, it is virtually always faster to repeatedly regenerate the graph than to buffer it in external memory.

6.5.2 Memory Consumption

The memory consumption is measured for the same parameter settings as above. We consider the maximal resident set size (i.e., the peak allocation of the generator) as reported by the operating system. While all implementations seem to have potential for further savings, figures 6.4 and 6.7 (Appendix) show a clear trend: With the exception of HYPERGEN, all generators seem to converge to a linear growth for large n requiring ≈ 80 byte per node. RHG (PRELIM) exhibits higher constants which may be partially caused by overheads due to its MPI architecture spawning independent processes rather than lightweight threads and preventing cheap shared-memory utilization.

Consistent with our analysis, HYPERGEN exhibits a sub-linear footprint rendering it orders of magnitude cheaper for large n . As the number n of nodes increases (and hence R for fixed \bar{d}), more points lie in the outer bands. Thus, a smaller fraction of points has to be handled (and stored) during the global phase. For the same reason, the memory footprint decreases with increasing α . To support Theorem 6.8 and the analysis in Section 6.3.1, we carried out additional runs up to $n = 10^{11}$ whose memory footprint is well within the noise observed for $n = 10^8$. We do not include measurements for (Phi) since the memory allocation scheme adopted for the high number of threads does not yield meaningful set sizes.

Figure 6.5: Strong scaling of HYPERGEN on (Phi) for a graph with $n = 10^8$ and $\bar{d} = 10$. $S = 8$. Each vertical division marks a new level of HyperThreading.



6.5.3 Scalability

We measure HYPERGEN’s scalability using strong scaling experiments on (Phi). This processor features 60 physical cores each offering four virtual threads (HyperThreading). While fixing the graph instance to $n = 10^8$ and $\bar{d} = 10$, we record the runtime for an increasing number p of threads. As illustrated in Section 6.5.3, the implementation exhibits a nearly linear speedup of 43.0 ± 1.5 when utilizing $p = 58$ threads. Surpassing this point, the computational power provided by the hardware does not scale linearly. Thus, the additional speedup is less pronounced peaking at 71.4 ± 6 for $p = 240$.

Acknowledgments

The author thanks Ulrich Meyer, Kamil René König, Moritz von Looz and Alexander Schickedanz for valuable discussions and suggestions, Sebastian Lamm for providing the code and support for RHG (PRELIM), Ivan Kisel and Egor Ovcharenko for their help with the Xeon Phi, as well as the anonymous reviewers for their insightful comments.

Appendix 6.A Definitions, Useful Identities and Approximations

6.A.1 Hyperbolic Functions

$$\begin{aligned} \sinh(x) &:= (e^x - e^{-x})/2 & \operatorname{asinh}(y) = x &\Rightarrow \sinh(x) = y \\ \cosh(x) &:= (e^x + e^{-x})/2 & \operatorname{acosh}(y) = x &\Rightarrow \cosh(x) = y \end{aligned}$$

6.A.2 Definitions Related To Geometry

$$\rho(r) := \alpha \sinh(\alpha r) / \cosh(\alpha R) \quad \text{radial density, cf. Eq. 6.1}$$

$$\mu(B_r(0)) := \int_0^r \rho(x) dx = \frac{\cosh(\alpha x) - 1}{\cosh(\alpha R)} \quad \text{radial cdf}$$

$$\Delta\theta(r, b) := \begin{cases} \pi & \text{if } r + b < R \\ \arccos \left[\frac{\cosh(r) \cosh(b) - \cosh(R)}{\sinh(r) \sinh(b)} \right] & \text{otherwise} \end{cases} \quad \text{cf Eq. (6.4)}$$

6.A.3 Approximations

Gugelmann et al. derived the following approximations¹⁵ [159]:

$$\Delta\theta(r, b) = \begin{cases} \pi & \text{if } r + b < R \\ 2e^{\frac{R-r-y}{2}} (1 + \Theta(e^{R-r-y})) & \text{if } r + b \geq R \end{cases} \quad (6.13)$$

$$\mu(B_r(0)) = \int_0^r \rho(x) dx = \frac{\cosh(\alpha r)}{\cosh(\alpha R) - 1} = e^{\alpha(r-R)} (1 + o(1)) \quad (6.14)$$

$$\begin{aligned} \mu[(B_R(r) \cap B_R(0)) \setminus B_x(0)] &= \frac{2}{\pi} \frac{\alpha e^{-r/2}}{\alpha - \frac{1}{2}} \cdot \\ &\begin{cases} 1 \pm \mathcal{O}\left(e^{-(\alpha - \frac{1}{2})r} + e^{-r}\right) & \text{if } x < R - r \\ \left[1 - \left(1 + \frac{\alpha - \frac{1}{2}}{\alpha + \frac{1}{2}} e^{-2\alpha x}\right) e^{-(\alpha - \frac{1}{2})(R-x)}\right] (1 \pm \mathcal{O}\left(e^{-r} + e^{-r - (\alpha - \frac{3}{2})(R-x)}\right)) & \text{if } x \geq R - r \end{cases} \end{aligned} \quad (6.15)$$

¹⁵We drop the $(1 + \mathcal{O}(\cdot))$ error terms in our calculations without further notice if they are either irrelevant or dominated by other simplifications made

Appendix 6.B Additional Experimental Results

$n=2^{26}, \bar{d}=10, \alpha=0.55, R=39.2$			in total			per edge		relative to HYPERGEN		
Algo	S	Degree	Comp. [10^8]	RSS [GB]	Time [s]	Comp.	Time [ns]	Comp.	RSS	Time
HYPERGEN	6	10.3 ± 0.4	7.5 ± 0.2	0.0 ± 0.0	11.8 ± 0.1	2.1 ± 0.1	34.0 ± 1.4	1	1	1
NkBAND	6	9.8 ± 0.7	5.6 ± 0.3	4.6 ± 0.3	57.1 ± 3.0	1.7 ± 0.2	173 ± 22	0.8 ± 0.1	290 ± 22	4.8 ± 0.3
NkBANDOPT	6	9.6 ± 0.4	5.3 ± 0.2	4.1 ± 0.0	36.0 ± 0.3	1.7 ± 0.1	111.7 ± 6.1	0.7 ± 0.0	263.5 ± 3.2	3.1 ± 0.0
RHG (PRELIM)	4	8.3 ± 0.1	7.9 ± 0.0	6.7 ± 0.6	110.0 ± 1.0	2.8 ± 0.0	395.8 ± 6.7	1.1 ± 0.0	428 ± 39	9.3 ± 0.1
GIRGS	3	10.0 ± 0.0	18.1 ± 0.0	3.9 ± 0.0	884.3 ± 0.8	5.4 ± 0.0	2635.5 ± 2.8	2.4 ± 0.1	248.1 ± 2.2	75.0 ± 0.5

$n=2^{26}, \bar{d}=10, \alpha=1.00, R=33.3$			in total			per edge		relative to HYPERGEN		
Algo	S	Degree	Comp. [10^8]	RSS [GB]	Time [s]	Comp.	Time [ns]	Comp.	RSS	Time
HYPERGEN	5	9.7 ± 0.0	7.0 ± 0.0	0.0 ± 0.0	12.9 ± 0.1	2.1 ± 0.0	39.6 ± 0.3	1	1	1
NkBAND	5	10.0 ± 0.0	5.5 ± 0.0	4.1 ± 0.0	54.9 ± 0.5	1.6 ± 0.0	163.5 ± 1.6	0.8 ± 0.0	602 ± 13	4.3 ± 0.1
NkBANDOPT	5	10.0 ± 0.0	5.2 ± 0.0	4.1 ± 0.0	34.4 ± 0.2	1.6 ± 0.0	102.3 ± 0.8	0.8 ± 0.0	596 ± 12	2.7 ± 0.0
RHG (PRELIM)	5	10.0 ± 0.0	8.1 ± 0.0	7.5 ± 0.5	120.5 ± 0.4	2.4 ± 0.0	359.2 ± 1.2	1.2 ± 0.0	1100 ± 99	9.4 ± 0.1
GIRGS	3	10.0 ± 0.0	16.6 ± 0.0	3.9 ± 0.0	819.8 ± 7.1	5.0 ± 0.0	2443 ± 21	2.4 ± 0.0	566 ± 11	63.6 ± 1.0

$n=2^{26}, \bar{d}=1000, \alpha=0.55, R=29.5$			in total			per edge		relative to HYPERGEN		
Algo	S	Degree	Comp. [10^8]	RSS [GB]	Time [s]	Comp.	Time [ns]	Comp.	RSS	Time
HYPERGEN	5	1052.2 ± 1.6	622.8 ± 1.2	0.2 ± 0.0	86.9 ± 0.4	1.8 ± 0.0	2.5 ± 0.0	1	1	1
NkBAND	5	994.4 ± 3.3	456.2 ± 1.3	6.4 ± 0.3	955.5 ± 4.9	1.4 ± 0.0	28.6 ± 0.2	0.7 ± 0.0	27.2 ± 1.3	11.0 ± 0.1
NkBANDOPT	5	991 ± 19	441.6 ± 7.8	4.2 ± 0.0	299.5 ± 5.1	1.3 ± 0.0	9.0 ± 0.3	0.7 ± 0.0	17.6 ± 0.3	3.4 ± 0.1
RHG (PRELIM)	5	889.1 ± 2.2	426.0 ± 1.1	23.9 ± 2.4	2205 ± 64	1.4 ± 0.0	73.9 ± 2.3	0.7 ± 0.0	101 ± 11	25.4 ± 0.9
GIRGS	1	1000.0	1160.6	3.8	55756.0	3.5	1661.6	1.9 ± 0.0	16.2 ± 0.1	641.5 ± 2.8

$n=2^{26}, \bar{d}=1000, \alpha=1.00, R=24.1$			in total			per edge		relative to HYPERGEN		
Algo	S	Degree	Comp. [10^8]	RSS [GB]	Time [s]	Comp.	Time [ns]	Comp.	RSS	Time
HYPERGEN	5	1015.8 ± 1.3	616.2 ± 0.7	0.1 ± 0.0	84.3 ± 0.5	1.8 ± 0.0	2.5 ± 0.0	1	1	1
NkBAND	5	999.9 ± 0.6	443.3 ± 0.3	4.4 ± 0.1	1878 ± 468	1.3 ± 0.0	56 ± 14	0.7 ± 0.0	43.7 ± 2.4	22.3 ± 5.7
NkBANDOPT	5	999.7 ± 0.4	428.5 ± 0.2	4.2 ± 0.0	261.1 ± 6.7	1.3 ± 0.0	7.8 ± 0.2	0.7 ± 0.0	41.6 ± 1.1	3.1 ± 0.1
RHG (PRELIM)	5	999.1 ± 0.0	410.5 ± 0.0	8.1 ± 0.2	1234.8 ± 5.8	1.2 ± 0.0	36.8 ± 0.2	0.7 ± 0.0	79.8 ± 3.7	14.6 ± 0.2
GIRGS †	1				$\geq 10^5$					≥ 1150

Table 6.1: Comparison of generators for $n = 2^{26}$, $\alpha \in \{0.55, 1\}$, and $\bar{d} \in \{10, 1000\}$. *Comp* refers to the number of distance computations between two points. It does not include node pairs that could be ruled out earlier (e.g., by comparing indices or radii). For HYPERGEN the value is higher due to vectorization which often prevents such early discarding. *RSS* is the maximal resident set size (i.e., peak memory allocation) as reported by the operating system. In case of RHG (PRELIM) it is the sum of RSS of all MPI processes yielding a higher overhead. GIRGS is a purely sequential implementation and includes fewer data points due to the high runtime. We report the standard deviation of the S measurements as uncertainty and apply statistical error propagation. † Experiment was canceled after a runtime of 10^5 s.

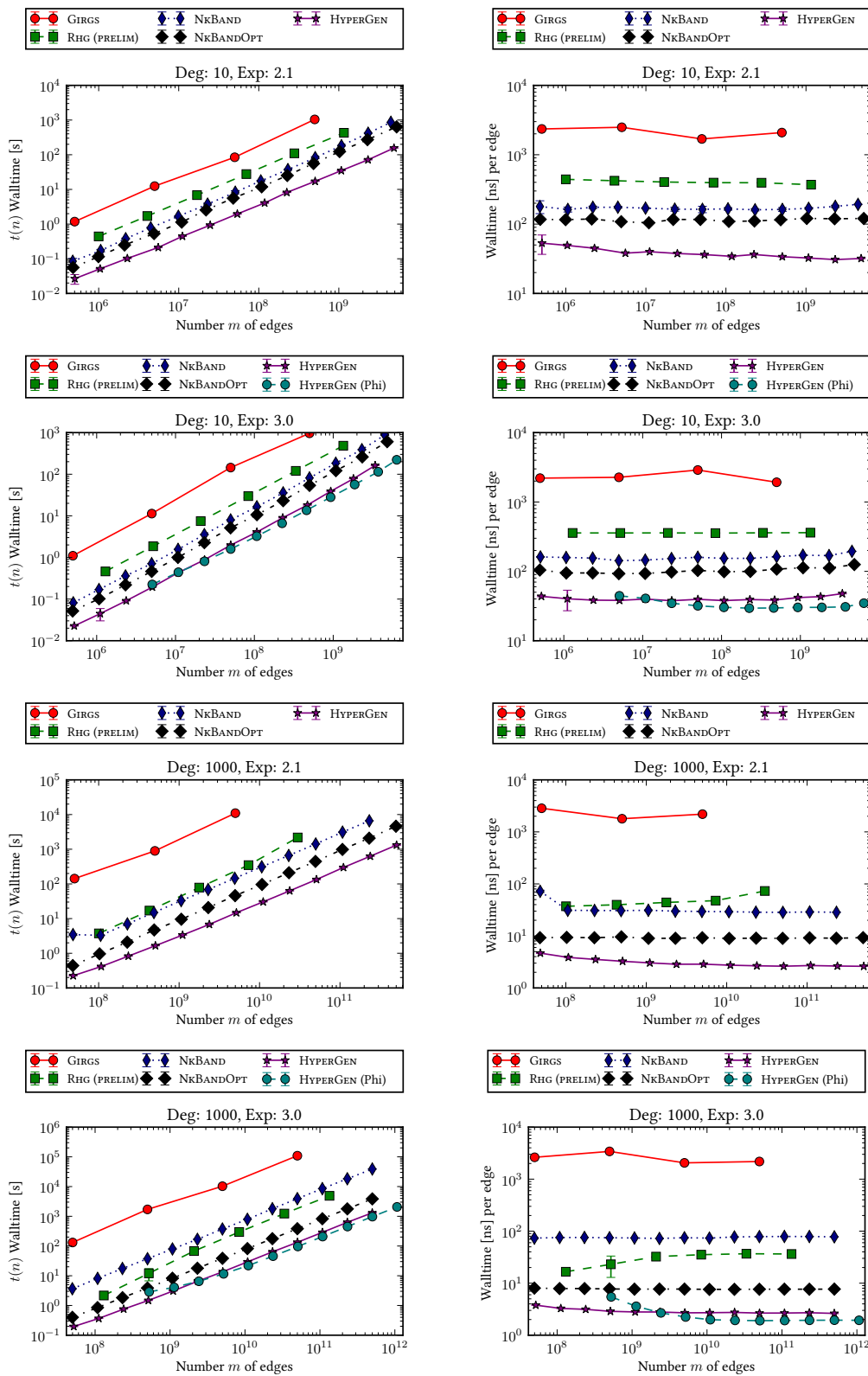


Figure 6.6: Runtime of generators as function of the number n of nodes.

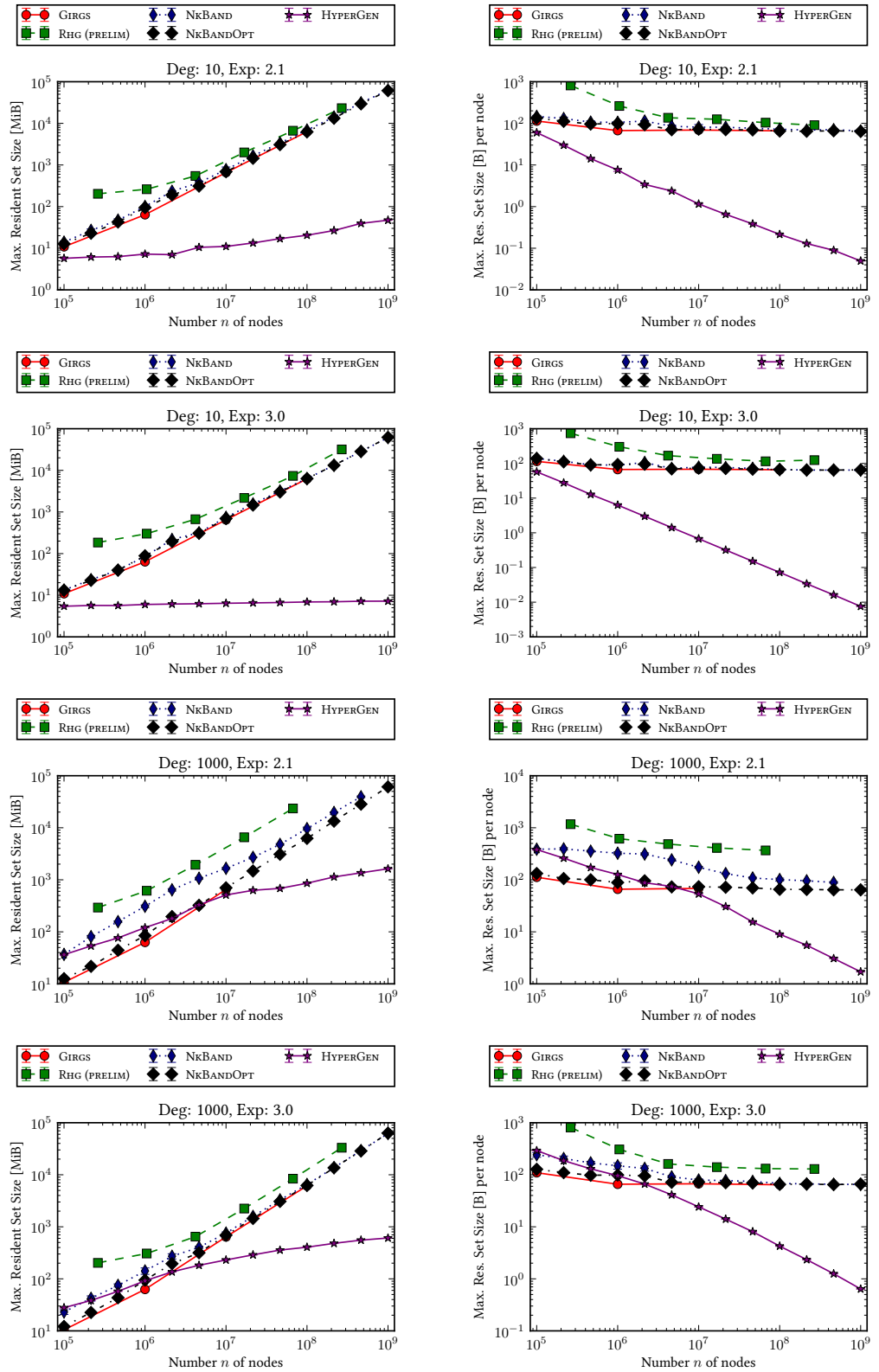


Figure 6.7: Max. memory allocation of generators as function of the number n of nodes.

Communication-free Massively Distributed Graph Generation

joint work with

D. Funke, S. Lamm, U. Meyer, P. Sanders, C. Schulz, D. Strash, and M. v. Looz

Analyzing massive complex networks yields promising insights about our everyday lives. Building scalable algorithms to do so is a challenging task that requires a careful analysis and an extensive evaluation. However, engineering such algorithms is often hindered by the scarcity of publicly available datasets.

Network generators serve as a tool to alleviate this problem by providing synthetic instances with controllable parameters. However, many network generators fail to provide instances on a massive scale due to their sequential nature or resource constraints. Additionally, truly scalable network generators are few and often limited in their realism.

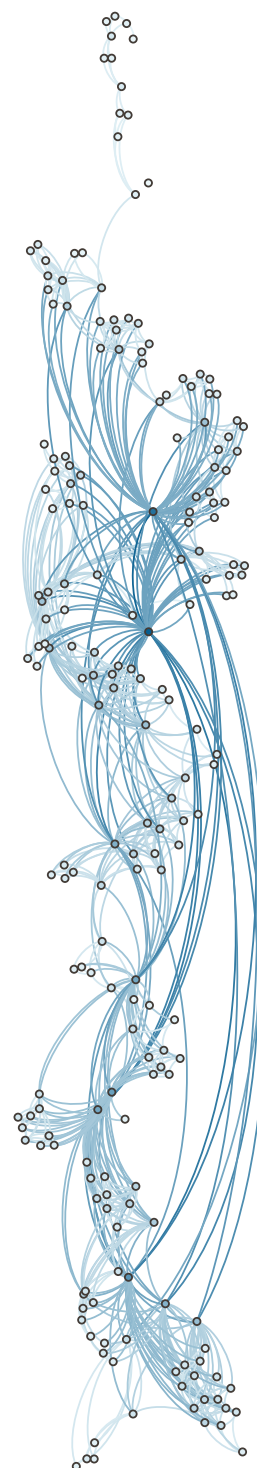
In this work, we present novel generators for a variety of network models that are frequently used as benchmarks. By making use of pseudorandomization and divide-and-conquer schemes, our generators follow a communication-free paradigm. The resulting generators are thus pleasingly parallel and have a near optimal scaling behavior. This allows us to generate instances of up to 2^{43} vertices and 2^{47} edges in less than 22 minutes on 32 768 cores. Therefore, our generators allow new graph families to be used on an unprecedented scale.

This chapter is based on the peer-reviewed journal article [138]:

- [138] D. Funke, S. Lamm, U. Meyer, M. Penschuck, P. Sanders, C. Schulz, D. Strash, and M. v. Looz. Communication-free massively distributed graph generation. *J. Parallel Distributed Comput.*, 131:200–217, 2019. doi:10.1016/j.jpdc.2019.03.011 .

My contribution

I contributed sRHG as the main author and developer, as well as to the analysis of RHG. I do *not* claim authorship for the remaining generators previously published as [139].



Threshold Random
Hyperbolic Graph with
 $n = 200$, $\bar{d} = 8$, $\alpha = 1$

7.1 Introduction

Complex networks are prevalent in every aspect of our lives: from technological networks to biological systems like the human brain. These networks are composed of billions of entities that give rise to emerging properties and structures. Analyzing these structures aids us in gaining new insights about our surroundings. In order to find these patterns, massive amounts of data have to be acquired and processed. Designing and evaluating algorithms to handle these datasets is a crucial task on the road to understanding the underlying systems. However, real-world datasets are often scarce or are too small to reflect future requirement.

Network generators solve this problem to some extent. They provide synthetic instances based on random network models. These models are able to accurately describe a wide variety of different real-world scenarios: from ad-hoc wireless networks to protein-protein interactions. [254, 96] A substantial amount of work has been contributed to understanding the properties and behavior of these models. In theory, network generators allow us to build instances of arbitrary size with controllable parameters. This makes them an indispensable tool for the systematic evaluation of algorithms on a massive scale. For example, the well known Graph 500 benchmark¹, uses the *R-MAT* graph generator [87] to build instances of up to 2^{42} vertices and 2^{46} edges.

Even though generators like *R-MAT* scale well, the generated instances are limited to a specific family of graphs. [87] Many other important network models still fall short when it comes to offering a scalable network generator and in turn to make them a viable replacement for *R-MAT*. These shortcomings can often be attributed to the apparently sequential nature of the underlying model or prohibitive hardware requirements.

Our Contribution

In this work we introduce a set of novel network generators that focus on scalability. We achieve this by using a communication-free paradigm [294], i.e., our generators require no communication between processing entities (PEs). An implementation is available as the *KaGen* library at <https://github.com/sebalamm/KaGen>.

Each PE is assigned a disjoint set of local vertices. It then is responsible for generating all incident edges for this set of vertices. This is a common setting in distributed computation. [225] The models that we target are the classic $G(n, m)$ and $G(n, p)$ models of *Erdős-Rényi* [121] and *Gilbert* [148], respectively, and different spatial network models including *Random Geometric Graphs (RGG)* [188], *Random Hyperbolic Graph (RHG)* [200] and *Random Delaunay Triangulation (RDTs)*. The *KaGen* library also supports *Barabási-Albert* [32] using the algorithm by Sanders and Schulz. [294]

For each new generator, we provide bounds for their parallel (and sequential) running times. A key-component of our algorithms is the combination of pseudorandomization and divide-and-conquer strategies. These components enable us to perform efficient recomputations in a distributed setting without the need for communication.

¹<https://graph500.org>

each PE uniquely owns
nodes and compute their
incident edges

supported models:
 $G(n, m)$, $G(n, p)$, RGG,
RHG, RDT, BA

To highlight the practical impact of our generators, we also present an extensive experimental evaluation. First, we show that our generators rival the current state-of-the-art in terms of sequential and/or parallel running time. Second, we are able to show that our generators have near optimal scaling behavior in terms of weak scaling (and strong scaling). Finally, our experiments show that we are able to produce instances of up to 2^{43} vertices and 2^{47} edges in less than 22 minutes. These instances are in the same order of magnitude as those generated by *R-MAT* for the Graph 500 benchmark. Hence, our generators enable the underlying network models to be used in massively distributed settings.

7.2 Preliminaries

We define a *graph* (network) as a pair $G = (V, E)$. The set $V = \{0, \dots, n-1\}$ ($|V| = n$) denotes the vertices of G . For a directed graph $E \subseteq V \times V$ ($|E| = m$) is the set of edges consisting of ordered pairs of vertices. Likewise, in an undirected graph E is a set of unordered pairs of vertices. Two vertices that are endpoints of an edge $e = \{u, v\}$ are called *adjacent*. Edges in directed graphs are ordered tuples $e = (u, v)$. An edge $(u, u) \in E$ is called a *self-loop*. If not mentioned otherwise, we only consider simple graphs that contain no self-loops or parallel edges.

The set of *neighbors* for any vertex $v \in V$ is defined as $N(v) = \{u \in V \mid \{u, v\} \in E\}$. For an undirected graph, we define the *degree* of a vertex $v \in V$ as $d(v) = \Delta(v) = |N(v)|$. In the directed case, we have to distinguish between the *indegree* and *outdegree* of a vertex.

We denote that a random variable X is distributed according to a probability distribution \mathcal{P} with parameters p_1, \dots, p_i as $X \sim \mathcal{P}(p_1, \dots, p_i)$. The probability mass function of a random variable X is denoted as $\mu(X)$.

7.2.1 Network Models

7.2.1.1 Erdős-Rényi Graphs

The Erdős-Rényi (*ER*) model was the first model for generating random graphs and supports both directed and undirected graphs. For both cases, we are interested in graph instances without parallel edges. We now briefly introduce the two closely related variants of the model.

The first version, proposed by Gilbert [148], is denoted as the $G(n, p)$ model. Here, each of the $n(n-1)/2$ possible edges of an n -node graph is independently sampled with probability $0 < p < 1$ (Bernoulli sampling of the edges).

The second version, proposed by Erdős and Rényi [121], is denoted as the $G(n, m)$ model. In the $G(n, m)$ model, we chose a graph uniformly at random from the set of all possible graphs which have n vertices and m edges.

For sake of brevity, we only revisit the generation of graphs in the $G(n, m)$ model. However, all of our algorithms can easily be transferred to the $G(n, p)$ model.

7.2.1.2 Random Geometric Graphs

Random geometric graphs (RGGs) are undirected spatial networks where we place n vertices uniformly at random in a d -dimensional unit cube $[0, 1]^d$. Two vertices $p, q \in V$ are connected by an edge iff their d -dimensional Euclidean distance

$$\text{dist}(p, q) = \left(\sum_{i=1}^d (p_i - q_i)^2 \right)^{1/2}$$

is within a given threshold radius r . Thus, the RGG model can be fully described using the two parameters n and r . Note that the expected degree of any vertex that does not lie on the border, i.e., whose neighborhood sphere is completely contained within the unit cube, is $\bar{d}(v) = \pi^{\frac{d}{2}} r^d / \Gamma(\frac{d}{2} + 1)$. [270] In our work we focus on two- and three-dimensional random geometric graphs, as these are very common in real-world scenarios. [277]

7.2.1.3 Random Hyperbolic Graphs

Random hyperbolic graphs (RHGs) are undirected spatial networks generated in the hyperbolic plane with negative curvature. To generate a RHG, n points are randomly placed on a disk with radius

$$R = 2 \log n + C, \quad (7.1)$$

where C controls the average degree \bar{d} with

$$\bar{d} = \frac{2}{\pi} \left[\frac{\alpha}{\alpha - 1/2} \right]^2 e^{-C/2} (1 + o(1)) \quad (7.2)$$

with high probability. [200, 159] Additionally, the model features a dispersion factor $\alpha > 1/2$ affecting concentration of points near the center of the disk.

Each vertex q corresponds to a point with a polar coordinate θ_q and a radial coordinate r_q . Its angle θ_q is sampled uniformly at random from the interval $[0, 2\pi)$, while its radius r_q is chosen according to the probability density function

$$f(r) = \alpha \frac{\sinh(\alpha r)}{\cosh(\alpha R) - 1}. \quad (7.3)$$

Krioukov et al. [200] and Gugelmann et al. [159] show that for $\alpha > 1/2$ the degree distribution in the *threshold model* follows a powerlaw distribution with exponent $\gamma = 1 + 2\alpha$. In this RHG variant, two vertices p, q are connected iff their hyperbolic distance

$$\text{dist}_H(p, q) = \text{acosh}(\cosh r_p \cosh r_q - \sinh r_p \sinh r_q \cos(\theta_p - \theta_q)) \quad (7.4)$$

is below the threshold R . Therefore, the neighborhood of a vertex consists of all the vertices that are within the hyperbolic circle of radius R around it.

7.2.1.4 Random Delaunay Graphs (RDT)

A two-dimensional Delaunay graph is a planar graph whose vertices represent points in the plane. Its edges form a triangulation of this point set, i.e., they partition the convex hull of the point set into triangles. Furthermore, the circumcircle of each triangle must not contain other vertices in its interior. This concept can be generalized for d -dimensional Euclidean space. [115] In particular, for $d = 3$ we get a tetrahedralization, i.e., a decomposition of the space into tetrahedra whose circumsphere may not contain vertices in their interior.

In this paper, we are concerned with Delaunay graphs defined by points sampled uniformly at random from the d -dimensional unit cube $[0, 1]^d$ for $d \in \{2, 3\}$. We view this as a good model for meshes as they are frequently used in scientific computing. Indeed, these simulations frequently use *periodic boundary conditions*, in order to make small simulations representative for a large simulated system (e.g., [322]). This can also be viewed as replacing the infinite Euclidean space by a d -dimensional torus. We adopt these periodic boundary conditions, i.e., we implicitly compute the Delaunay-Triangulation of a point set where for every point \mathbf{x} in the unit cube, also the points $\mathbf{x} + \mathbf{o}$ with $\mathbf{o} \in \{-1, 0, 1\}^d$ are in the point set. Two points in the unit cube are connected in the output, if any of their copies are connected. For a scalable distributed graph generator, periodic boundary conditions have the advantage that we avoid the need to compute some very long edges that appear at the convex hull of random point set.

7.2.2 Sampling Algorithms

Most of our generators require sampling (with/without replacement) of n elements from a (finite) universe N . Sequentially, both of these problems can be solved in expected time $\mathcal{O}(n)$. [334] These bounds still hold true, even if a sorted sample has to be generated. [334, 39] However, most of these algorithms are hard to apply in a distributed setting since they are inherently sequential.

Recently, Sanders et al. [292] proposed a set of simple divide-and-conquer algorithms that allow sampling n elements on P PEs. Their algorithms follow the observation that by splitting the current universe into equal-sized subsets, the number of samples in each subset follows a hypergeometric distribution. Based on this observation, they develop a divide-and-conquer algorithm to determine the number of samples for each PE. In particular, each PE first determines its local interval of the input universe and then recursively generates a set of hypergeometric random variates. At each level of the recursion, it follows the remaining subset of the universe that contains its local interval. Hypergeometric random variates are synchronized without the need for communication by making use of pseudorandomization via high-quality hash functions.

To be more specific, for each subtree of the recursion, a unique seed value is computed (independent of the rank of the PE). Afterwards, a hash value for this seed is computed and used to initialize the pseudorandom number generator (PRNG) for the random variates. Therefore, PEs that follow the same recursion subtrees generate the same random variates, while variates in different subtrees are independent of each

other. Once the remaining subset is smaller than a given threshold, a linear time sequential algorithm [334] is used to determine the local samples. They continue to show that their algorithm runs in time $\mathcal{O}(n/P + \log P)$ with high probability² (whp.) if the maximum universe size per PE is $\mathcal{O}(N/P)$. [292] Additionally, they demonstrate that their algorithm can be efficiently implemented on Single Instruction Multiple Data (SIMD) architectures, such as vector units of modern CPUs and general purpose graphic processors (GPGPUs).

We also require the sampling of random numbers that follow a particular probability distribution, e.g., binomial or hypergeometric distributions. For this purpose, we use the acceptance-rejection method. [283, 337] Thus, we are able to generate binomial and hypergeometric random variates in expected constant time $\mathcal{O}(1)$. [313, 315]

7.2.3 GPGPU Computation Model

The computation and programming model for GPGPUs varies from traditional CPU programming in several aspects. Computations are organized in blocks of threads. All threads of a block have access to a common memory and are able to use synchronization between them. Blocks, on the other hand, are scheduled independent from each other and have no means of synchronization or communication. The threads of a block are processed in a SIMD-style manner. Branches in the code are possible, however threads of a block taking different branches are no longer processed in parallel but sequentially. We consider an *accelerator model* where every PE has a GPGPU available to offload computations to but the CPU is considered the main processing and steering facility.

7.3 Related Work

This paper is the journal version of [139] augmented with more proofs and experiments as well as with material based on the results in [271]. We now cover important related work for each of the network models used in our work. Additionally, we highlight recent advances for other network models that are relevant for the design of our algorithms.

7.3.1 ER Model

Batagelj and Brandes [35] present optimal sequential algorithms for the $G(n, m)$ as well as the $G(n, p)$ model. Their $G(n, p)$ generator makes use of an adaptation of a linear time sampling algorithm (Algorithm D) by Vitter. [334] In particular, the algorithm samples skip distances between edges of the resulting graph. Thus, they are able to generate a $G(n, p)$ graph in optimal time $\mathcal{O}(n + m)$. Their $G(n, m)$ generator is based on a virtual Fisher-Yates shuffle [128] and also has an optimal running time of $\mathcal{O}(n + m)$.

Nobari et al. [264] proposed a data parallel generator for both the directed and undirected $G(n, p)$ model. Their generators are designed for graphics processing units (GPUs). Like the generators of Batagelj and Brandes [35], their algorithm is based on

²i.e., with probability at least $1 - P^{-c}$ for any constant c

sampling skip distances but uses precomputations and prefix sums to adapt it for a data parallel setting.

7.3.2 RGG Model

Generating random geometric graphs with n vertices and radius r can be done naïvely in time $\Theta(n^2)$. This bound can be improved if the vertices are known to be generated uniformly at random. [178] To this end, a partitioning of the unit square into squares with side length r is created. To find the neighbors of each vertex, we consider each cell and its neighbors. The resulting generator has an expected running time of $\mathcal{O}(n + m)$.

Holtgrewe et al. [178, 179] proposed a distributed memory parallelization of this algorithm for the two dimensional case. Using sorting and vertex exchanges between PEs, they distribute vertices such that edges can be generated locally. The expected time for the local computation of their generator is $\mathcal{O}(n/P \log(n/P))$, due to sorting. Perhaps more important for large supercomputers is that they need to exchange all vertices resulting in a communication volume of $\mathcal{O}(n/P)$ per PE.

We are not aware of efficient distributed implementations of RGG generators for dimensions greater than two.


7.3.3 RHG Model


Von Looz et al. [223, 224] propose two different algorithms for generating random hyperbolic graphs. Their first algorithm relates the hyperbolic space to Euclidean geometry using the Poincaré disk model to perform neighborhood queries on a polar quadtree. The resulting generator has a running time of $\mathcal{O}((n^{3/2} + m) \log n)$.

In their second approach, von Looz et al. [224] propose a generator with an observed running time of $\mathcal{O}(n \log n + m)$. Their algorithm uses a partitioning of the hyperbolic plane into concentric ring-shaped annuli where vertices are stored in sorted order. Neighborhood queries are computed using angular boundaries for each annulus to bound the number of vertex comparisons.

Bringmann et al. [69] introduce a generalization of random hyperbolic graphs called *Geometric Inhomogenous Random Graphs (GIRGs)*. Their model simplifies theoretical studies of random hyperbolic graphs by ignoring constant factors while maintaining their qualitative behavior. Additionally, they propose an optimal sampling algorithm for GIRGs with expected linear time.

Finally, independent of this work, Penshuck [271] proposed a memory-efficient streaming generator that can be adapted to a distributed setting. They partition the hyperbolic plane into concentric annuli similarly to von Looz et al. [224]. They use a sweep-line based algorithm to generate nodes and edges on the fly in a request-centric fashion. They propose two practical algorithms optimized for either a time complexity of $\mathcal{O}(n \log \log n + m)$ or a memory footprint of $\mathcal{O}([n^{1-\alpha} \bar{d}^\alpha + \log n] \log n)$ whp.. Additionally, they present a shared memory parallelization of their algorithms that can be adapted to a distributed setting with a constant communication overhead.

GIRG:
 [Chapter 8](#)

HYPERGEN: sweep-line streaming generator:
 [Chapter 6](#)

7.3.4 *RDT* Model

As the Delaunay triangulation (DT) of a point set is uniquely defined, generating random Delaunay graphs can be separated into generating a random point set and computing its DT. A plethora of algorithms for computing the DT of a given point set in two and three dimensions exist. Funke and Sanders [140] review recent work on parallel DT algorithms and propose a competitive DT algorithm suitable for large clusters. The generation of a random point set is identical to the one in the *RGG* model.

7.3.5 Miscellaneous

7.3.5.1 Barabasi and Albert Model

Batagelj and Brandes [35] give an optimal sequential algorithm for the preferential attachment model of Barabasi and Albert [32]. Sanders and Schulz [294] parallelize this algorithm that appears to be inherently sequential. They observe that each edge can be generated independently if the randomness used for generating other edges is reproduced redundantly and consistently using a pseudorandom hash function. We adapt this technique to other random graph models.

7.3.5.2 Recursive Matrix Model

The *Recursive Matrix Model* (*R-MAT*) by Chakrabarti et al. [87] is a special case of the stochastic Kronecker graph model. [213] This model is well known for its usage in the popular Graph 500 benchmark. Generating a graph with n vertices and m edges is done by sampling each of the m edges independently. For this purpose, the adjacency matrix is recursively subdivided into four partitions. Each partition is assigned an edge probability $a + b + c + d = 1$. Recursion continues until a 1×1 partition is encountered, in which case the corresponding edge is added to the graph. The time complexity of the *R-MAT* generator thus is $\mathcal{O}(m \log n)$ since recursion has to be repeated for each edge.

7.4 *ER* Generator

We now introduce our distributed graph generators, starting with the Erdős-Rényi generators for both the directed and undirected case.

7.4.1 Directed Graphs

Generating a directed graph in the $G(n, m)$ model is the same as sampling a graph from the set of all possible graphs with n vertices and m edges. To do so, we can sample m edges uniformly at random from the set of all possible $n(n - 1)$ edges. Since we are not interested in graphs with parallel edges, sampling has to be done *without* replacement. To this end, we adapt the distributed sampling algorithm by Sanders et al. [292].

Our generator starts by dividing the set of possible edges into P *chunks*, one for each PE. Each chunk represents a set of rows of the adjacency matrix of our graph. We then assign each chunk to its corresponding PE using its id i . Afterwards, PE i is

responsible for generating the sample (set of edges) for its chunk. Note that we can easily adapt this algorithm to an arbitrary number of consecutive chunks per PE.

To compute the correct sample size (number of edges) for each chunk, we use the same divide-and-conquer technique used by the distributed sampling algorithm [292] (see Section 7.2.2). The resulting samples are then converted to directed edges using simple offset computations.

Theorem 7.1. The directed $G(n, m)$ generator runs in time $\mathcal{O}((n + m)/P + \log P)$ with high probability. ◀

Proof. Our algorithm is an adaptation of the distributed sampling algorithm that evenly divides the set of vertices, and therefore the set of potential edges, between P PEs. Thus, the universe per PE has size $\mathcal{O}(n(n - 1)/P)$ and the running time directly follows from the proof given by Sanders et al. [292]. ◻

7.4.2 Undirected Graphs

In the undirected case, we have to ensure that an edge $\{i, j\}$ is sampled by *both* PEs, the one that is assigned i and the one that is assigned j . Since each PE is assigned a different chunk, they follow different paths in the recursion tree. Hence, due to the independence of the random variables generated in each recursion tree, it is highly unlikely that they both sample the edge $\{i, j\}$ independently. To solve this issue, we introduce a different partitioning of the graphs adjacency matrix into chunks.

Our generator begins by dividing each dimension of the adjacency matrix into P sections of size roughly n/P . A chunk is then defined as a set of edges that correspond to a $(n/P) \times (n/P)$ submatrix of the adjacency matrix. Due to the symmetry of the adjacency matrix, we are able to restrict the sampling to the lower triangular adjacency matrix. Thus, we have a total of $P(P + 1)/2$ chunks that can be arranged into a triangular $P \times P$ chunk matrix. Afterwards, each PE is assigned a row and column of this matrix based on its id i as seen in Figure 7.1. By generating rectangular chunks instead of whole rows or columns, we can make sure that both PE i and PE $j \leq i$ redundantly generate chunk (i, j) using the same set of random values. In turn, they both sample the same set of edges independently without requiring communication. Note that our partitioning scheme into chunks results in each chunk being computed twice (once for each associated PE) except for the chunks on the diagonal of our chunk matrix. Therefore, the recomputation overhead is bounded by $2m$.

We now explain how to adapt the divide-and-conquer algorithm by Sanders et al. [292] for our chunk matrix. To generate the required partitioning of the adjacency matrix, we start by dividing the $P \times P$ chunk matrix into equal-sized quadrants. This is done by splitting the rows (and columns) into two equal-sized sections $\{1, \dots, l\}$ and $\{l + 1, \dots, P\}$. We choose $l = \lceil P/2 \rceil$ as our splitting value.

We then compute the number of edges within each of the resulting quadrants. Since we are only concerned with the lower triangular adjacency matrix, there are two different types of quadrants: triangular and rectangular. The second and fourth quadrant are triangular matrices with l and $P - l$ rows (and columns) respectively. We

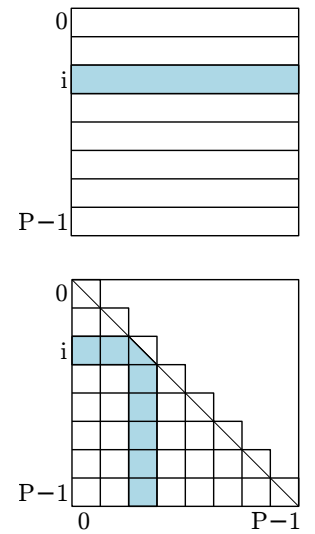


Figure 7.1: An adjacency matrix subdivided into chunks in the directed (top) and undirected (bottom) case. The chunk(s) for PE i are highlighted in blue.

then generate a set of three hypergeometric random variates to determine the number of samples (edges) in each quadrant. As for the distributed sampling algorithm [292], we make use of pseudorandomization via hash functions that are seeded based on the current recursion subtree to synchronize variates between PEs.

Each PE then uses its id to decide which quadrants to handle recursively. Note that at each level of the recursion, a PE only has to handle two of the four quadrants. We use a sequential sampling algorithm once a single chunk remains. Offset computations are performed to map samples to edges based on the type of the chunk (rectangular or triangular). The resulting recursion trees has at most $\lceil \log P \rceil$ levels and size $(4P^2 - 1)/3$.

Theorem 7.2. The undirected $G(n, m)$ generator requires time $\mathcal{O}((n + m)/P + P)$ with high probability. \blacktriangleleft

Proof. Each PE i has to generate a total of P chunks consisting of a single triangular submatrix and $P - 1$ rectangular submatrices. Additionally, each edge $\{i, j\}$ has to be generated twice (except when $P = 1$), once by the PE that is assigned vertex i , and once by the PE that is assigned vertex j . Thus, we have to sample a total of $2m$ edges. At every level of our recursion, we need to split the quadrants and in turn compute three hypergeometric random variates. Therefore, the time spent at every level only takes expected constant time. Since there are at most $\lceil \log P \rceil$ levels until each PE reaches its P chunks, the total time spent on recursion is whp. $\sum_{i=0}^{\lceil \log P \rceil} 2^i = 2(P - 1) = \mathcal{O}(P)$.

Following the proof by Sanders et al. [292], we can use Chernoff bounds to show that the total number of samples (edges) that is assigned to any PE is whp. $\mathcal{O}(m/P)$. Thus, the undirected $G(n, m)$ generator has a running time of $\mathcal{O}((n + m)/P + P)$. \square

7.4.3 Adaptations for the $G(n, p)$ Model

We now discuss how to adapt our previous generators for the $G(n, p)$ model. The key observation for the $G(n, p)$ generators is that we do not have to recursively compute hypergeometric random variates in order to derive the correct number of edges for each chunk. Since the distribution of vertices for each individual chunk is predetermined, we can determine the sample size for each chunk by generating binomial random variates. To make sure the sample size for an individual chunk is the same across PEs, the binomial random generator is seeded using a hash value based on the id of the chunk. Afterwards, we perform the same sampling procedure used to generate edges in the $G(n, m)$ generator.

7.4.3.1 Adaptation to GPGPUs

Since the *ER* generators are a direct application of sampling, the GPGPU implementation from [292] can be used to generate graphs on PEs with GPGPUs available. As before, each PE is assigned a chunk and computes the correct sample size and seeds for the pseudorandom generator on the CPU and then invokes the GPGPU algorithm to sample the edges of the graph.

7.5 RGG Generator

Generating a d -dimensional random geometric graph can be done naively in $\Theta(n^2)$ time. We reduce this bound by introducing a spatial grid data structure similar to the one used by Holtgrewe et al. [178]. We use a uniform grid of cells with side length $\max(r, n^{-1/d})$. The vertices of the graph are first placed into the grid cells. Edges must then run between vertices within one cell or between neighboring cells. Hence, for a point A assigned to a cell C , it suffices to perform distance calculations to the points in cell C and its neighboring cells (3^d cells overall).

Theorem 7.3. The expected work for generating a random geometric graph with n nodes and m edges is $\mathcal{O}(n + m)$. ◀

Proof. The work for placing the points is $\Theta(n)$. The work for initializing the cell array is proportional to its size

$$\left(\frac{1}{\max(r, n^{-1/d})} \right)^d \leq \left(\frac{1}{n^{-1/d}} \right)^d = n .$$

For estimating the remaining work, we estimate the expected number of edges m as well as the number of comparisons Y between points.

Consider the indicator random variable Z_{ij} that is 1 if there is an edge between points i and j and 0 otherwise. Then, $m = \sum_{i \neq j} Z_{ij}$. There is an edge between a fixed point i and another point j if j is placed in a ball of radius r (and volume $\Theta(r^d)$) around point i . Note that at least a constant fraction of this ball intersects with the unit cube. The ratio between the volume of this ball and the volume of the unit cube is $\Theta(r^d)$. Overall, $\mathbb{P}[Z_{ij} = 1] = \Theta(r^d)$, and, exploiting the linearity of expectation,

$$\mathbb{E}[m] = \sum_{i \neq j} \mathbb{E}[Z]_{ij} = \Theta(n^2 r^d) . \tag{7.5}$$

Similarly, consider the indicator random variable Y_{ij} that is 1 if points i and j are compared and 0 otherwise. Then, $Y = \sum_{i \neq j} Y_{ij}$. Points i and j are compared if they are placed into neighboring cells. Recall that each cell has $\Theta(1)$ neighboring cells (including itself). We now make a case distinction depending on what determines the cell size. If $r \geq n^{-1/d}$ we have r^{-d} cells. Considering a fixed point i , a point j is thus placed into one of the $\Theta(1)$ neighboring cells with probability $\Theta(r^d)$. Hence, exploiting the linearity of expectation,

$$\mathbb{E}[Y] = \sum_{i \neq j} \mathbb{E}[Y_{ij}] = \sum_{i \neq j} \mathbb{P}[Y_{ij} = 1] = \Theta(n^2 r^d) = \Theta(\mathbb{E}[m]) .$$

When $r < n^{-1/d}$, there are n cells. We can make a similar calculation as above, now with $\mathbb{P}[Y_{ij}] = \Theta(1/n)$ which yields $\mathbb{E}[Y] = n^2/\Theta(n) = \Theta(n)$. ◻

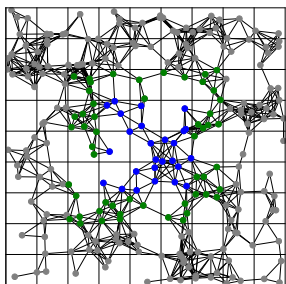


Figure 7.2: A two dimensional random geometric graph with 256 vertices and a radius of 0.11 on nine PEs. The local vertices of PE 4 are highlighted in blue. The non-local vertices computed redundantly by PE 4 are highlighted in green.

7.5.1 Parallelization

We now discuss how to parallelize our approach in a communication-free way. To the best of our knowledge, the resulting generator is the first efficient distributed implementation of a *RGG* generator for $d > 2$.

Our generator again uses the notion of chunks. A chunk in the *RGG* case represents a rectangular section of the unit cube. We therefore partition the unit cube into P disjoint chunks and assign one of them to each PE. There is one caveat with this approach, in that the possible values for P are limited to powers of d . To alleviate this issue, we generate more than P chunks and distribute them evenly between PEs. To be more specific, we are able to generate $k = 2^{db} \geq P$ chunks and then distribute them to the PEs in a locality-aware way by using a Z-order curve. [251]

Each PE is then responsible for generating the vertices in its chunk(s) as well as all their incident edges. Again, we use a divide-and-conquer approach similar to the previous generators.

For this purpose the unit cube is evenly partitioned into 2^d equal-sized subcubes. In turn, the probability for a vertex to be assigned to an individual subcube is the ratio of the area of the subcube to the area of the whole cube. Thus, we can generate $2^d - 1$ binomial random variates to compute the number of vertices within each of the subcubes. The binomial distribution is parameterized using the number of remaining vertices n and the aforementioned subcube probability p . As for the *ER* generators, variates are generated by exploiting pseudorandomization via hash functions seeded on the current recursion subtree. Therefore, we generate the same variates on different PEs that follow the same recursion. In turn, we require no communication for generating local vertices. Note that the resulting recursion tree has at most $\lceil \log P \rceil$ levels and size $(2^d P - 1)/2^d - 1$. Once a PE is left with a single chunk, we compute additional binomial random variates to get the number of vertices in each cell of side length $c \geq r$.

As we want each PE to generate *all* incident edges for its local vertices, we have to make sure that the cells of neighboring chunks that are within the radius of local vertices are also generated. Because each cell has a side length c of at least r , this means we have to generate all cells directly adjacent to the chunk(s) of a PE. Due to the communication-free design of our algorithm, the generation of these cells is done through recomputations using the same divide-and-conquer algorithms as for the local cells. We therefore repeat the vertex generation process for the neighboring cells. Note that for sufficiently large graphs, each chunk consists of many cells so that redundantly generating border layers of cells becomes a negligible overhead. An example of the subgraph that a single PE generates for the two dimensional case is given in Figure 7.2.

Afterwards, we can simply iterate over all local cells and generate the corresponding edges by vertex comparisons with all vertices in each neighboring cell. To avoid duplicate edges, we only perform vertex comparisons for local neighboring cells with a higher id.

7.5.2 Analysis of the Parallel Algorithm

The above communication-free free parallel algorithm emulates a more traditional algorithms that places n points uniformly at random into their cells and performs the necessary distance calculations. Each PE takes time $\mathcal{O}(n/P + \log P)$ for generating chunks together with the required parts of the cell array. We do not analyze the number of performed distance calculation directly but indirectly by analyzing the emulated algorithm. We first note that using standard Chernoff bound arguments, one can prove the following lemmas.

Lemma 7.4. If the random variable Occ denotes the occupancy of a cell then $\text{Occ} = \mathcal{O}(\mathbb{E}[\text{Occ}] + \ln n)$ with probability $1 - n^{-c}$ for any constant $c > 0$. ◀

Lemma 7.5. If the random variable W denotes the number of cells allocated to one PE then $W = \mathcal{O}(n/P + \ln n)$ with probability $1 - n^{-c}$ for any constant $c > 0$. ◀

Furthermore, analogous to Theorem 7.3, one can prove that the expected work at each PE is $\mathcal{O}((m+n)/P)$ (taking into account that at most a constant fraction of cells has to be generated redundantly).

Lemma 7.6. There are expected $\mathcal{O}((m+n)/P)$ distance calculations per PE. ◀

However, this does not suffice to bound the parallel execution time since the PE assigned the largest work determines the overall running time. We conjecture that the amount of work performed on each PE is $\mathcal{O}((m+n)/P)$ whp. for $n = \Omega(p \log^2 n)$. However, we do not know how to prove that formally due to dependencies in the involved random variables (e.g., the variables Y_{ij} and Z_{ij} from the proof of Theorem 7.3). Instead, we prove the following more loose result.

Theorem 7.7. For any $c > 0$, there is a constant $a(c)$ such that $n \geq a(c)P^2 \log^3 P$ implies that the amount of work performed by each PE is $\mathcal{O}((m+n)/P)$ with probability at least $1 - n^{-c}$. ◀

Proof. Let $\mathbf{X} = X_1, \dots, X_n$ denote the vector of positions of the n randomly placed points. Let $Y(\mathbf{X})$ denote the number of distance computations performed by a fixed PE. Since this is a function of n independent random variables, we can apply the bounded difference inequality. [233] We have

$$\mathbb{P}[Y(\mathbf{X}) > \mathbb{E}[Y(\mathbf{X})] + \delta] \leq \exp\left(-\frac{2\delta^2}{nb^2}\right), \quad (7.6)$$

where b is a bound on the maximum change in the value of $Y(\mathbf{X})$ when one of the random variables X_i is changed. Changing X_i means moving point i . This changes the number of distance computation by the occupancy of the $\mathcal{O}(1)$ cells neighboring the source and target cell of the moved point. Since the worst-case value of the occupancies is very large (n), we condition on the case that the bound from Lemma 7.4 applies. Note that the remaining cases are sufficiently unlikely, say have probability $\leq n^{-c}/2$.

If δ is large enough such that $\exp(-2\delta^2/nb^2) \leq n^{-c}/2P$, Eq. (7.6) yields the desired result. The factor 2 in the right hand side comes from the fact that we reserve half of the

allowed failure probability for the above conditioning. The factor P comes from the fact that we want to bound the work done on *all* PEs. Since we already assume that $n > 2P$, we will make the stronger requirement $\exp(-2\delta^2/nb^2) \leq n^{-(c+1)}$. Solving for δ yields

$$\delta \geq b\sqrt{(c+1)n \ln(n)/2} = \Omega\left(b\sqrt{n \ln n}\right). \quad (7.7)$$

We now make a case distinction on the ball radius r . If $r \geq (\ln(n)/n)^{1/d}$, the expected occupancy $\Theta(nr^d)$ of a cell is $\Omega(\ln n)$ and Lemma 7.4 yields that $b = \Theta(nr^d)$ is also a high probability bound. Condition (7.7) then becomes $\delta = \Omega(n^{3/2}r^d \sqrt{\ln(n)})$. At the same time we want $\delta = \mathcal{O}(\mathbb{E}[m/P]) = \mathcal{O}(n^2r^d/P)$; see also Eq. (7.5). Both conditions can hold if $n^{3/2}r^d \sqrt{\ln(n)} = \mathcal{O}(n^2r^d/P)$. This is equivalent to $n = \Omega(P^2 \ln n)$. Since this is only a nontrivial condition when n is polynomial in P , we get the equivalent condition $n = \Omega(P^2 \ln P) \leq \Omega(P^2 \ln^3 P)$.

Similarly, for the case $r < (\ln(n)/n)^{1/d}$, the expected occupancy of a cell is $\mathcal{O}(\ln n)$ and Lemma 7.4 yields $b = \Theta(\ln n)$. Condition (7.7) becomes $\delta = \Omega(\sqrt{n} \ln^{1.5} n)$. We want at the same time that $\delta = \mathcal{O}(n/P)$. Both conditions hold when $\sqrt{n} \ln^{1.5} n = \mathcal{O}(n/P)$, or, equivalently, $n = \Omega(P^2 \ln^3 n)$ implying $n = \Omega(P^2 \ln^3 P)$. \square

7.5.3 Adaptation to GPGPUs

As before, each PE is responsible for generating the vertices and edges of one chunk. The algorithm for GPGPUs follows two phases. In the first phase, the PE generates the appropriate seeds and vertex numbers for the cells of its chunk and all neighboring cells on the CPU. Subsequently, the vertices of these cells are sampled on the GPGPU. Depending on the expected number of vertices per cell, a cell is either processed by a whole block with several threads or by a single thread, therefore grouping several cells in one block. Recall, as the cell side length c is greater than r , only the cells of neighboring chunks immediately adjacent to the PEs chunk need to be generated.

In the second phase, the actual edges between the vertices are determined, which requires a three step algorithm. In the first step, for each cell and its neighbors, the number of edges with length smaller than r are counted on the GPGPU. Secondly, the prefix sum of these counts provides both the total number of edges generated as well as offsets into the edge array for each block. The CPU can then allocate memory on the GPGPU for the third stage and the cells are processed again, this time actually outputting all edges into the newly allocated array. The amount of work performed per vertex is the same for all vertices of a cell – as the same number of vertices need to be considered in the neighboring cells – but can differ between cells. Therefore, each cell is processed by one block on the GPGPU to avoid any load balancing issues.

7.6 RDT Generator

The point generation for Delaunay graphs follows the principles of *RGG*, differing only in the definition of the cell side length c set to the mean distance of the $(d+1)$ th-nearest neighbor for n vertices in the unit d -hypercube, $c \approx [(d+1)/n]^{1/d}$. [42]

To produce the DT of the generated point set, our algorithm proceeds as follows. For each assigned chunk, the PE considers the chunk itself plus a *halo* of neighboring cells. Initially, the cells directly adjacent to the chunk are added to the halo. The PE computes the DT of the chunk plus halo and checks whether all points of the convex hull are from the halo and whether each computed simplex s that contains at least one point from the inside of the assigned chunk has a circumsphere that is completely contained within the chunk plus halo. The local computation finishes when both conditions are fulfilled. Otherwise, the halo is expanded by one layer of cells and the DT is updated; see also [140, 220]. As for RGG, all PEs generate the same vertices for the same cell.

We do not have a complete analysis of the algorithm but note that by Lemma 7.5, each PE get assigned $\mathcal{O}(n/P + \log P)$ nodes from the chunks whp. and that the halo contributes only a lower order term as long as the number of extension steps remains constant. Our experiments indicate that usually no repetitions at all are needed. Moreover, a Delaunay triangulation of a random point set can be computed in linear time. [114] Hence, we might conjecture a running time of $\mathcal{O}(n/P + \log P)$ for our algorithm.

Adaptation to GPGPUs

For the *RDT* generator, the points can be sampled on the GPGPU according to the algorithm outlined for the *RGG* generator in the previous section. Following point generation, the algorithm of Cao et al. [78] can be used to compute the DT in two and three dimensions on the GPGPU. Their algorithm initially produces a near-Delaunay triangulation on the GPGPU and then fixes potential violations using a star splaying technique on the CPU. The subsequent steps can be efficiently performed on the GPGPU again: checking whether the circumhypersphere of all simplices is contained within the halo and, if not, generating the next layer of halo cells by first generating the seeds and vertex numbers of those cells on the CPU and then sampling the points on the GPGPU. Since Cao et al. propose an incremental construction algorithm, it can be directly applied to insert the newly generated halo points into the DT.

7.7 RHG Generators

We now describe two approaches for generating random hyperbolic graphs. The first generator (RHG) requires precomputing local vertices before processing neighborhood queries and allows for an already partitioned output graph. However, this comes at the cost of load balancing and memory limitations. The second generator (sRHG) processes vertices and their incident edges in a streaming fashion. Although this does not directly give a partitioned output graph, it significantly improves load balancing and memory requirements. Additionally, we cover important optimizations that improve the performance of both generators.

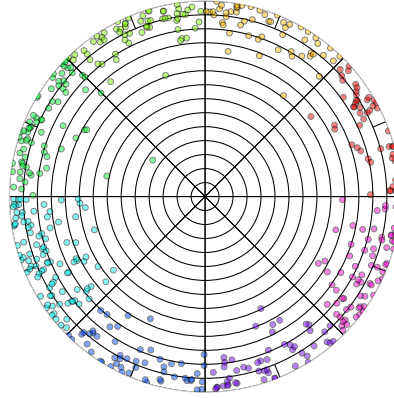


Figure 7.3: Partitioning of the hyperbolic plane into set of equidistant annuli and chunks and cells. Each chunk is distributed to one PE and further subdivided into cells. The number of vertices is $n = 512$ with an average degree of $\bar{d} = 4$ and a powerlaw exponent of $\gamma = 2.6$. The expected number of vertices per cell is set to 2^4 . The local vertices for each PE are highlighted in different colors.

7.7.1 In-memory Generator

As for the *RGG* generator, we can naïvely create a random hyperbolic graph in $\Theta(n^2)$ by comparing all pairs of vertices. We first improve this bound by partitioning the hyperbolic plane (cf. [68, 223, 224]). To this end, we split the hyperbolic disk of radius R into $k := \lfloor \alpha R / \ln(2) \rfloor$ concentric annuli of constant height, i.e. annulus i covers the radial interval $[\ell_{i-1}, \ell_i)$ with $\ell_0 = 0$ and $\ell_k = R$. This results in $k = \mathcal{O}(\log n)$ annuli due to Eq. (7.1).

Since each PE has to determine the number of vertices in each annulus, we compute a multinomial random variate with k outcomes: we iteratively compute a set of dependent binomial random variates via pseudorandomization. The probability that a specific vertex is assigned to annulus i is given by integration over the radial density function (Eq. (7.3)) between the annulus' limits ℓ_i and ℓ_{i+1} :

$$p_i = \int_{\ell_i}^{\ell_{i+1}} f(r) dr \stackrel{\text{(Eq.7.3)}}{=} \frac{\cosh(\alpha \ell_{i+1}) - \cosh(\alpha \ell_i)}{\cosh(R) - 1}.$$

Hence, the expected number n_i of vertices in annulus i follows $\mathbb{E}[n_i] = n \cdot p_i$.

We further partition each annulus in the angular direction into P chunks using a divide-and-conquer approach that uses binomial random variates as for the other generators. The resulting recursion tree within a single annulus has a height of $\lceil \log P \rceil$.

Finally, we perform a third partitioning of chunks into a set of equal-sized cells in the angular direction. The number of cells per chunk is chosen such that each cell contains an expected constant number of vertices k . Figure 7.3 shows the resulting partitioning of vertices in the hyperbolic plane into cells and annuli.

Lemma 7.8. Assuming $n = \Omega(P \log P)$, our partitioning algorithm assigns each PE $\mathcal{O}(n/P)$ vertices with high probability. ◀

Proof. Chunks are chosen such that they assign each PE i an equally sized angular interval of the hyperbolic plane $[i \cdot 2\pi/P, (i + 1) \cdot 2\pi/P)$. The number of vertices per chunk in an annulus is generated through a set of binomial random variates. This results in a uniform distribution of the vertices in the interval $[0, 2\pi)$ with respect to their angular coordinate. Thus, each PE is assigned $\mathcal{O}(n/P)$ vertices in expectation. Assuming $n = \Omega(P \log P)$ this holds whp. by a standard Chernoff bound. \square

Lemma 7.9. Generating the grid data structure for P PEs takes time $\mathcal{O}(P \log n + n/P)$ with high probability. \blacktriangleleft

Proof. The time spent during the chunk creation per annulus is $\mathcal{O}(P)$ whp. since the size of the recursion tree is at most $2P - 1$ and we only spend expected constant time per level for generating the binomial random variates. We have to repeat this recursion for each of the $\mathcal{O}(\log n)$ annuli. Thus, the total time spent for the recursion over all annuli is $\mathcal{O}(P \log n)$. The runtime bound then follows by adding the time for vertex creation (Lemma 7.8). \square

7.7.2 Neighborhood Queries

We now describe how we use our grid data structure to efficiently reduce the number of vertex comparisons. For this purpose, we begin by iterating over the cells in increasing order from the innermost annulus outwards and perform a neighborhood query for each vertex.

The query begins by determining how far the angular coordinate of a potential neighbor $u = (r_u, \theta_u)$ is allowed to deviate from the angular coordinate of our query vertex $v = (r_v, \theta_v)$. If we assume that u lies in annulus i with a lower radial boundary ℓ_i , we can use the distance inequality

$$|\theta_u - \theta_v| \leq \arccos \left(\frac{\cosh(r_v) \cosh(\ell_i) - \cosh(R)}{\sinh(r_v) \sinh(\ell_i)} \right)$$

to determine this deviation. We then gather the set of cells that lie within the resulting boundary coordinates. To do so, we start from the cell that intersects the angular coordinate of our query vertex and then continue outward in both angular directions until we encounter a cell that lies outside the boundary. For each cell that we encounter, we perform distance comparisons to our query vertex and add edges accordingly. To avoid the costly evaluation of trigonometric functions for each comparison we maintain a set of precomputed values (see Section 7.7.5.1). Note that in order to avoid duplicate edges in the sequential case, we can limit neighborhood queries to annuli that have an equal or larger radial boundary than our starting annulus.

Lemma 7.10. Consider a query vertex with radius r and an annulus with boundaries $[a, b)$. Our candidate selection overestimates the probability mass of the actual query range by a factor of $OE(b-a, \alpha)$ where $OE(x, \alpha) := \frac{\alpha-1/2}{\alpha} \frac{1-e^{\alpha x}}{1-e^{(\alpha-1/2)x}}$. \blacktriangleleft

Proof. If $r < R-b$, the circle around the querying vertex covers the annulus completely. Hence, each candidate is a true neighbor and the selection process is optimal.

We now consider $r \geq R-a$ and omit the fringe case of $R-b < r < R-a$ which follows analogously. The probability mass $\mu_Q := \mu [B_R(r) \cap (B_b(0) \setminus B_a(0))]$ of the intersection of the actual query circle $B_R(r)$ with the annulus $B_b(0) \setminus B_a(0)$ is calculated in Eq. (7.19) (Appendix). Our generator overestimates the actual query range at the border and covers the mass $\mu_H := \frac{1}{\pi} \Delta\theta(r, a) \int_a^b \rho(y) dy$ as detailed in Eq. (7.20) (Appendix). The claim follows by the division of both mass functions μ_H/μ_Q . \square

Corollary 7.11. Given a constant annulus height, i.e., $b-a = \mathcal{O}(1)$, Lemma 7.10 implies a constant overestimation for any $\alpha > 1/2$. In case of $b-a = \lfloor \ln(2)/\alpha \rfloor$, we have $OE(1, \alpha) \leq \sqrt{e} \approx 1.64 \quad \forall \alpha > 1/2$. \blacktriangleleft

Lemma 7.12. Let N_j be the number of neighbors the point $p_j = (r_j, \theta_j)$ has from below, i.e. neighbors with smaller radius. With high probability there exist only $\mathcal{O}(n/\log^2 n)$ points with $N_j = \mathcal{O}(n^{1-\alpha} \bar{d}^\alpha \log(n))$ while the remainder of points with $r_j > R/2$ has $N_j = \mathcal{O}(n^{1-2\alpha} \log^{2\alpha}(n) \bar{d}^{2\alpha})$ neighbors. \blacktriangleleft

Proof. Let X_1, \dots, X_n be indicator variables with $X_i=1$ if p and p_i are adjacent. Due to radial symmetry we directly obtain the expectation value of X_i conditioned on the radius p_i :

$$\mathbb{E}[X_i \mid r_i = x] = \mathbb{P}[X_i=1 \mid r_i = x] = \begin{cases} 1 & \text{if } x < R-r \\ \Delta\theta(x, r)/\pi & \text{otherwise} \end{cases}$$

We remove the conditional using the Law of Total Expectation and Eqs. (7.16) and (7.17):

$$\mathbb{E}[X_i] = \int_0^{R-r} \rho(x) dx + \frac{1}{\pi} \int_{R-r}^r \rho(x) \Delta\theta(x, R) dx \quad (7.8)$$

$$= [e^{-\alpha r} - e^{-\alpha R}] (1+o(1)) + \frac{1}{\pi} \frac{\alpha}{\alpha - \frac{1}{2}} e^{-\alpha r} \quad (7.9)$$

$$\cdot [e^{(\alpha - \frac{1}{2})(2r-R)} - 1] (1 \pm \mathcal{O}(e^{-r})) \quad (7.10)$$

Fix the radius $r_T = R - \frac{2}{\alpha} \log \log n$ with $R/2 < r_T$ (wlog) and consider three cases for r : First, we ignore all points $r \leq R/2$ as they belong to the central clique and are irrelevant here. Second, observe that with high probability there exist $\mathcal{O}(n/\log^2(n))$ points below r_T . Exploiting the monotonicity of Eq. (7.10) in r , we maximize it by setting $r = R/2$, which cancels out the second term. Linearity of the expectation value, substitution of $R = 2 \log(n) + C$, and the definition of the expected degree yield $\mathbb{E}[\sum_i X_i] = \mathcal{O}(n (\bar{d}/n)^\alpha)$. Then, Chernoff's bound gives $\sum_i X_i = \mathcal{O}(n^{1-\alpha} \bar{d}^\alpha \log(n))$ with high probability. Third, for all points above r_T , set $r = r_T$ yielding $\sum_i X_i = \mathcal{O}(n^{1-2\alpha} \log^{2\alpha}(n) \bar{d}^{2\alpha})$ with high probability analogously. \square

Lemma 7.13. The time complexity of the sequential RHG generator for n vertices with radius R , an average degree \bar{d} , and a powerlaw exponent $\gamma \geq 2$ is $\mathcal{O}(m)$ with probability $1 - n^{-c}$ for any constant $c > 0$. \blacktriangleleft

Proof. We bound our generators time complexity for each component individually:

- The preprocessing requires $\mathcal{O}(1)$ time per point making it non-substantial.
- Processing the vertices within the cells requires $\mathcal{O}(n)$ time in total with high probability.
- By applying Lemma 7.12 and Corollary 7.11, the candidate selection requires $\mathcal{O}(n \log \bar{d}) = \mathcal{O}(m)$ time with high probability.
- All distance calculations require in total $\mathcal{O}(m)$ time since Corollary 7.11 bounds the fraction of computations that do not yield an edge to $\mathcal{O}(1)$. \square

To adapt our queries for a distributed setting, we need to recompute all non-local vertices that lie within the hyperbolic circle (of radius R) of any of our local vertices. To find these vertices, we perform an additional inward neighborhood query.

Queries in the distributed setting work similarly to the sequential case, with the addition that any non-local chunks that we encounter during the search are recomputed. These newly generated vertices are then assigned their respective cells and stored for future searches.

One issue with this approach is that the innermost annulus, which contains only a constant number of vertices (w.h.p.), is divided into P chunks. However, since all vertices with radius $r \leq R/2$ form a clique and are almost always contained within the search radius for any given vertex, we compute and store these points redundantly in a single chunk on all PEs. This severely lowers the running time for inward searches, especially for a large number of PEs.

Theorem 7.14. The expected time complexity of the parallel RHG generator for n vertices with radius R , average degree \bar{d} and a powerlaw exponent $\gamma \geq 2$ is

$$\mathcal{O}\left(\frac{n+m}{P} + P \log n + n^{1-\alpha} (P\bar{d})^\alpha n^{\frac{1}{2\alpha}}\right). \quad \blacktriangleleft$$

Proof. We bound the time complexity by considering each component individually:

- Building our cell data structures takes time $\mathcal{O}(P \log n)$ as shown in Lemma 7.9.
- Sampling local vertices and edges has an expected running time of $\mathcal{O}((n+m)/P)$ as for the sequential approach.
- The expected number of vertices recomputed during the inward search can be bounded by $\mu(B_r(0)) = \mathcal{O}(n^{1-\alpha} (P\bar{d})^\alpha)$ (see Lemma 7.15).
- The expected number of edges computed for high degree vertices (as well as the number of vertices recomputed during the outward search) can pessimistically be bounded by assuming that vertices in the inner annuli (see Lemma 7.15) all have maximum degree $\Delta = n^{1/(2\alpha)+o(1)}$. [159] This yields an expected number of $\mathcal{O}(n^{1-\alpha} (P\bar{d})^\alpha n^{1/(2\alpha)})$ edges. \square

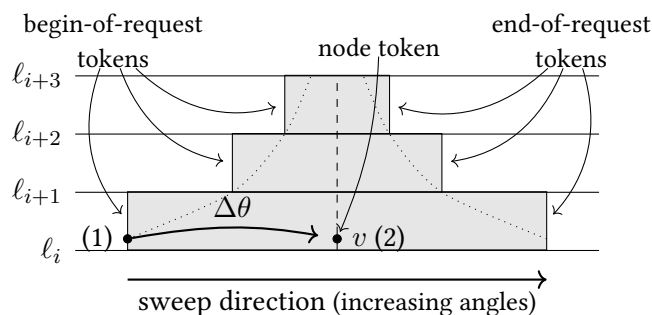


Figure 7.4: The shaded area illustrates the region in which candidates for node v can be found (an overestimate of the dotted hyperbolic query circle). It is encoded with one request per annulus. Instead of generating points at random, sRHG draws the *beginning of requests* (1) and then places the points (2) accordingly by increasing their angle by $\Delta\theta$. Only when the sweep-line encounters the begin of a request in annulus i , the request is propagated to annulus $i+1$.

7.7.3 Streaming Generator

We now present sRHG, a generator that improves the load balancing and the memory requirements of RHG. Extending [271], its main idea is to invert the neighborhood search: while RHG selects a node and then directly searches all neighbors by ruling out wrong candidates, sRHG does the opposite and first accumulates all queries a node is candidate in before processing them in a single batch. This not only reduces unstructured accesses to main memory, but more importantly allows us to narrow the window of space that has to be kept in memory.

As for RHG, we decompose the hyperbolic plane into a set of concentric annuli and draw the number of points in each annuli uniformly at random. This step is performed by all PEs independently, and we use pseudorandomization to ensure that each PE draws the same numbers without communication. Next we perform the second decomposition by splitting each annulus in the angular direction into a set of P chunks of equal size.

Conceptually, sRHG then executes a sweep-line algorithm in angular direction starting with the innermost annulus. To this end, the PE maintains a sweep-line state per annulus storing the currently active requests and pending events: as illustrated in Figure 7.4, we use tokens for each node v to mark the *position* of v , as well as the *begin* and *end* of the region in which neighbor candidates of v can be found.

sRHG executes the next pending event (i.e. the unprocessed token with smallest polar coordinate) of the current annulus:

- A *begin-of-request* token adds the request to the current sweep state, and creates a *begin-of-request* token for the next higher annulus (if existing). It also generates an appropriate *end-of-request* token for the current annulus.
- An *end-of-request* removes the request from the current sweep state.
- If a *node token* is found, the distance to each node with a request in the current sweep state is computed and an is edge emitted if it is below the threshold R .

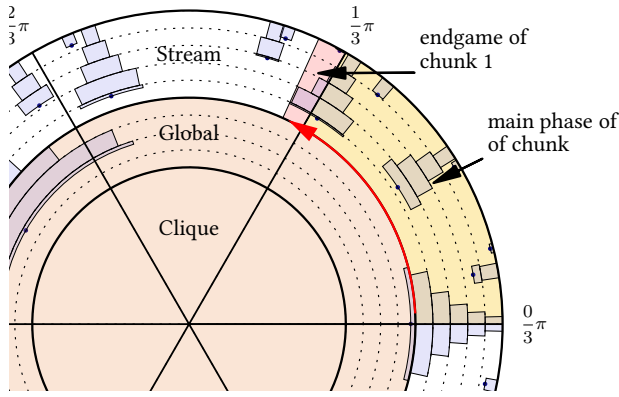


Figure 7.5: The hyperbolic plane is partitioned along the polar axis into P chunks of equal size. Radially, there are two groups: the *lower global annuli* which are preprocessed and kept in memory, and the *upper streaming annuli*. In the main phase, each PE streams through its chunks towards increasing polar angles (red arrow). Requests overlapping into the next chunk are then completed in the *final phase*.

Observe this design causes sRHG to process the begin-of-request token of a node v *before* its node token becomes active. We hence invert the causality relation, and draw begin-of-request tokens from a monotonic sequence of uniform variates and position a matching point token accordingly.

To reduce the memory footprint, we do not complete an annulus before starting the next higher one. Instead, each PE interleaves the processing of its local annuli with the only constraint that no sweep-line may overtake its adjacent neighbor below it. This is legal, since the only information flow is from lower to higher annuli: it is triggered by begin-of-request tokens which never move towards smaller angles during this process.

sRHG partitions the annuli into two groups, lower *global annuli* and upper *streaming annuli*, and starts by processing the global annuli first (see Figure 7.5):

- *Global annuli* are those where the maximum potential request width of a point within that annulus is larger than $2\pi/P$ (the width of a single chunk). Similarly to RHG, we merge the innermost annuli with a radial boundary below $R/2$ into a special clique annulus.
- *Streaming annuli* are those where the maximum potential request width of a point within that annulus is at most $2\pi/P$.

Let r_G be the smallest radial boundary of a streaming annulus (i.e. every streaming annulus i has a lower boundary of $\ell_{i-1} \geq r_G$). We obtain $r_G \lesssim R/2 + \log(2P/\pi)$ for $P \geq 2$ by solving $2\Delta\theta(r_G, r_G) = 2\pi/P$ for r_G and applying Eq. (7.16).

Lemma 7.15. The expected number of vertices generated in the global annuli is $\mathcal{O}(n^{1-\alpha}(P\bar{d})^\alpha)$. ◀

Proof. Consider a point (r_G, θ) with a request width of at most $2\Delta\theta(r_G, r_G)$. The number of vertices $n_G(P)$ that each PE has to generate during the global phase is thus binomially distributed around the mean of

$$\mathbb{E}[n_G(P)] = n\mu(B_{r_G}(0)) = n \left(\frac{\bar{d}P}{2n} \right)^\alpha \left(\frac{\alpha - \frac{1}{2}}{\alpha} \right)^{2\alpha} = \mathcal{O}(n^{1-\alpha}(P\bar{d})^\alpha). \quad \square$$

Lemma 7.16. Assuming $n = \Omega(P \log P)$, the number of vertices generated in the streaming annuli of any PE is $\mathcal{O}(n/P)$ with high probability. \blacktriangleleft

Proof. Each PE is assigned a polar interval of $2\pi/P$ width and generates all streaming points whose request begins there. As the angle at which a request starts is drawn uniformly at random, the numbers (n_1, \dots, n_P) of vertices generated by each PE are distributed multinomially with an equal bucket mass of $p = 1/P$. We pessimistically place all point with in the streaming annuli (cf. Lemma 7.15), and hence have $\sum_i n_i = n$ and $\mathbb{E}[n_i] = \mathcal{O}(n/P)$ for all i . Concentration follows directly from Chernoff bounds. \square

7.7.4 Global Annuli

By construction, points in the global annuli have long requests, potentially covering the whole hyperbolic space. In order to guarantee that no PE has to generate all vertices, requests within the lower global annuli are computed redundantly on all PEs. Again, consistency across PEs is achieved using pseudorandomness. Each PE then restricts the requests to its own streaming chunk and propagates applicable ones to a designated insertion buffer in the first upper streaming annulus.

During the creation of a request, it might happen that we encounter an angular deviation of $[a, b]$ where either $a < 0$ or $b > 2\pi$. Taking the angular 2π -period of the hyperbolic plane into account, these requests are separated into two ranges. To be more specific, we separate the angular deviation $[a, b]$ with $a < 0$ into the two ranges $[a + 2\pi, 2\pi]$ and $[0, b]$. The case $b > 2\pi$ is treated analogously.

Due to the nature of hyperbolic space, vertices in the global annuli are likely to have a very high degree as they have a relatively small hyperbolic distance to any other point. However, due to our request-centric approach the computation of their neighbors in the upper streaming annuli is fully distributed.

We distribute the execution of requests for the inner annuli evenly across all PEs. This results in a much more even distribution of work compared to the query-centric approach. However, it does not directly produce a partitioned output graph.

7.7.5 Streaming Annuli

After this so-called global phase, we continue with the upper streaming annuli. By construction, each PE is responsible for generating and processing all requests within its local chunks (i.e. one per annulus).

It maintains a context for each of its streaming annuli consisting of:

- The sweep state containing all active requests.
- An PRNG emitting monotonically increasing variates distributed uniformly over the chunk's polar interval.
- A priority queue to receive *begin-of-request* tokens from the annuli below (referred to as insertion buffer).
- A priority queue storing points generated after drawing their *begin-of-request*.
- A priority queue storing *end-of-request* tokens.

As aforementioned, we execute a sweep-line algorithm and always process (and remove) the token with smallest angle from one of the four sources. Note that by definition of the angular width of each request, the candidate selection for each request gives the same overestimation as our previous generator (see Lemma 7.13). The in-order generation of requests and edges however significantly decreases unstructured memory accesses compared to RHG.

While sRHG needs to process annuli in an interleaved fashion to bound the insertion buffers' sizes, it tries to infrequently switch between annuli to improve data locality. We implemented this by splitting each chunk into cells; the number of cells per annulus is chosen such that the expected number of points in them is constant. The algorithm then switches between annuli only after processing complete cells.

We conclude the main generation if all sweep-lines reached the upper bound of the PE's polar interval. Observe that at this point, unprocessed *node* or *end-of-request* token can remain which are dealt with during final phase.

For the final phase, each PE is responsible for generating edges from pairs where either the request or vertex stem from the main phase. To do so, we repeat the same process as in the local phase, but replicate the configuration of the PE responsible for the adjacent chunks. We also annotate vertices and requests from these foreign chunks in order to not generate duplicate edges.

The final phase involves at most one additional chunk as all points and requests with large polar shift were processed during the global phase. In practice, it can often be stopped earlier once all old vertices and requests are processed, we count their number.

Lemma 7.17. If we limit the final phase to the size of a chunk, the expected number of edges generated for streaming annuli is $\mathcal{O}\left(\left(\frac{n}{P}\right)^{2-\alpha} \bar{d}^\alpha\right)$. ◀

Proof. Following the proof of Lemma 7.12, we introduce indicator variables X_1, \dots, X_n with $X_i = 1$ if two points p and p_i are adjacent. This yields the expected value

$$\begin{aligned} \mathbb{E}[X_i] &= \int_0^{R-r} \rho(x) dx + \frac{1}{\pi} \int_{R-r}^r \rho(x) \Delta\theta(x, R) dx \\ &= [e^{-\alpha r} - e^{-\alpha R}] (1+o(1)) + \frac{1}{\pi} \frac{\alpha}{\alpha - \frac{1}{2}} e^{-\alpha r} \cdot [e^{(\alpha - \frac{1}{2})(2r-R)} - 1] (1 \pm \mathcal{O}(e^{-r})) \end{aligned}$$

We now (pessimistically) assume $r = r_G$ (Lemma 7.15). The resulting expected number of neighbors is given by $\mathbb{E}[X_i] = \mathcal{O}(P^{\alpha-1}(\bar{d}/n)^\alpha - (\bar{d}/n)^{2\alpha})$. In turn, we can use Lemma 7.16 to bound the expected total number of request and vertex pairs for vertices in the streaming annuli by $\mathcal{O}((n/P)^{2-\alpha}\bar{d}^\alpha)$. \square

Lemma 7.18. The time complexity of the sequential sRHG generator for n vertices with radius R , an average degree \bar{d} , and a powerlaw exponent $\gamma \geq 2$ is $\mathcal{O}(m)$ with probability $1 - n^{-c}$ for any constant $c > 0$. \blacktriangleleft

Proof. We bound the time complexity by considering each component individually:

- The preprocessing requires $\mathcal{O}(1)$ time per point making it non-substantial.
- Handling of cliques is trivially bounded by $\mathcal{O}(m)$ as every step emits an edge.
- By applying Lemma 7.12 and Corollary 7.11, the candidate selection requires $\mathcal{O}(n \log \bar{d}) = \mathcal{O}(m)$ time with high probability. Here we exploit that request tokens can be addressed to discrete cells allowing for linear time integer sorting.
- All distance calculations require in total $\mathcal{O}(m)$ time since Corollary 7.11 bounds the fraction of computations that do not yield an edge to $\mathcal{O}(1)$. \square

Theorem 7.19. The expected time complexity of the parallel sRHG generator for n vertices with radius R , average degree \bar{d} and a powerlaw exponent $\gamma \geq 2$ is bounded by $\mathcal{O}(\frac{n+m}{P} + P \log n + n^{1-\alpha}(P\bar{d})^\alpha + (\frac{n}{P})^{2-\alpha}\bar{d}^\alpha)$. \blacktriangleleft

Proof. We bound the time complexity by considering each component individually:

- Building our cell data structures takes time $\mathcal{O}(P \log n)$ as shown in Lemma 7.9.
- The expected number of vertices that have to be recomputed for the global annuli $\mathcal{O}(n^{1-\alpha}(P\bar{d})^\alpha)$ due to Lemma 7.15.
- Lemma 7.17 bounds the expected number of distance comparisons for the outer annuli to $\mathcal{O}((n/P)^{2-\alpha}\bar{d}^\alpha)$. \square

Given the running time of our parallel algorithm, we assume $n/P = k = \Omega(\log n)$ vertices per PE and set k to $n^{2/3}$. For values of $\gamma \geq 3$ this results in a running time linear in the number of edges per PE $\mathcal{O}(m/P)$. However, for very small values of γ close to 2, the running time is dominated by the global phase and becomes nearly linear.

7.7.5.1 Further Optimizations

Adjacency tests without trigonometric functions

The runtime of preliminary versions of our generators was dominated by repeated evaluations of trigonometric functions. We address this issue with a precomputing scheme. Let $p = (r_p, \theta_p)$ and $q = (r_q, \theta_q)$ be two arbitrary vertices in \mathcal{D}_R and let ℓ_i be the lower radial boundary of annulus i .

The scheme accelerates the two most frequent query types, namely the computation of request widths of the form $\Delta\theta(r_p, \ell_i)$ and tests whether the hyperbolic distance $d(p, q) < R$ of two vertices falls below the threshold R . The former computation type occurs $\Omega(n)$ times, while the latter is required in each of the $\Omega(m)$ candidate checks. Based on Eq. (7.15) we obtain

$$\Delta\theta(r_p, \ell_i) = \text{acos} \left(\coth(r_p) \cdot \coth(\ell_i) - \frac{\cosh(R)}{\sinh(\ell_i)} \cdot \frac{1}{\sinh(r_p)} \right), \quad (7.11)$$

where $\coth(x) := \cosh(x)/\sinh(x)$.

By precomputing $\coth(r_p)$ and $1/\sinh(r_p)$ for each vertex, as well as $\coth(\ell_i)$ and $\cosh(R)/\sinh(\ell_i)$ for each annulus, the argument of \cos^{-1} follows from two multiplications and a subtraction. Similarly, we test $d(p, q) < R$ by rewriting Eq. (7.4) as

$$\begin{aligned} & \cos(\theta_p) \cos(\theta_q) + \sin(\theta_p) \sin(\theta_q) \\ & > \coth(r_p) \coth(r_q) - \cosh(R) \frac{1}{\sinh(r_p)} \frac{1}{\sinh(r_q)}. \end{aligned} \quad (7.12)$$

The left-hand-side is an expansion of $\cos(\theta_p - \theta_q)$. Hence, the precomputation of $\cos(\theta_p)$ and $\sin(\theta_p)$ for each vertex gives distance checks in at most five multiplications and two additions; it can be further improved by reusing parts of earlier computations.

After precomputation, Section 7.7.5.1 can be vectorized efficiently to compute the distance between a node and multiple requests in a data parallel fashion. To support vectorized computations, we also use a structure-of-arrays memory layout to store active candidates. We employ the Vc library [199] for explicit vectorization.

Batch-processing of Requests

Our streaming generator effectively sweeps all annuli in an interleaved fashion and maintains for each annulus a separate state containing the active candidates. During this sweep, it encounters three event types, namely the occurrence of a vertex, the beginning of a request, and eventually its end.

Our implementation splits each annulus into cells of equal width and then processes these events batch-wise. Given the number n_j of vertices in annulus j , we select the number c_j of cells in annulus j such that $c \leq n_j/c_i < 2c$ where c is a small tuning parameter (typically 8). More precisely, we choose c_j as a power-of-two which by construction aligns cell boundaries between annuli and avoids corner cases when traversing the geometry.

When entering a cell, we move all requests contained into the active state by first overwriting obsolete requests that went out-of-scope in the last cell; we thereby avoid redundant operations otherwise caused by separated deletions and insertions. Subsequently, all vertices contained are matched against the active candidates again increasing data locality and exploiting minor synergies.

The usage of cells also allows us to relax the sorting of requests received from below since we only need to distribute start and endpoints of requests to the appropriate cells. Our implementations hence only stores for each request the indices of cells in which it starts and ends, and orders the items in a radix heap.

7.8 Experimental Evaluation

We now present the experimental evaluation of our graph generators. For each algorithm, we perform a running time comparison and analyze its scaling behavior.

7.8.1 Implementation

An implementation of our graph generators (*KaGen*) in C++ is available at <https://github.com/sebalamm/KaGen>. We use Spooky Hash³ as a hash function for pseudorandomization. Hash values are used to initialize a Mersenne Twister [232] for generating uniform random variates. Non-uniform random variates are generated using the stoc library⁴. If the size of our inputs (e.g., the adjacency matrix size) exceeds 64 bit, we use the multiple-precision floating points library GMP⁵ and a reimplementation of the stoc library. Profiling indicates that most generators spend only a negligible fraction of their time in random number generation ($\leq 1\%$). For the *ER* generator this figure is about 20%. Hence, we did not experiment with alternative implementations. All algorithms and libraries are compiled using g++ version 5.4.1 using optimization level `fast` and `-march=native`. In the distributed setting, we use Intel MPI version 1.4 compiled with g++ version 4.9.3.

7.8.2 Experimental Setup

We use two different machines to conduct our experiments. Sequential comparisons are performed on a single core of a dual-socket Intel Xeon E4-2670 v3 system with 128 GB of DDR4-2133 memory, running Ubuntu 14.04 with kernel version 3.13.0-91-generic. If not mentioned otherwise, all results are averages of ten iterations with different seeds.

For scaling experiments and parallel comparisons we use the Phase 1 thin nodes of the SuperMUC supercomputer. The SuperMUC thin nodes consist of 18 islands and a total of 9216 nodes. Each compute node has two Sandy Bridge-EP Xeon E5-2680 8-core processors, as well as 32 GB of main memory. Each node runs the SUSE Linux Enterprise Server (SLES) operating system. We use the maximum number of 16 cores per node for our scaling experiments. The maximum size of our generated instances is limited by the memory per core (2 GB). To generate even larger instances, one could use a full streaming approach which will be discussed in Section 7.9.

We analyze the scaling behavior of our algorithms in terms of weak and strong scaling. Weak scaling measures how the running time varies with the number of PEs for a fixed problem size *per PE*. Analogously, strong scaling measures the running time for a fixed problem size over *all PEs*. Due to memory limitations of the SuperMUC, strong scaling experiments are performed with a minimum of 1024 PEs. Again, results are averaged over ten iterations with different seeds.

³<http://www.burtleburtle.net/bob/hash/spooky.html>

⁴<http://www.agner.org/random/>

⁵<http://www.mpfr.org>

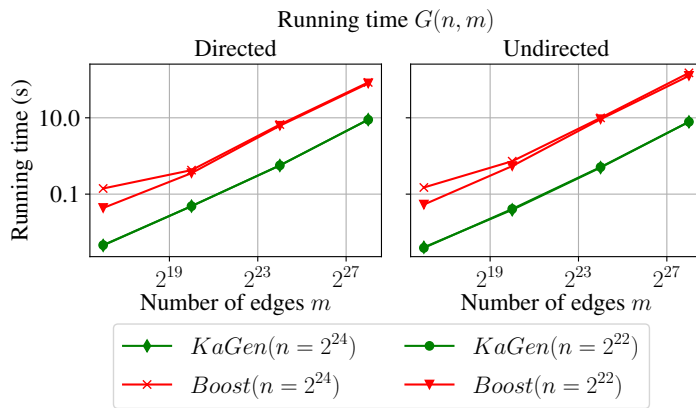


Figure 7.6: Running time for the sequential directed (left) and undirected (right) Erdős-Rényi generators for 2^{22} and 2^{24} vertices and 2^{16} to 2^{28} edges.

7.8.3 Erdős-Rényi Generator

There are various implementations of efficient sequential Erdős-Rényi generators available (e.g., [35]). However, there is little to no work on distributed memory generators. Thus, we perform a sequential comparison of our generator to the implementation found in the Boost⁶ library. Their generator uses a sampling procedure similar to Algorithm D [334] and serves as an example for an efficient linear time generator.

For our comparison, we vary the number of vertices from 2^{18} to 2^{24} and the number of edges from 2^{16} to 2^{28} . Figure 7.6 shows the running time for both generators for the two largest sets of vertices.

First, we note that both implementations have a constant time per edge for large m . However, the Boost implementation also has an increasing time per edge for growing numbers of vertices n . In contrast, the running time of our generator is independent of n . This is no surprise, since our generator uses a simple edge list and does not maintain a full graph data structure.

For the directed $G(n, m)$ model, our generator is roughly 10 times faster than Boost for the largest value of $m = 2^{28}$. In the undirected case, our $G(n, m)$ generator is roughly 21 times faster and has an equally lower running time. All in all, the results are consistent with the optimal theoretical running times of $\mathcal{O}(n + m)$ for both algorithms.

Next, we discuss the scaling behavior of our Erdős-Rényi generators. For the weak scaling experiments, each PE is assigned an equal number of n/P vertices and m/P edges to sample. In particular, we set $n = m/2^4$ and let the number of edges per PE range from 2^{22} to 2^{26} . For the strong scaling experiments, we keep the number of edges fixed from 2^{34} to 2^{38} . Results are presented in Figure 7.7 and Figure 7.8 respectively.

We can see that our directed generator has an almost perfect scaling behavior. Only for the smaller input sizes and more than 2^{12} PEs, the logarithmic term of our running time becomes noticeable. The minor irregularities that we observe for the largest number of PEs are due to performance differences for nodes in the supercomputer. Nonetheless, our results are consistent with our asymptotic running time $\mathcal{O}((n + m)/P + \log P)$.

If we look at the scaling behavior of our undirected generator, we can see that

⁶http://www.boost.org/doc/libs/1_62_0/libs/graph/doc/erdos_renyi_generator.html

Figure 7.7: Running time for generating m edges and $n = m/2^4$ vertices on P PEs using the $G(n, m)$ generators.

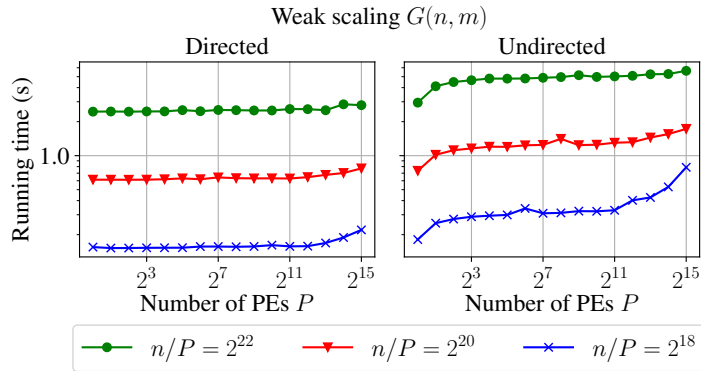
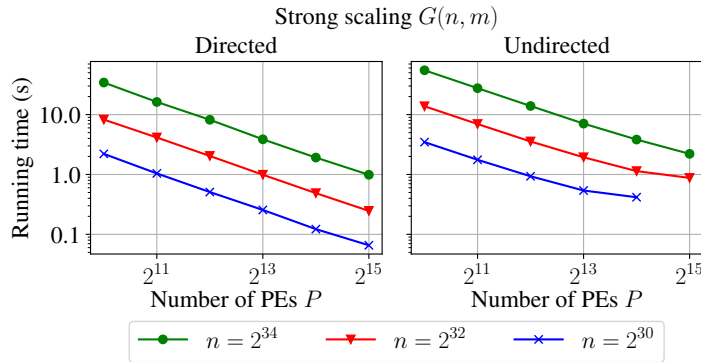


Figure 7.8: Running time for generating m edges and $n = m/2^4$ vertices on P PEs using the $G(n, m)$ generators.



for small numbers of PEs the running time starts to increase and then remains nearly constant. This is due to the fact that as the number of PEs/chunks increases, the number of redundantly generated edges also increases up to twice the number of sequentially sampled edges. This effect is not noticeable in the strong scaling case, since we perform these experiments with a minimum of 1024 PEs. Furthermore, for smaller values of m/P and large P , we also see a linear increase in running time. We attribute this to the linear time $\mathcal{O}(P)$ needed to locate the correct chunks for each PE.

7.8.4 RGG Generator

There are various implementations of the naïve $\Theta(n^2)$ generator available (e.g., [163]). However, a more efficient and distributed algorithm is presented by Holtgrewe et al. [178].

Since their algorithm and our own generator are nearly identical in the sequential case, we are mainly interested in their parallel running time for a growing number of PEs. Therefore, we measure the total running time and vary the input size per PE n/P from 2^{16} to 2^{20} . It should be noted that Holtgrewe et al. only support two dimensional random geometric graphs and thus the three-dimensional generator is excluded. The radius is set to $r = 0.55\sqrt{\frac{\ln n}{n}}/\sqrt{P}$. This choice ensures that the resulting graph is almost always connected [21] and is used in many previous papers. Figure 7.9 shows the running time of both competitors for a growing number of $P = p^2$ PEs. Additionally, Figure 7.10 shows weak scaling experiments for our two- and three-dimensional generators. Finally, we present the strong scaling behavior of our generators in Figure 7.11. For these

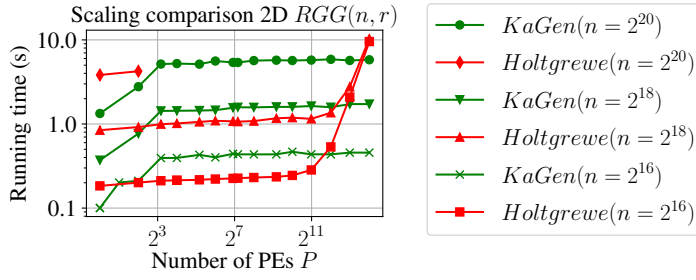


Figure 7.9: Running time of 2d-RGG generators for growing numbers of PEs $P = p^2$ and a constant input size n/P per PE. The radius r is set to $r = 0.55 \sqrt{\frac{\ln n}{n}} / \sqrt{P}$.

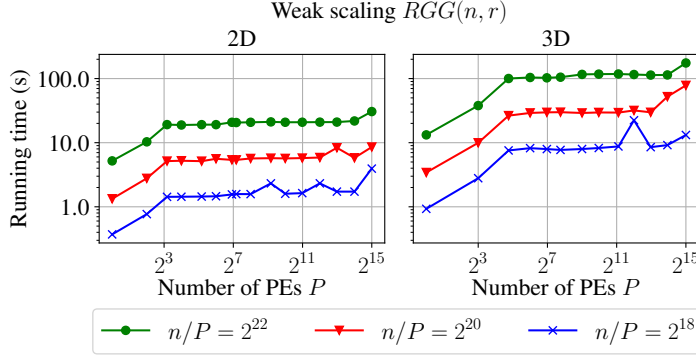


Figure 7.10: Running time for generating n vertices on P PEs using the RGG generators. The radius r is set to $0.55 \sqrt[2,3]{\frac{\ln n}{n}} / \sqrt{P}$.

experiments, the number of vertices n is fixed from 2^{26} to 2^{32} .

Due to the recomputations used by our generator, Holtgrewe et al. quickly become faster by up to a factor of two as the number of PEs increases. To be more specific, the number of neighbors that we have to generate redundantly increases from zero for one PE up to eight neighbors for more than four PEs. This increase in running time can be bounded by computing the additional amount of vertices created through redundant computations and multiplying it by the average degree $n\pi r^2$. For our particular choice of r this yields roughly twice the running time needed for the sequential computation, which is consistent with the experimental results. However, for sufficiently sparse graphs, the additional time for recomputations is negligible as the number of vertices in each cell becomes constant. A very similar analysis can also be done for our three-dimensional generator. Again, minor irregularities are due to performance differences for individual nodes in the supercomputer.

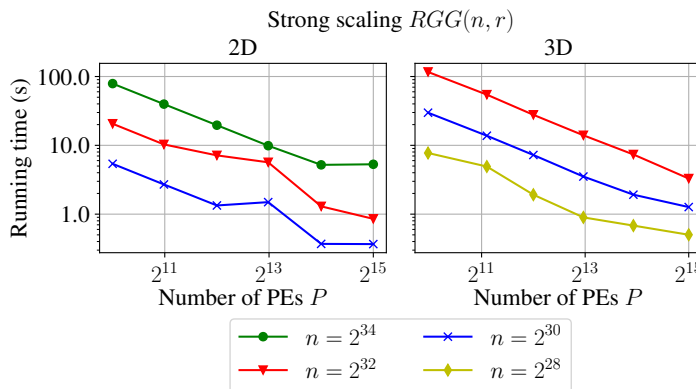


Figure 7.11: Running time for generating n vertices on P PEs using the RGG generators. The radius r is set to $0.55 \sqrt[2,3]{\frac{\ln n}{n}}$.

Figure 7.12: Running time for generating a graph with n vertices on P PEs using the RDT generators.

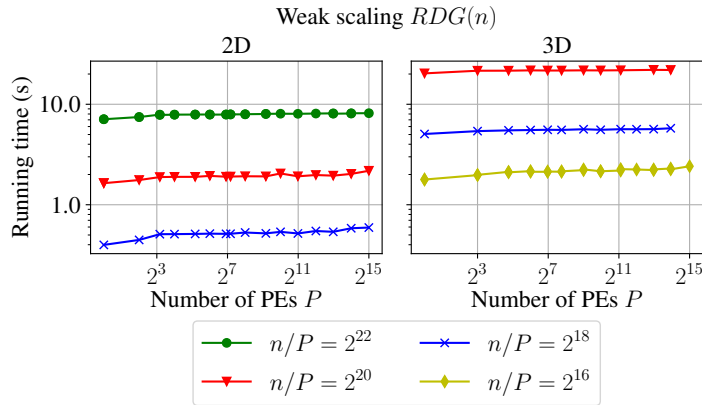
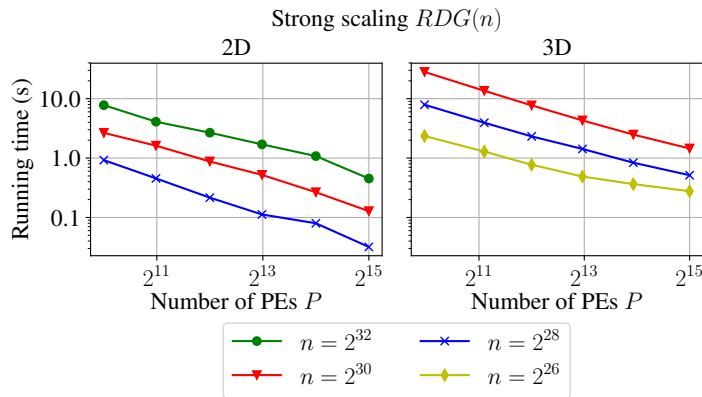


Figure 7.13: Running time for generating a graph with n vertices on P PEs using the RDT generators.



Once we reach 2^{12} PEs, the communication required by Holtgrewe et al. rapidly becomes noticeable and *KaGen* is significantly faster. Overall, the results are in line with the asymptotic running time presented in Section 7.5.

7.8.5 RDT Generator

Our implementation uses the CGAL library [175] to compute the DT of the vertices in a chunk and its halo. CGAL provides a state-of-the-art implementation also used by most of the other available DT generators. We therefore omit sequential measurements.

The experimental setup for the *RDT* is equivalent to the *RGG* scaling experiments. For the weak scaling experiments, we vary the input size per PE from 2^{18} to 2^{22} for the 2D *RDT* and – due to memory constraints – from 2^{16} to 2^{20} for the three-dimensional one. Moreover, for 3D *RDT* and 2^{15} PEs, only the smallest input size could be computed within the memory limit per core of SuperMUC. For the strong scaling experiments, the input size varies from 2^{26} to 2^{32} . Our experiments show an almost constant time –depicted in Figure 7.12 and Figure 7.13– well in agreement with our conjectured asymptotic running time of $\mathcal{O}(n/P + \log P)$. Similarly to the *RGG*, the initial increase in runtime can be attributed to the redundant vertex generation of neighboring cells. As the halo rarely grows beyond the directly adjacent cells, no significant further increase in runtime can be observed for more than 2^8 PEs.

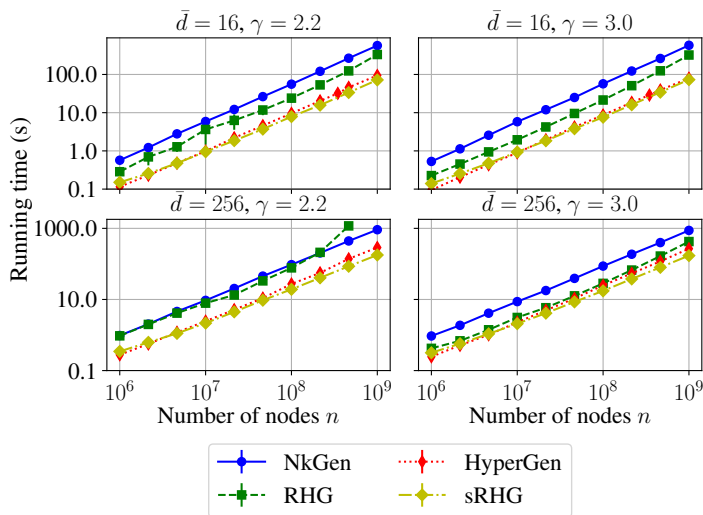


Figure 7.14: Running time as function of number n of nodes for powerlaw exponents $\gamma \in \{2.2, 3\}$ (i.e., $\alpha \in \{0.6, 1\}$) and average degrees $\bar{d} \in \{16, 256\}$. All generators use 39 threads or processes, on a dual-socket Intel Xeon Broadwell E5-2640 v4 machine with and 128 GB of RAM.

7.8.6 RHG Generator

We compare RHG and sRHG to the state-of-the-art generators NkGen⁷ [224], and HyperGen [271] (see Section 7.3.3). Since both reference implementations support shared-memory parallelism only, we restrict the experiments to a single machine with 40 hardware threads.

We measure the runtime of all generators as the number n of nodes varies between $10^6 \leq n \leq 10^9$ for different average degrees \bar{d} and powerlaw exponents γ . These values mimic settings found in various real-world networks. [223]

In general, NkGen exhibits the highest runtime per edge generated. We attribute this to the fact that the implementation uses only partial precomputation and heavily relies on unstructured accesses to main memory. NkGen is typically followed by RHG which demonstrates limited scaling for small values of γ ; in this setting, the increase in runtime for large graphs is primarily caused by exhaustion of the system’s main memory. We consider this unrepresentative for the distributed case, in which the collective memory available is typically much larger.

The related sRHG and HyperGen are consistently the fastest implementations with sRHG producing up to $7.5 \cdot 10^8$ edges per second for $\bar{d} = 256$ and $\gamma = 3$ on a single machine. Similar to RHG both use precomputation as a means to speed up distance computation. Additional performance gains are due their emphasis on cache and memory efficiency and data parallelism.

Finally, we present the results of our scaling experiments for the RHG generators. For the weak scaling experiments each PE is again assigned an equal number of n vertices. Note, that we use a designated computing node with 16 cores for calculating the inner core for our second generator. Thus, the corresponding scaling experiments start at 32 cores. The number of vertices per PE ranges from 2^{16} to 2^{24} . Again, for the strong scaling experiments the number of vertices is fixed and ranges from 2^{28} to 2^{32} . The powerlaw exponent γ varies from 2.2 to 3.0 to cover different extremes of the degree

⁷Optimized version of the generator found in the NetworKit library <http://network-analysis.info> [271].

Figure 7.15: Running time for generating a graph with n vertices, average degree $\bar{d} = 16$ and $\gamma = 3.0$ on P PEs using both RHG generators.

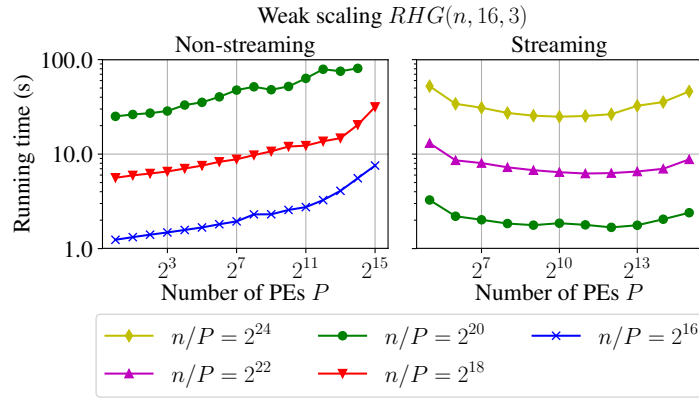
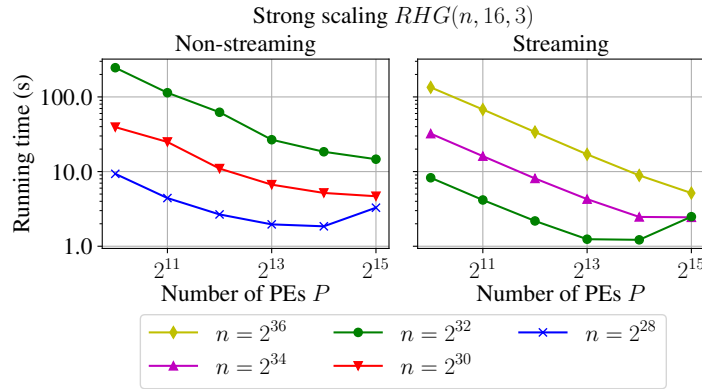


Figure 7.16: Running time for generating a graph with n vertices, average degree $\bar{d} = 16$ and $\gamma = 3$ on P PEs using the non-streaming RHG generator.



distribution. Figure 7.15 ($\gamma = 3$) shows the weak scaling results of the RHG generators with an average degree of $\bar{d} = 16$. Likewise, Figure 7.16 ($\gamma = 3$) shows the strong scaling results of the RHG generators with an average degree of $\bar{d} = 16$.

Looking at the scaling behavior of our first generator, we see that there is a considerable increase in running time for a growing number of PEs. We can attribute this behavior to the redundant computations that are introduced through parallelization. Additionally, high degree vertices are hard to distribute efficiently if we want a partitioned output graph. This severely impedes the scaling behavior. Since the maximum degree is $\mathcal{O}(n^{1/(2\alpha)})$ with high probability [159], these vertices dominate the running time of our algorithm. Overall, these effects are less noticeable in the case of our strong scaling experiments, because we start with a minimum of 1024 PEs.

If we examine the scaling behavior of our second algorithm, we can see that these effects are less noticeable, especially for larger values of γ . For smaller values of γ , the running time of our algorithm is again dominated by the time needed to generate the inner core (global annuli) on its dedicated computing node. Overall, our second generator is roughly 16 times faster than our first generator. However, keep in mind that the resulting graph is not fully partitioned, i.e., not all incident edges are generated on the corresponding PEs. Thus, we can achieve a similar speedup for our first generator, by only performing outward queries and omitting the inward ones. Nonetheless, the lower memory requirements of our second generator enable us to generate up to 16 times larger instances.

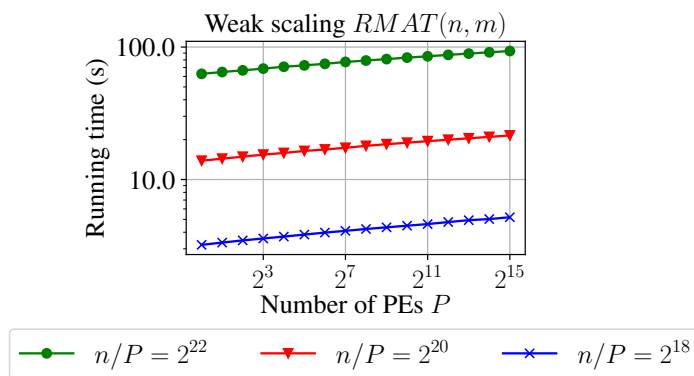


Figure 7.17: Running time for generating a graph with n vertices and $m = 2^4 n$ edges on P PEs using the R -MAT generator.

7.8.6.1 Comparison with R -MAT

In order to further evaluate the performance of our generators, in particular of our hyperbolic generators, we now compare them to the R -MAT generator. To this end, we use the reference implementation available at the Graph 500 website (graph500.org). The R -MAT generator is commonly used in benchmarks for large-scale graph computations due to its scalability and flexibility. Figure 7.17 shows the weak scaling behavior of R -MAT for an equal number of m/P edges per PE. In particular, we set the number of vertices to $n = m/2^4$ and let the number of edges per PE range from 2^{22} to 2^{26} .

First, we see that R -MAT has a slight increase in running time for growing numbers of PEs (and thus a growing number of vertices). This increase in running time is due to the fact that R -MAT needs to generate $\mathcal{O}(\log n)$ random variates for each edges. Second, if we compare the overall performance of R -MAT with our own generators we can see that it is roughly ten times slower than the streaming version of our hyperbolic graph generator. Furthermore, it is up to 15 times slower than our undirected Erdős-Rényi generator. This difference in performance can be attributed to the difference in the number of random variates that each generator consumes. Due to their generation cost, minimizing the number of variates yields large performance benefits.

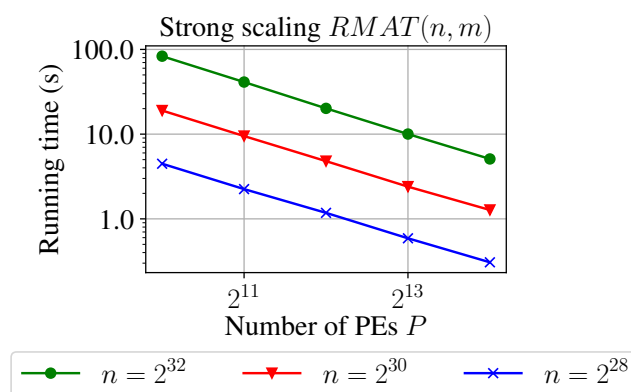
Finally, Figure 7.18 shows the strong scaling behavior of R -MAT for a fixed number of m edges ranging from 2^{28} to 2^{32} . We see that R -MAT has near-optimal scaling behavior even for the smallest inputs tested. Comparing this to the performance of our own generators, we can see that it does not suffer from the same increase in running time that our hyperbolic graph generators show for small inputs. This is due to the fact that the running time of R -MAT has no additive term that increases with a growing number of PEs.

7.9 Conclusion

We presented scalable graph generators for a set of commonly used network models. Our work includes the $G(n, m)$ and $G(n, p)$ models, random geometric graphs, random Delaunay graphs and random hyperbolic graphs.

Our algorithms make use of a combination of divide-and-conquer schemes and grid data structures to narrow down their local sampling space. We redundantly compute

Figure 7.18: Running time for generating a graph with n vertices and $m = 2^4 n$ edges on P PEs using the R-MAT generator.



parts of the network that are within the neighborhood of local vertices. These computations are made possible through the use of pseudorandomization via hash functions. The resulting algorithms are embarrassingly parallel and *communication-free*.

Our extensive experimental evaluation indicates that our generators are competitive to state-of-the-art algorithms while also providing near-optimal scaling behavior. In turn, we are able to generate instances with up to 2^{43} vertices and 2^{47} edges in less than 22 minutes on 32 768 cores.⁸ Therefore, our generators enable new network models to be used for research on a massive scale. In order to help researchers to use our generators, we provide all algorithms in a widely usable open source library. Finally, to show the broad applicability of the concepts used in our generators, we provide adaptations for their use in a GPGPU setting.

Future Work

As mentioned in Section 7.8.2, we would like to extend our remaining generators to use a streaming approach similar to sRHG (see Section 7.7.3). This would drastically reduce the memory needed for the auxiliary data structures, especially for the spatial network generators. Allowing the efficient generation of random hyperbolic graphs on GPGPUs also remains for future work.

Furthermore, we would like to extend our communication-free paradigm to various other network models such as the *Stochastic Block Model* and the *Binomial Random Hyperbolic Graph*. [200] More specifically, we would like to extend the *KaGen* library by a faster generator for *R-MAT* graphs than currently available.

Finally, our generators allow us to perform an extensive study on new graph models for high-performance computing benchmarks. In turn, these benchmarks could target a wider variety of real-world models and scenarios. A more detailed theoretical analysis using tighter bounds, especially for the parallel running times of our generators would be beneficial for this purpose.

⁸Using the directed $G(n, m)$ generator.

Acknowledgment

The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer SuperMUC at Leibniz Supercomputing Centre (www.lrz.de). The authors gratefully acknowledge the Gauss Centre for Supercomputing (GCS) for providing computing time through the John von Neumann Institute for Computing (NIC) on the GCS share of the supercomputer JUQUEEN [317] at Jülich Supercomputing Centre (JSC). GCS is the alliance of the three national supercomputing centres HLRS (Universität Stuttgart), JSC (Forschungszentrum Jülich), and LRZ (Bayerische Akademie der Wissenschaften), funded by the German Federal Ministry of Education and Research (BMBF) and the German State Ministries for Research of Baden-Württemberg (MWK), Bayern (StMWFK) and Nordrhein-Westfalen (MIWF). We thank the Center for Scientific Computing, University of Frankfurt for making their HPC facilities available. This work was partially supported by Deutsche Forschungsgemeinschaft (DFG) under grants ME 2088/3-2, and ME 2088/4-2.

Appendix 7.A Hyperbolic Geometry Related Definitions

Radial density:

$$\rho(r) := \alpha \frac{\sinh(\alpha r)}{\cosh(\alpha R) - 1} \quad (7.13)$$

Radial cdf:

$$\mu(B_r(0)) := \int_0^r \rho(x) dx = \frac{\cosh(\alpha x) - 1}{\cosh(\alpha R)} \quad (7.14)$$

Angular deviation:

$$\Delta\theta(r, b) := \begin{cases} \pi & \text{if } r+b < R \\ \arccos \left[\frac{\cosh(r) \cosh(b) - \cosh(R)}{\sinh(r) \sinh(b)} \right] & \text{otherwise} \end{cases} \quad (7.15)$$

Appendix 7.B Hyperbolic Geometry Related Approximations

Gugelmann et al. derived the following approximations⁹. [159]

Angular deviation:

$$\Delta\theta(r, b) = \begin{cases} \pi & \text{if } r + b < R \\ 2e^{\frac{R-r-b}{2}}(1 + \Theta(e^{R-r-b})) & \text{if } r + b \geq R \end{cases} \quad (7.16)$$

Radial cdf:

$$\mu(B_r(0)) = \int_0^r \rho(x)dx = \frac{\cosh(\alpha r)}{\cosh(\alpha R) - 1} \quad (7.17)$$

$$= e^{\alpha(r-R)}(1 + o(1)) \quad (7.18)$$

The probability mass μ_Q of the intersection of the actual query circle $B_R(r)$ with the annulus $B_b(0) \setminus B_a(0)$ as defined in Lemma 7.10 is given by:

$$\begin{aligned} \mu_Q &:= \mu[(B_b(0) \setminus B_a(0)) \cap B_R(r)] \\ &= \frac{2}{\pi} e^{-\frac{r}{2} - (\alpha - \frac{1}{2})R} \left[\frac{\alpha}{\alpha - \frac{1}{2}} \left(e^{(\alpha - \frac{1}{2})b} - e^{(\alpha - \frac{1}{2})a} \right) + \mathcal{O}\left(e^{-(\alpha - \frac{1}{2})a}\right) \right] \end{aligned} \quad (7.19)$$

RHG and sRHG overestimate the query range at the border and covers the mass μ_H :

$$\begin{aligned} \mu_H &:= \frac{1}{\pi} \Delta\theta(r, a) \int_a^b \rho(y)dy \\ &= \frac{2}{\pi} e^{-\frac{r}{2} - (\alpha - \frac{1}{2})R} \left[e^{\alpha b - a/2} - e^{(\alpha - \frac{1}{2})a} \right] \cdot \left(1 \pm \mathcal{O}\left(e^{(1-\alpha)(R-a)-r}\right) \right) \end{aligned} \quad (7.20)$$

⁹We drop the $(1 + \mathcal{O}(\cdot))$ error terms in our calculations without further notice if they are either irrelevant or dominated by other simplifications made

Efficiently Generating Geometric Inhomogeneous and Hyperbolic Random Graphs

joint work with T. Bläsius, T. Friedrich, M. Katzmann, U. Meyer, and C. Weyand

Random Hyperbolic Graph (RHG) and *Geometric Inhomogeneous Random Graphs (GIRG)* are two similar generative network models designed to resemble complex real-world networks; they have a powerlaw degree distribution with controllable exponent β , and high clustering that can be controlled via the temperature T .

We present the first implementation of an efficient *GIRG* generator running in expected linear time. Besides varying temperatures, it also supports underlying geometries of higher dimensions. It is capable of generating graphs with ten million edges in under a second on commodity hardware. The algorithm can be adapted to *RHG*s. Our resulting implementation is the fastest sequential *RHG* generator, despite the fact that we support non-zero temperatures. Though non-zero temperatures are crucial for many applications, most existing generators are restricted to $T = 0$. We also support parallelization, although this is not the focus of this paper. Moreover, we note that our generators draw from the correct probability distribution, i.e., they involve no approximation.

Besides the generators themselves, we also provide an efficient algorithm to determine the non-trivial dependency between the average degree of the resulting graph and the input parameters of the *GIRG* model. This makes it possible to specify the desired expected average degree as input.

Moreover, we investigate the differences between *RHG*s and *GIRG*s, shedding new light on the nature of the relation between the two models. Although *RHG*s represent, in a certain sense, a special case of the *GIRG* model, we find that a straight-forward inclusion does not hold in practice. However, the difference is negligible for most use cases.

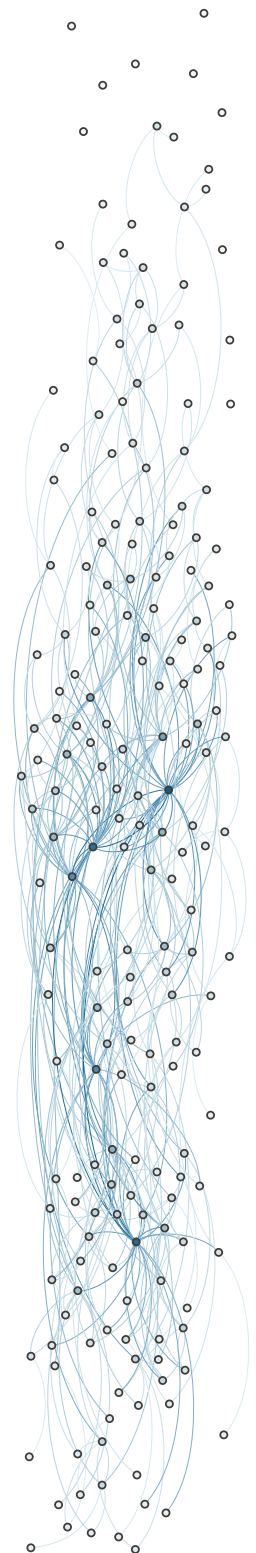
This chapter is based on the peer-reviewed conference article [48] (Best Paper ESA-B).

- [48] T. Bläsius, T. Friedrich, M. Katzmann, U. Meyer, M. Penschuck, and C. Weyand. Efficiently generating geometric inhomogeneous and hyperbolic random graphs. In M. A. Bender, O. Svensson, and G. Herman, editors, *European Symp. on Algorithms ESA*, volume 144 of *LIPICs*, pages 21:1–21:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.ESA.2019.21 .

My contribution

I contributed substantially to the engineering of *GIRG*s and *HYPERGIRG*s.

8



Binomial Random Hyperbolic Graph with $n = 200$, $\bar{d} = 8$, $\alpha = 1$, $T = 0.9$

8.1 Introduction

Network models play an important role in different fields of science [87]. From the perspective of network science, models can be used to explain observed behavior in the real world. To mention one example, Watts and Strogatz [342] observed that few random long-range connections suffice to guarantee a small diameter. This explains why many real-world networks exhibit the small-world property despite heavily favoring local over long-range connections. From the perspective of computer science, and specifically algorithmics, realistic random networks can provide input instances for graph algorithms. This facilitates theoretical approaches (e.g., average-case analysis), as well as extensive empirical evaluations by providing an abundance of benchmark instances, solving the pervasive scarcity of real-world instances.

There are some crucial features that make a network model useful. The generated instances have to resemble real-world networks. The model should be as simple and natural as possible to facilitate theoretical analysis, and to prevent atypical artifacts. And it should be possible to efficiently draw networks from the model. This is important for the empirical analysis of model properties and for generating benchmark instances.

A model that has proven itself useful in recent years is the *hyperbolic random graph* (*RHG*) model [200]. *RHG*s are generated by drawing vertex positions uniformly at random from a disk in the hyperbolic plane. Two vertices are joined by an edge if and only if their distance lies below a certain threshold; see Section 8.2.2. *RHG*s resemble real-world networks with respect to crucial properties. Most notable are the *powerlaw degree distribution* [159] (i.e., the number of vertices of degree k is roughly proportional to $k^{-\beta}$ with $\beta \in (2, 3)$), the high *clustering coefficient* [159] (i.e., two vertices are more likely to be connected if they have a common neighbor), and the small diameter [135, 256]. Moreover, *RHG*s are accessible for theoretical analysis (see, e.g., [159, 135, 256, 47]). Finally there is a multitude of efficient generators with different emphases [16, 223, 222, 224, 271, 139, 138]; see Section 8.1.2 for a discussion.

Closely related to *RHG*s is the *geometric inhomogeneous random graph* (*GIRG*) model [70]. Here every vertex has a position on the d -dimensional torus and a weight following a power law. Two vertices are then connected if and only if their distance on the torus is smaller than a threshold based on the product of their weights. When using positions on the circle ($d = 1$), *GIRG*s approximate *RHG*s in the following sense: the processes of generating a *RHG* and a *GIRG* can be coupled such that it suffices to decrease and increase the average degree of the *GIRG* by only a constant factor to obtain a subgraph and a supergraph of the corresponding *RHG*, respectively. Compared to *RHG*s, *GIRG*s are potentially easier to analyze, generalize nicely to higher dimensions, and the weights allow to directly adjust the degree distribution.

Above, we described the idealized *threshold variants* of the models, where two vertices are connected if and only if their distance is small enough. Arguably more realistic are the *binomial variants*, which allow longer edges and shorter non-edges with a small probability. This is achieved with an additional parameter T , called *temperature*. For $T \rightarrow 0$, the binomial and threshold variants coincide. Many publications focus on

the threshold case, as it is typically simpler. This is particularly true for generation algorithms: in the threshold variants one can ignore all vertex pairs with sufficient distance, which can be done using geometric data structures. In the binomial case, any pair of vertices could be adjacent, and the search space cannot be reduced as easily. For practical purposes, however, a non-zero temperature is crucial as real-world networks are generally assumed to have positive temperature. Moreover, from an algorithmic perspective, the threshold variants typically produce particularly well-behaved instances, while a higher temperature leads to difficult problem inputs. Thus, to obtain benchmark instances of varying difficulty, generators for the binomial variants are key.

8.1.1 Contribution and Outline

Based on the algorithm by Bringmann, Keusch, and Lengler [70], we provide an efficient *GIRG* generator. It includes the binomial case and allows higher dimensions. Its expected running time is linear. To the best of our knowledge, this is the first efficient generator for the *GIRG* model. Moreover, we adapt the algorithm to the *RHG* model, including the binomial variant. Compared to existing *RHG* generators (most of which only support the threshold variant), our implementation is the fastest sequential *RHG* generator.

Refactoring the original *GIRG* algorithm [70] allows us to parallelize our generators. They do not use multiple processors as effectively as *HYPERGEN* tailored towards parallelism. Still, we achieve comparable runtimes on commodity hardware (16 threads).

Our generators come as an open source C++ library¹ with documentation, command-line interface, unit tests, micro benchmarks, and OpenMP [267] parallelization using shared memory. An integration into NetworKit [316] is available.

Besides the efficient generators, we have three secondary contributions. (I) We provide a comprehensible description of the sampling algorithm that should make it easy to understand how the algorithm works, why it works, and how it can be implemented. Although the core idea of the algorithm is not new [70], the previous description is somewhat technical. (II) The expected average degree can be controlled via an input parameter. However, the dependence of the average degree on the actual parameter is non-trivial. In fact, given the average degree, there is no closed formula to determine the parameter. We provide a linear-time algorithm to estimate it. (III) We investigate how *GIRGs* and *RHG*s actually relate to each other by measuring how much the average degree of the *GIRG* has to be decreased and increased to obtain a subgraph and supergraph of the *RHG*, respectively.

In the following we first discuss our main contribution in the context of existing *RHG* generators. In Section 8.2, we formally define the *GIRG* and *RHG* models. Afterwards we describe the sampling algorithm in Section 8.3. In Section 8.4 we discuss implementation details, including the parameter estimation for the average degree (Section 8.4.1) as well as multiple performance improvements. Section 8.5 contains our experiments: we investigate the scaling behavior of our generator in Section 8.5.1, compare our *RHG* generator to existing ones in Section 8.5.2, and compare *GIRGs* to *RHG*s in Section 8.5.3.

¹<https://github.com/chistopher/girgs>

Table 8.1: Existing hyperbolic random graph generators. The columns show the names used throughout the paper; the conference appearance; a reference (journal if available); whether the generator supports the binomial model; and the asymptotic running time. The time bounds hold in the worst-case (wc), with high probability (whp), in expectation (exp), or empirically (emp).

Name	First Published	Ref.	Binom.	Running Time
<i>Pairwise</i>	CPC'15	[16]	✓	$\Theta(n^2)$ (wc)
RHGQUADTREE	ISAAC'15	[223]		$\mathcal{O}((n^{3/2} + m) \log n)$ (wc)
NKQUAD	IWOCA'16	[222]	✓	$\mathcal{O}((n^{3/2} + m) \log n)$ (wc)
NKBAND, NKBANDOPT	HPEC'16	[224]		$\mathcal{O}(n \log n + m)$ (emp)
GIRGS	ESA'16	[50]	✓	$\Theta(n + m)$ (exp)
HYPERGEN	SEA'17	[271]		$\mathcal{O}(n \log \log n + m)$ (whp)
RHG	IPDPS'18	[139]		$\Theta(n + m)$ (exp)
sRHG	JPDC'19	[138]		$\Theta(n + m)$ (exp)
HYPERGIRGS	this paper		✓	$\Theta(n + m)$ (exp)

8.1.2 Comparison with Existing Generators

We are not aware of previous *GIRG* implementations. Concerning *RHGs*, most algorithms only support the threshold case; see Table 8.1. The only published exceptions are the trivial quadratic algorithm [16], and an $\mathcal{O}((n^{3/2} + m) \log n)$ algorithm [222] based on a quad-tree data structure [223]. The latter is part of NetworKit; we call it NKQUAD. Moreover, the code for a hyperbolic embedding algorithm [50] includes an *RHG* generator implemented by Bringmann based on the *GIRG* algorithm [70]; we refer to it as GIRGS in the following.

GIRGS has been widely ignored as a high-performance generator. This is because it was somewhat hidden, and it is heavily outperformed by other threshold generators. Experiments show that our generator HYPERGIRGS is much faster than NKQUAD, which is to be expected considering the asymptotic running time. Moreover, on a single processor, we outperform GIRGS by an order of magnitude for $T = 0$ and by a factor of 4 for higher temperatures. As GIRGS does not support parallelization, this speedup increases for multiple processors.

For *Threshold RHGs*, there are the following generators. The quad-tree data structure mentioned above was initially used for the threshold generator RHGQUADTREE [223]. It was later improved leading to the algorithm currently implemented in NetworKit (NKBAND) [224]. A later re-implementation by Penshuck [271] improves it by about a factor of 2 (NKBANDOPT). However, the main contribution of [271] was a new generator that features sublinear memory and near optimal parallelization (HYPERGEN). Up to date, HYPERGEN was the fastest *Threshold RHG* generator on a single processor. Our generator, HYPERGIRGS, improves by a factor of 1.3 – 2 (depending on the parameters) but scales worse for more processors. Finally, Funke et al. [139] provide a generator designed for a distributed setting to generate enormous instances (RHG). Its runtime was later further reduced (sRHG) [138].

8.2 Models

8.2.1 Geometric Inhomogeneous Random Graphs

GIRGs [70] combine elements of random geometric graphs [149] and *Chung-Lu* graphs [92, 93]. Let $V = \{1, \dots, n\}$ be a set of vertices with positive weights w_1, \dots, w_n following a powerlaw with exponent $\beta > 2$. Let W be their sum. Let \mathbb{T}^d be the d -dimensional torus for a fixed dimension $d \geq 1$ represented by the d -dimensional cube $[0, 1]^d$ where opposite boundaries are identified. For each vertex $v \in V$, let $x_v \in \mathbb{T}^d$ be a point drawn uniformly and independently at random. For $x, y \in \mathbb{T}^d$ let $\|x - y\|$ denote the L_∞ -norm on the torus, i.e. $\|x - y\| = \max_{1 \leq i \leq d} \min\{|x_i - y_i|, 1 - |x_i - y_i|\}$. Two vertices $u \neq v$ are independently connected with probability p_{uv} . For a positive temperature $0 < T < 1$,

$$p_{uv} = \min \left\{ 1, c \left(\frac{w_u w_v / W}{\|x_u - x_v\|^d} \right)^{1/T} \right\} \quad (8.1)$$

while for $T = 0$ a threshold variant of the model is obtained with

$$p_{uv} = \begin{cases} 1 & \text{if } \|x_u - x_v\| \leq c(w_u w_v / W)^{1/d}, \\ 0 & \text{else.} \end{cases}$$

The constant $c > 0$ controls the expected average degree. We note that the above formulation slightly deviates from the original definition; see Section 8.2.3 for details.

8.2.2 Hyperbolic Random Graphs

*RHG*s [200] are generated by sampling random positions in the hyperbolic plane and connecting vertices that are close. More formally, let $V = \{1, \dots, n\}$ be a set of vertices. Let $\alpha > 1/2$ and $C \in \mathbb{R}$ be two constants, where α controls the powerlaw degree distribution with exponent $\beta = 2\alpha + 1 > 2$, and C determines the average degree \bar{d} . For each vertex $v \in V$, we sample a random point $p_v = (r_v, \theta_v)$ in the hyperbolic plane, using polar coordinates. Its angular coordinate θ_v is chosen uniformly from $[0, 2\pi)$ while its radius $0 \leq r_v < R$ with $R = 2 \log(n) + C$ is drawn according to the density function $f(r) = \frac{\alpha \sinh(\alpha r)}{\cosh(\alpha R) - 1}$. In the threshold case of *RHG*s two vertices $u \neq v$ are connected if and only if their distance is below R . The hyperbolic distance $d(p_u, p_v)$ is defined via $\cosh(d(p_u, p_v)) = \cosh(r_u) \cosh(r_v) - \sinh(r_u) \sinh(r_v) \cos(\theta_u - \theta_v)$.

The binomial variant adds a temperature $T \in [0, 1]$ to control the clustering, with lower temperatures leading to higher clustering. Two nodes $u, v \in V$ are then connected with probability $p_T(d(p_u, p_v))$ where $p_T(d) = (\exp[(d - R)/(2T)] + 1)^{-1}$. For $T \rightarrow 0$, the two definitions (threshold and binomial) coincide.

8.2.3 Comparison of *GIRGs* and *RHG*s

Bringmann et al. [70] show that the *RHG* model can be seen as a special case of the *GIRG* model in the following sense. Let d_{HRG} be the average degree of a *RHG*. Then there

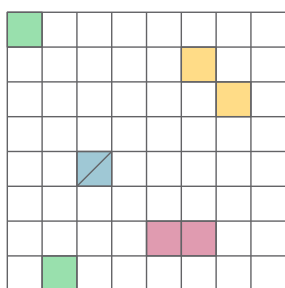


Figure 8.1: 2d grid for weight bucket pairs with a connection probability of $[2^{-3}, 2^{-4})$. Colored cell pairs represent neighbors. Note that the ground space is a torus and a cell is also a neighbor to itself.

exist *GIRGs* with average degree d_{GIRG} and D_{GIRG} with $d_{\text{GIRG}} \leq d_{\text{HRG}} \leq D_{\text{GIRG}}$ such that they are sub- and supergraphs of the *RHG*, respectively. Moreover, d_{GIRG} and D_{GIRG} differ only by a constant factor. Formally, this is achieved by using the big- O notation instead of a single constant c for the connection probability. We call this the *generic GIRG framework*. It essentially captures any specific model whose connection probabilities differ from Eq. (8.1) by only a constant factor. From a theoretical point of view this is useful as proving something for the generic *GIRG* framework also proves it for any manifestation, including *RHG*s.

To see how *RHG*s fit into the generic *GIRG* framework, consider the following mapping [70]. Radii are mapped to weights $w_v = e^{(R-r_v)/2}$, and angles are scaled to fit on a 1-dimensional torus $x_v = \theta_v/(2\pi)$. One can then see that the hyperbolic connection probability $p_T(d)$ under the provided mapping deviates from Eq. (8.1) by only a constant. Thus, c in Eq. (8.1) can be chosen such that all *GIRG* probabilities are larger or smaller than the corresponding *RHG* probabilities, leading to the two average degrees d_{GIRG} and D_{GIRG} mentioned above. Bringmann et al. [70] note that the two constants, which they hide in the big- O notation, do not have to match. They leave it open if they match, converge asymptotically, or how large the interval between them is in practice. We investigate this empirically in Section 8.5.3.

8.3 Sampling Algorithm

As mentioned in the introduction, the core of our sampling algorithm is based on the algorithm by Bringmann et al. [70]. In the following, we first give a description of the core ideas and then work out the details that lead to an efficient implementation.

To explain the idea, we make two temporary assumptions and relax them in Section 8.3.1 and Section 8.3.2, respectively. For now, assume that all weights are equal and consider only the threshold variant $T = 0$. The task is to find all vertex pairs that form an edge, i.e., their distance is below the threshold $c(w_u w_v / W)^{1/d}$. Since all weights are equal, the threshold in this restricted scenario is the same for all vertex pairs.

One approach to quickly identify adjacent vertices is to partition the ground space into a grid of cells. The size of the cells should be chosen, such that (I) the cells are as small as possible and (II) the diameter of cells is larger than the threshold $c(w_u w_v / W)^{1/d}$. The latter implies that only vertices in neighboring cells can be connected thus narrowing down the search space. The former ensures that neighboring cells contain as few vertex pairs as possible reducing the number of comparisons. Figure 8.1 shows an example of such a grid for a 2-dimensional ground space.

8.3.1 Inhomogeneous Weights

Assume that we have vertices with two different weights w_1, w_2 , rather than one. As before, the cells should still be as small as possible while having a diameter larger than the connection threshold. However, there are three different thresholds now, one for each combination of weights. To resolve this, we can group the vertices by weight and use three differently sized grids to find the edges between them.

As *GIRGs* require not only two but many weights, considering one grid for every weight pair is infeasible. The solution is to discretize the weights by grouping ranges of weights into *weight buckets*. When searching for edges between vertices in two weight buckets, the pair of largest weights in these buckets provides the threshold for the cell diameter. This choice of the cell diameter satisfies property (II). Property (I) is violated only slightly, if the weight range within the bucket is not too large. Thus, each combination of two weight buckets uses a grid of cells, whose granularity is based on the maximum weight in the respective buckets.

As a tradeoff, we choose $\lceil \log_2 n \rceil$ many buckets which yields a sublinear number of grids. Moreover, the largest and smallest weight in a bucket are at most a factor two apart. Thus, the diameter of a cell is too large by at most a factor of four.

With this approach, a single vertex has to appear in grids of different granularity. To do this in an efficient manner, we recursively divide the space into ever smaller grid cells, leading to a hierarchical subdivision of the space. This hierarchy is naturally described by a tree. For a 2-dimensional ground space, each node has four children, which is why we call it *quadtree*. Note that each level of the quadtree represents a grid of different granularity. Moreover, the side length of a grid cell on level ℓ is $2^{-\ell}$. For a pair (i, j) of weight buckets, we then choose the level that fits best for the corresponding weights, i.e., the deepest level such that the diameter of each grid cell is above the connection threshold for the largest weights in bucket i and j , respectively. We call this level the *comparison level*, denoted by $CL(i, j)$. It suffices to insert vertices of a bucket into the deepest level among all its comparison levels. This level is called the *insertion level* and we denote it by $I(i)$. In Section 8.3.4, we discuss in detail how to efficiently access all vertices in a given grid cell belonging to a given weight bucket.

8.3.2 Binomial Variant of the Model

For $T > 0$, neighboring cell pairs are still easy to handle: a constant fraction of vertex pairs will have an edge and one can sample them by explicitly checking every pair. For distant cell pairs and a fixed pair of weight buckets, the distance between the cells yields an upper bound on the connection probability of included vertices; see Eq. (8.1). The probability bound depends on both, the weight buckets and the cell pair distance, using the maximum weight within the buckets and the minimum distance between points in the cells. We note that the individual connection probabilities are only a constant factor smaller than the upper bound.

Knowing this, we can use geometric jumps to skip most vertex pairs [8]. The approach works as follows. Assume that we want to create an edge with probability \bar{p} for each vertex pair. For this process, we define the random variable X to be the number of vertex pairs we see until we add the next edge. Then X follows a geometric distribution. Thus, instead of throwing a coin for each vertex pair, we can do a single experiment that samples X from the geometric distribution and then skip X vertex pairs ahead. Since not all vertex pairs reach the upper bound \bar{p} , we accept encountered pairs with probability p_{uv}/\bar{p} to get correct results.

Although distant cell pairs are handled efficiently, their number is still quadratic,

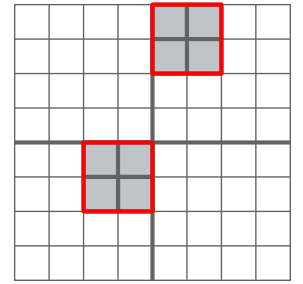


Figure 8.2: 2d grid for weight bucket pairs with a connection probability of $[2^{-3}, 2^{-4})$. The gray cells represent multiple distant cell pairs combined to one pair consisting of the red outlined parent cell pair.

Algorithm 11: Sample *GIRG* by Recursive Iteration of Cell Pairs

Input: cell pair (A, B) ; initially called with A, B set to the root of the quadtree

```

1 forall bucket pairs  $(i, j)$  that process the cell pair  $(A, B)$  do
2   if  $A$  and  $B$  are neighbors then
3     | emit each edge  $(u, v) \in V_i^A \times V_j^B$  with probability  $p_{uv}$ 
4   else
5     | choose candidates  $S \subseteq V_i^A \times V_j^B$  using geometric jumps and  $\bar{p}$ 
6     | emit each edge  $(u, v) \in S$  with probability  $p_{uv}/\bar{p}$ 
7 if  $A$  and  $B$  are neighbors and not maximum depth reached then
8   forall children  $X$  of  $A$  do
9     | forall children  $Y$  of  $B$  do
10    |   RECURSE( $X, Y$ )
    
```

most of which yield no edges. To circumvent this problem, the sampling algorithm, yet again, uses a quadtree. In the set of cell pairs to compare for one weight bucket pair, non-neighboring cells are grouped together along the quadtree hierarchy. They are replaced by their parents as shown in Figure 8.2 until their parents become neighbors.

In conclusion, for each pair of weight buckets (i, j) the following two types of cell pairs have to be processed. Any two neighboring cell pairs on the comparison level $CL(i, j)$; and any distant cell pair with level larger or equal $CL(i, j)$ that has neighboring parents. The resulting set of distant and neighboring cell pairs for a fixed bucket pair partitions $\mathbb{T}^d \times \mathbb{T}^d$.

8.3.3 Efficiently Iterating over Cell Pairs

The previous description sketches the algorithm as originally published. Here, we propose a refactoring that greatly simplifies the implementation and enables parallelization. We attribute a significant amount of *HYPERGIRGS*' speed up over *GIRGS* to this change.

Instead of first iterating over all bucket pairs and then over all corresponding cell pairs, we reverse this order. This removes the need to repeatedly determine the cell pairs to process for a given bucket pair. Instead it suffices to find the bucket pairs that process a given cell pair. This only depends on the level of the two cells and their type (neighboring or distant). Inverting the mapping from bucket pairs to cell pairs in the previous section yields the following. A neighboring cell pair on level ℓ is processed for bucket pairs with a comparison level of exactly ℓ . A distant cell pair on level ℓ (with neighboring parents) is processed for bucket pairs with a comparison level larger than or equal to ℓ . Thus, for each level of the quadtree we must enumerate all neighboring cell pairs, as well as distant cell pairs with neighboring parents. Algorithm 11 recursively enumerates exactly these cell pairs.

8.3.4 Efficient Access to Vertices by Bucket and Cell

A crucial part of the algorithm is to quickly access the set of vertices restricted to a weight bucket i and a cell A , which we denote by V_i^A . To this end, we linearize the cells of each level as illustrated in Figure 8.3. This linearization is called Morton code [251] or z-order curve [268]. It has the nice properties that (I) for each cell in level ℓ , its descendants in level $\ell' > \ell$ in the quadtree appear consecutively; and (II) it is easy to convert between a cell's position in the linear order and its d -dimensional coordinates (see Section 8.4.2).

We sort the vertices of a fixed weight bucket i by the Morton code of their containing cell on the insertion level $I(i)$, using arbitrary tie-breaking for vertices in the same cell. This has the effect that for any cell A with $\text{level}(A) \leq I(i)$, the vertices of V_i^A appear consecutive. Thus, to efficiently enumerate them, it suffices to know for each cell A the index of the first vertex in V_i^A . This can be precomputed using prefix sums leading to the following lemma.

Lemma 8.1. After linear preprocessing, for all cells A and weight buckets i with $\text{level}(A) \leq I(i)$, vertices in the set V_i^A can be enumerated in $\mathcal{O}(|V_i^A|)$. ◀

Proof. The proof is discussed in Section 8.A (Appendix). ◻

8.4 Implementation Details

The description in the previous section is an idealized version of the algorithm. For an actual implementation, there are some gaps to fill in. Omitting many minor tweaks, we want to sketch optimizations that are crucial to achieve a good practical runtime in the following and refer to Section 8.B (Appendix) for more details.

8.4.1 Estimating the Average Degree Parameter

Here, we sketch how to estimate the parameter c in Eq. (8.1) to achieve a given expected average degree. We estimate the constant based on the actual weights, not on their probability distribution. This leads to lower variance and allows user-defined weights.

We start with an arbitrary constant c , calculate the resulting expected average degree $\mathbb{E}[\bar{d}]$ and adjust c accordingly, using a modified binary search. This is possible, as $\mathbb{E}[\bar{d}]$ is monotone in c . We derive an exact formula for $\mathbb{E}[\bar{d}]$, depending on c and the weights. It cannot simply be solved for c , which is why we use binary search.

For the binary search, we need to evaluate $\mathbb{E}[\bar{d}]$ for different values of c . This is potentially problematic, as the formula for $\mathbb{E}[\bar{d}]$ sums over all vertex pairs. The issue preventing us from simplifying this formula is the minimum in the connection probability. Therefore, we first ignore the minimum and subtract an error term for those vertex pairs, where the minimum takes effect. The remaining hard part is to calculate this error term. Let E_S be the set of vertex pairs appearing in the error term and let S be the set of vertices with at least one partner in E_S .

	x	0	1
y	0	0	1
	1	2	3

	x	0	1	2	3
y	0	0	1	4	5
	1	2	3	6	7
	2	8	9	12	13
	3	10	11	14	15

Figure 8.3: Linearization of the cells on level 1 (top) and 2 (bottom) for $d = 2$.

Although $|E_S|$ itself is sufficiently small, S is too large to determine E_S by iterating over all pairs in $S \times S$. We solve this by iterating over the vertices in S , sorted by weight. Then, for each vertex we encounter, the set of partners in E_S is a superset of the partners of the previous vertex with smaller weight. This allows us to compute $\mathbb{E}[\bar{d}]$ in time $\mathcal{O}(S)$.

8.4.2 Efficiently Encoding and Decoding Morton Codes

Recall from Section 8.3.4 that we linearize the d -dimensional grid of cells using Morton code. As vertex positions are given as d -dimensional coordinates, we have to convert the coordinates to Morton codes (i.e., the index in the linearization) and vice versa. This is done by bitwise interleaving the coordinates.

For example, the 2-dimensional Morton code of the four-bit coordinates $a = a_3a_2a_1a_0$ and $b = b_3b_2b_1b_0$ is $a_3b_3a_2b_2a_1b_1a_0b_0$. We evaluated different encoding and decoding approaches via micro benchmarks. The fastest approach, at least on Intel processors, was an assembler instruction from BMI2 proposed by Intel in 2013 [184].

8.4.3 Generating RHGs Avoiding Expensive Mathematical Operations

The algorithm in Section 8.3 can be used to generate RHGs. It works conceptually the same, except that most formulas change. This has for example the effect that we no longer get a closed formula to determine the insertion level of a weight bucket or the comparison level of a bucket pair. Instead, one has to search them, by iterating over the levels of the quadtree. Further, RHGs introduce many computationally expensive mathematical operations like the hyperbolic cosine. This can be mitigated as follows.

For the threshold model, an edge exists if the distance d is smaller than R . Considering how the hyperbolic distance is defined (Section 8.2.2), reformulating it to $\cosh(d) < \cosh(R)$ avoids the expensive acosh , while $\cosh(R)$ remains constant during execution and can thus be precomputed. Similar to recent *Threshold RHG* generators, we compute intermediate values per vertex such that $\cosh(d)$ can be computed using only multiplication and addition [138, 271].

For the binomial model, evaluating the connection probability is a performance bottleneck. The straightforward way to sample edges is: compute the connection probability $p_T(d)$ depending on the distance, sample a uniform random value $u \in [0, 1]$, and create the edge if and only if $u < p_T(d)$. We can improve this by precomputing the inverse of $p_T(d)$ for equidistant values in $[0, 1]$. This lets us, for small ranges in $[0, 1]$, quickly access the corresponding range of distances. Changing the order, we first sample $u \in [0, 1]$, which falls in a range between two precomputed values, which in turn yields a range of distances. If the actual distance lies below that range, there has to be an edge and if it lies above, there is no edge. Only if it lies in the range, we actually have to compute the probability $p_T(d)$.

8.4.4 Parallelization

The algorithm has five steps: generate weights, generate positions, estimate the average degree constant, precompute the geometric data structure, and sample edges. The first two are trivial to parallelize. For estimating the constants, we parallelize the dominant computations with linear running time. To sample the edges, we make use of the fact that we iterate over cell pairs in a recursive manner. This can be parallelized by cutting the recursion tree at a certain level and distributing the independent subproblems among multiple processors.

For the preprocessing we have to do three subtasks: compute for each vertex its containing cells on its insertion level, sort the vertices according to their Morton code index, and compute the prefix sum for all cells. We parallelize all three tasks and optimize them by handling all weight buckets together, sorting by weight bucket first and Morton code second. This is done by encoding this criterion into integers that are sorted with parallel radix sort.

8.5 Experimental Evaluation

We perform three types of experiments. In Section 8.5.1 we investigate the scaling behavior of our *GIRG* generator, broken down into the different tasks performed by the algorithm. In Section 8.5.2 we compare our *RHG* generator with existing generators. In Section 8.5.3 we experimentally investigate the difference between *RHGs* and their *GIRG* counterpart. Whenever a data point represents the mean over multiple iterations, our plots include error bars that indicate the standard deviation. Besides the implementation itself, all benchmarks and analysis scripts are also accessible in our source repository.

8.5.1 Scaling of the GIRG Generator

We investigate the scaling of the generator, broken down into five steps.

Weights: Generate powerlaw weights.

Positions: Generate points on \mathbb{T}^d .

Binary: Estimate the constant controlling the average degree.

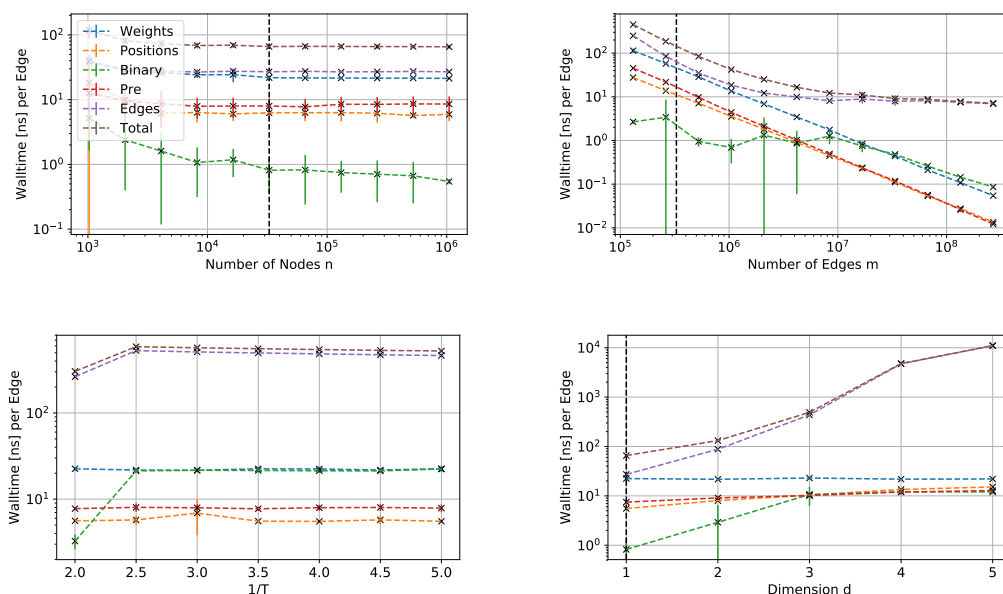
Pre: Preprocess the geometric data structure (Section 8.3.4).

Edges: Sample edges between all vertex pairs as described in Algorithm 11.

Figure 8.4 shows the sequential runtime over the number of nodes n (top left), number of edges m (top right), temperature T (bottom right), and dimension d (bottom right). The performance is measured in nanoseconds per edge. Each data point represents the mean over 10 iterations. To make the measurements independent of the graph representation, we do not save the edges, but accumulate a checksum instead. Note that the top right plot increases the average degree, resulting in a decreased time per edge.

The empirical runtimes match the theoretical bounds: it is linear in n and m , grows exponentially in the dimension d , and is unaffected by the temperature T . The overall time is dominated by the edge sampling. Generating the weights includes expensive

Figure 8.4: Sequential runtime for the steps of the GIRG sampling algorithm averaged over 10 iterations. Each plot varies a different model parameter deviating from a base configuration $d = 1$, $n = 2^{15}$, $T = 0$, $\beta = 2.5$, and $\bar{d} = 10$. The base configuration is indicated by a dashed vertical line.



exponential functions, making it the slowest step after edge sampling. Generating the positions is significantly faster even for higher dimensions. For the parameter estimation using binary search, one can see that the runtime never exceeds the time to generate the weights. For non-zero temperature T the performance of the binary search is similar to the generation of the weights, as it also requires exponential functions. The lower runtimes per edge for the increasing number of edges (top right) show that the runtime is dominated by the number of nodes n . Only for very high average degrees, the cost per edge outgrows the cost per vertex.

8.5.2 RHG Runtime Comparison

We evaluate the runtime performance of HYPERGIRGS compared to the generators in Table 8.1, excluding the generators with high asymptotic runtime as well as RHG and sRHG. Both are designed for distributed machines. Executed on a single compute node, the performance of the faster sRHG is comparable to HYPERGEN [138]. To avoid systematic biases between different graph representations, the implementations are modified² not to store the resulting graph. Instead, only count the edges produced and we ensure that the computation of incident nodes is not optimized away by the compiler.

We used different machines for our sequential and parallel experiments. The former are done on an *Intel Core i7-8700K* with 16 GB RAM, the latter on an *Intel Xeon CPU E5-2630 v3* with 8 cores (16 threads) and 64 GB RAM.

Our generator HYPERGIRGS is consistently faster than the competitors, independent of the parameter choices; see figures 8.5a and 8.5b. Only for unrealistic high average degrees of 1000, HYPERGEN slightly outperforms HYPERGIRGS.

Moreover, HYPERGIRGS beats GIRGS, the only other efficient generator supporting non-zero temperature, by an order of magnitude. For higher temperatures, we compare

²The modifications are publicly available and referenced in our GitHub repository.

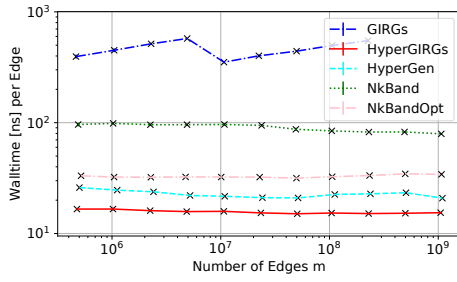
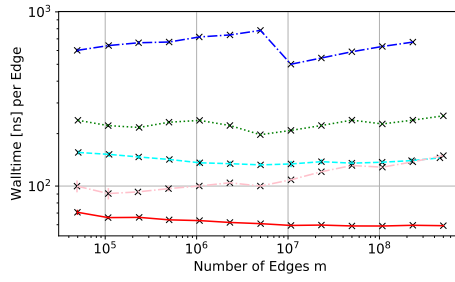
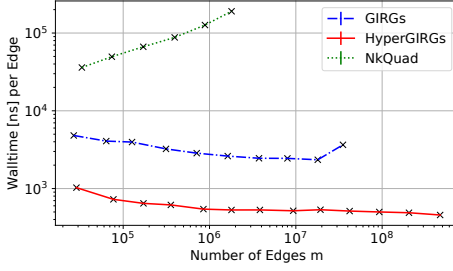
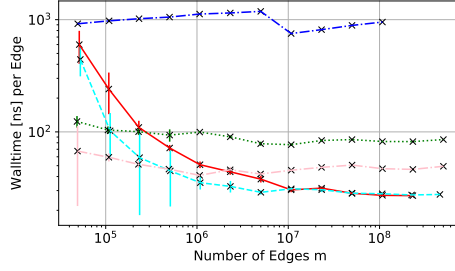

 (a) $\bar{d} = 100, \beta = 2.2, T = 0$, sequential

 (b) $\bar{d} = 10, \beta = 3, T = 0$, sequential

 (c) $\bar{d} = 10, \beta = 2.2, T = 0.5$, sequential

 (d) $\bar{d} = 10, \beta = 3, T = 0$, parallel (16 threads)

Figure 8.5: Comparison of RHG generators averaged over 5 iterations.

(a) and (b): Threshold variant for different average degrees \bar{d} and powerlaw exponents β .

(c): Binomial variant, temperature $T = 0.5$.

(d): The same configuration as (b) but utilizing multiple cores.

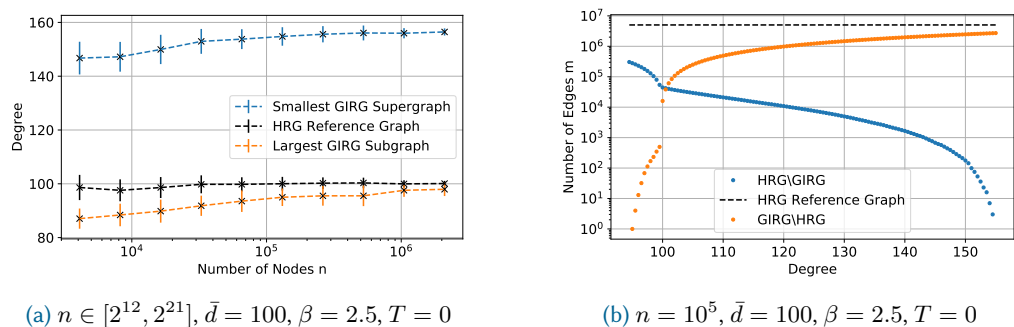
our algorithm with the two other non-quadratic generators NkQUAD (included in *NetworKit*) and GIRGs; see Figure 8.5c. We note that GIRGs uses a different estimation for R , which leads to an insignificant left-shift of the corresponding curve. In Figure 8.5c, one can clearly see the worse asymptotic running time of NkQUAD. Compared to GIRGs, HYPERGIRGS is consistently four times faster.

Figure 8.5d shows measurements for parallel experiments using 16 threads. The parameters coincide with Figure 8.5b. GIRGs does not support parallelization and is outperformed even more. For sufficiently large graphs, the fastest generator in this multi-core setting is HYPERGEN, which is specifically tailored towards parallel execution. Nonetheless, HYPERGIRGS shows comparable performance and overtakes the other two generators NkBAND and NkBANDOPT. We note that even on parallel machines, the sequential performance is of high importance: one often needs a large collection of graphs rather than a single huge instance. In this case, it is more efficient to run multiple instances of a sequential generator in parallel.

8.5.3 Difference Between RHGs and GIRGs

Recall from Section 8.2.3 that a RHG with average degree d_{HRG} has a corresponding GIRG sub- and supergraphs with average degrees d_{GIRG} and D_{GIRG} , respectively.

We experimentally determine, for given RHGs, the values for d_{GIRG} by decreasing the average degree of the corresponding GIRGs until it is a subgraph of the RHG. Analogously, we determine the value for D_{GIRG} . We focus on the threshold variant of the models, as this makes the coupling between RHGs and GIRGs much simpler (the graph is uniquely determined by the coordinates). Figure 8.6a shows d_{GIRG} and D_{GIRG} , compared to d_{HRG} for growing n . One can see that d_{GIRG} and D_{GIRG} are actually quite


 (a) $n \in [2^{12}, 2^{21}]$, $\bar{d} = 100$, $\beta = 2.5$, $T = 0$

 (b) $n = 10^5$, $\bar{d} = 100$, $\beta = 2.5$, $T = 0$

Figure 8.6: Relation between the *RHG* and the *GIRG* model. **(a)** The values for d_{HRG} , d_{GIRG} , D_{GIRG} averaged over 50 iterations. **(b)** The number of missing ($\text{RHG} \setminus \text{GIRG}$) and additional ($\text{GIRG} \setminus \text{RHG}$) edges depending on the expected degree of the corresponding *GIRG*. It can be interpreted as a cross-section of one iteration in (a).

far apart. They in particular do not converge to the same value for growing n . However, at least d_{GIRG} seems to approach d_{HRG} . This indicates that every *RHG* corresponds to a *GIRG* subgraph that is missing only a sublinear fraction of edges. On the other hand, the average degree of the *GIRG* has to be increased by a lot to actually contain all edges also contained in the *RHG*.

Figure 8.6b gives a more detailed view for a single *RHG*. Depending on the average degree of the *GIRG*, it shows how many edges the *GIRG* lacks and how many edges the *GIRG* has in addition to the *RHG*. For degree 100, the *GIRG* contains about 38 k additional and lacks about 42 k edges. These are rather small numbers compared to the 50 M edges of the graphs.

Appendix 8.A Omitted Proof of Lemma 8.1

Lemma 8.2. (Restated Lemma 8.1) After linear preprocessing, for all cells A and weight buckets i with $\text{level}(A) \leq I(i)$, vertices in the set V_i^A can be enumerated in $\mathcal{O}(|V_i^A|)$. ◀

Proof. As mentioned above, we have to sort the vertices V_i of each weight bucket i according to the index (Morton code) of the containing cell. Clearly, the d -dimensional coordinates of the cell containing a given vertex is obtained in constant time by rounding. From this one can obtain the index in constant time (also see Section 8.4.2). This can be done using, e.g., bucket sort with respect to this index to sort the vertices. In the following, we refer to this sorted array with V_i .

Besides these sorted arrays V_i of vertices, one for each weight bucket i , we store for each cell C at level $I(i)$ the number of vertices preceding the vertices in cell C . Note that this is simply the prefix sum of the number of vertices in all cells that come before cell C . Denote this prefix sum of cell C with P_C .

Now let i be a weight bucket and let A be a cell identifying the requested set of vertices V_i^A (with $\text{level}(A) \leq I(i)$). Let C_1, \dots, C_j be the descendants of cell A at level $I(i)$, appearing in this order according to the Morton code.

Recall that the vertices in C_1, \dots, C_j appear consecutive in the sorted array V_i . Thus, V_i^A is given by the range $[P_{C_1}, \dots, P_{C_{j+1}})$ in V_i .

In terms of running time, each weight bucket requires $\mathcal{O}(|V_i| + 2^{d-I(i)})$ time for bucket sort and $\mathcal{O}(2^{d-I(i)})$ time for the prefix sums, where $2^{d-I(i)}$ is the number of cells in the insertion level $I(i)$. Over all weight buckets, the term $|V_i|$ sums up to $|V|$ and Bringmann et al. [70] show that the same holds for $2^{d-I(i)}$. \square

Appendix 8.B Implementation Details

8.B.1 Avoiding Double Counting Buckets, Cells, and Vertices

The algorithm as described in Section 8.3 iterates over pairs of buckets, cells, and vertices. All three entities need to be handled correctly to avoid visiting vertex or cell pairs multiple times. Consider the recursive Algorithm 11. In lines 8 and 9, it is sufficient to consider only cell pairs with $A \leq B$, because the pairs (A, B) and (B, A) can be handled simultaneously. Meaning, if a cell pair is processed by the bucket pair (i, j) , then it needs to be processed by the bucket pair (j, i) (cf. line 1). However, the bucket pair (i, i) should occur once per cell pair. Alternatively, one can separate the cell pairs (A, B) and (B, A) , but instead consider only bucket pairs (i, j) with $i \leq j$. In any case, bucket pairs (i, i) require special treatment for cell pairs of the form (A, A) . Then, only edges between vertices $u < v$ should be checked (lines 3,5-6). If self-loops are desired, the constraint can be relaxed to $u \leq v$.

8.B.2 Estimating the Average Degree Parameter

This section covers the estimation for the binomial version of the model $T > 0$. The calculations for the threshold case $T = 0$ are analogous (and simpler).

Typically, a random graph generator accepts the expected average degree or the number of edges as an input parameter. In the following we describe how binary search can be used to estimate the constant c in the edge probability p_{uv} (Eq. (8.1)) for a desired expected average degree. The constant is found based on the actual weights instead of their probability distribution, because the resulting average degree has lower variance and the generator should be able to accept user-defined weights as well. Note that we implement *GIRGs* without explicitly modeling the constant c , because scaling all weights by c^T emulates the same behavior.

Let X_{uv} be a random indicator variable for the existence of edge uv .

$$\begin{aligned} \mathbb{E}[X_{uv}] &= \mathbb{E} \left[\min \left\{ 1, c \left(\frac{w_u w_v / W}{\|x_u - x_v\|^d} \right)^{1/T} \right\} \right] \\ &= \mathbb{E} \left[\min \left\{ 1, \left(\frac{c^{\frac{T}{d}} (w_u w_v)^{\frac{1}{d}}}{\|x_u - x_v\|} \right)^{d/T} \right\} \right] \end{aligned}$$

Let $k = c^{T/d} (w_u w_v / W)^{1/d}$. To remove the minimum, we distinguish between *short edges* that are guaranteed to exist and *long edges* that exist with probability below 1.

$$\begin{aligned} \mathbb{E}[X_{uv}] &= \mathbb{P}[\|x_u - x_v\| \leq k] \\ &+ \mathbb{P}[k < \|x_u - x_v\|] \cdot \mathbb{E} \left[c \left(\frac{w_u w_v / W}{\|x_u - x_v\|^d} \right)^{1/T} \mid k < \|x_u - x_v\| \right] \end{aligned}$$

If $k \geq 0.5$ then the weights guarantee the existence of the edge uv independent of position. For simplicity we assume that $k \leq 0.5$ for all vertex pairs. In the end we subtract an error to account for the ignored pairs. For any constant $t \leq 0.5$, $\mathbb{P}[\|x_u - x_v\| \leq t] = (2t)^d$, which is the fraction of the ground space which is covered by a hypercube with radius t . The probability for a short edge becomes

$$\mathbb{P}[\|x_u - x_v\| \leq k] = (2k)^d = 2^d c^T \left(\frac{w_u w_v}{W} \right). \quad (8.2)$$

The probability density function of $\|x_u - x_v\|$ between 0 and 0.5 is the derivative of $(2x)^d$, namely $d2^d x^{d-1}$. We calculate the probability for a long edge based on two specific weights.

$$\begin{aligned} &\mathbb{P}[k < \|x_u - x_v\|] \cdot \mathbb{E} \left[c \cdot \left(\frac{w_u w_v / W}{\|x_u - x_v\|^d} \right)^{1/T} \mid k < \|x_u - x_v\| \right] \\ &= \mathbb{P}[k < \|x_u - x_v\|] \cdot \frac{\int_k^{0.5} c \cdot \left(\frac{w_u w_v / W}{x^d} \right)^{1/T} \cdot d2^d x^{d-1} dx}{\mathbb{P}[k < \|x_u - x_v\|]} \\ &= c \left(\frac{w_u w_v}{W} \right)^{1/T} d2^d \int_k^{0.5} x^{d-1-d/T} dx \\ &= c \left(\frac{w_u w_v}{W} \right)^{1/T} d2^d \left[\frac{1}{d(1-1/T)} \cdot x^{d-d/T} \right]_k^{0.5} \\ &= c \left(\frac{w_u w_v}{W} \right)^{1/T} \frac{d2^d}{d(1-1/T)} \left(\left(\frac{1}{2} \right)^{d-d/T} - k^{d(1-1/T)} \right) \\ &= c \left(\frac{w_u w_v}{W} \right)^{1/T} \frac{2^d}{1-1/T} \left(\frac{2^{d/T}}{2^d} - \left(c^{T/d} \left(\frac{w_u w_v}{W} \right)^{1/d} \right)^{d(1-1/T)} \right) \\ &= c \left(\frac{w_u w_v}{W} \right)^{1/T} \frac{2^{d/T}}{1-1/T} - c \left(\frac{w_u w_v}{W} \right)^{1/T} \frac{2^d}{1-1/T} c^{T-1} \left(\frac{w_u w_v}{W} \right)^{1-1/T} \\ &= c \left(\frac{w_u w_v}{W} \right)^{1/T} \frac{2^{d/T}}{1-1/T} - c^T \left(\frac{w_u w_v}{W} \right) \frac{2^d}{1-1/T} \end{aligned} \quad (8.3)$$

We add short and long edges (Eq. (8.2) and Eq. (8.3)).

$$\begin{aligned}
 \mathbb{E}[X_{uv}] &= 2^d c^T \left(\frac{w_u w_v}{W} \right) + c \left(\frac{w_u w_v}{W} \right)^{1/T} \frac{2^{d/T}}{1 - 1/T} - c^T \left(\frac{w_u w_v}{W} \right) \frac{2^d}{1 - 1/T} \\
 &= 2^d c^T \left(\frac{w_u w_v}{W} \right) \left(1 + \frac{1}{1/T - 1} \right) - c \left(\frac{w_u w_v}{W} \right)^{1/T} \frac{2^{d/T}}{1/T - 1} \\
 &= c^T \frac{2^d}{1 - T} \left(\frac{w_u w_v}{W} \right) - c \frac{2^{d/T}}{1/T - 1} \left(\frac{w_u w_v}{W} \right)^{1/T}
 \end{aligned} \tag{8.4}$$

The expected average degree $\mathbb{E}[\bar{d}]$ is computed as follows.

$$\mathbb{E}[\bar{d}] \cdot n = c^T \frac{2^d}{1 - T} \sum_{u \in V} \sum_{v \neq u} \left(\frac{w_u w_v}{W} \right) - c \frac{2^{d/T}}{1/T - 1} \sum_{u \in V} \sum_{v \neq u} \left(\frac{w_u w_v}{W} \right)^{1/T} \tag{8.5}$$

We compute the sums for all vertex pairs by subtracting an error for the reflexive edges.

$$\begin{aligned}
 \sum_{u \in V} \sum_{v \neq u} \left(\frac{w_u w_v}{W} \right) &= \sum_{u \in V} \sum_{v \in V} \frac{w_u w_v}{W} - \sum_{v \in V} \frac{w_v^2}{W} = W - \sum_{v \in V} \frac{w_v^2}{W} \\
 \sum_{u \in V} \sum_{v \neq u} \left(\frac{w_u w_v}{W} \right)^{1/T} &= \sum_{u \in V} \left(\frac{w_u^{1/T}}{W^{1/T}} \sum_{v \in V} w_v^{1/T} \right) - \sum_{v \in V} \left(\frac{w_v^2}{W} \right)^{1/T}
 \end{aligned}$$

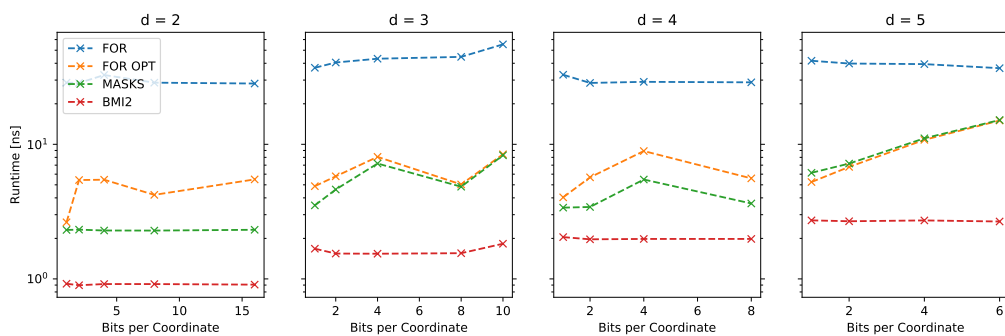
We earlier assumed $k \leq 0.5$ and still have to subtract an error for the ignored vertex pairs. Let E_R be the set of vertex pairs (u, v) with $0.5 < k = c^{\frac{T}{d}} \left(\frac{w_u w_v}{W} \right)^{1/d}$ and let R be the set of vertices with at least one edge in E_R . For each vertex pair in E_R , the probability for a short edge is 1 instead of what we got in Eq. (8.2) and the probability for a long edge is 0 instead of Eq. (8.3). Therefore, the error due to an edge $(u, v) \in E_R$ can be obtained by subtracting 1 from Eq. (8.4).

Now we are ready to find the constant c for a desired average degree using binary search over the monotone function $f(c) = \mathbb{E}[\bar{d}]$. The function f is given by Eq. (8.5) substituting the sums and subtracting the error for vertex pairs with $k > 0.5$.

$$\begin{aligned}
 f(c) &= c^T \cdot \frac{2^d}{n(1 - T)} \left(W - \sum_{v \in V} \frac{w_v^2}{W} \right) \\
 &\quad - c \cdot \frac{2^{d/T}}{n(1/T - 1)} \left(\sum_{u \in V} \left(\frac{w_u^{1/T}}{W^{1/T}} \sum_{v \in V} w_v^{1/T} \right) - \sum_{v \in V} \left(\frac{w_v^2}{W} \right)^{1/T} \right) \\
 &\quad - \frac{1}{n} \sum_{(u,v) \in E_R} \left(c^T \frac{2^d}{1 - T} \left(\frac{w_u w_v}{W} \right) - c \frac{2^{d/T}}{1/T - 1} \left(\frac{w_u w_v}{W} \right)^{1/T} - 1 \right)
 \end{aligned}$$

The binary search takes $\mathcal{O}(n)$ time to compute various sums and $\mathcal{O}(1 + |E_R|)$ per evaluation of $f(c)$. We partially sort the weights for all vertices in R to iterate efficiently over E_R . Since the upper and lower bound for the binary search are found with an exponential search, the size of R might grow until the upper bound is found. We lazily extend a sorted prefix of the weight array while raising the upper bound.

Figure 8.7: Performance of Morton code generation in dimensions 2 to 5 on an Intel processor. Input coordinates are limited to $\lfloor 32/d \rfloor$ bits each, because the result is saved as a 32 bit integer.



The time to evaluate $f(c)$ can be quadratically reduced from $\mathcal{O}(|E_R|)$ to $\mathcal{O}(|R|)$ by exploiting that for any two vertices u and v with $w_u \leq w_v$, a pair $(u, x) \in E_R$ implies $(v, x) \in E_R$. Thus, if we iterate the vertices in R by increasing weight, we can reuse the computations for the last vertex.

8.B.3 Efficiently Encoding and Decoding Morton Codes

Recall from Section 8.3.4 that we linearize the d -dimensional grid of cells using Morton code. As vertex positions are given as d -dimensional coordinates, we have to convert the coordinates to Morton codes (i.e., the index in the linearization) and vice versa. A Morton code is obtained by bitwise interleaving two or more coordinates. E.g., the 2-dimensional Morton code of the four-bit coordinates $a = a_3a_2a_1a_0$ and $b = b_3b_2b_1b_0$ is $a_3b_3a_2b_2a_1b_1a_0b_0$. Implementation-wise, there are the following encoding approaches.

FOR, FOR OPT Set each bit of the result with shifts and bitwise operations (FOR). Since we know the level of a cell, we know the number of relevant bits in each coordinate. Considering only relevant bits improves performance significantly (FOR OPT).

MASKS For details on this method, we refer to the open source library *libmorton* [31] and the author's related blog posts³. The approach is hard to generalize to multiple dimensions.

LUT A lookup table computed at compile time⁴ can be used. The input is divided into chunks; a precomputed result for each chunk is obtained and shifted into place.

BMI2 The *Parallel Bits Deposit/Extract* assembler instructions from Intel's Bit-Manipulation-Instruction-Set 2 [184] provide a solution with one assembler instruction per input coordinate. BMI2 is available on Intel CPUs since 2013 and supported by recent AMD CPUs (Zen).

All approaches except LUT support a complementary decoding operation. We measured the approaches, excluding LUT, on an Intel i7-8550U processor (see Figure 8.7)

³<https://www.forceflow.be>

⁴<https://github.com/kevinhartman/morton-nd>

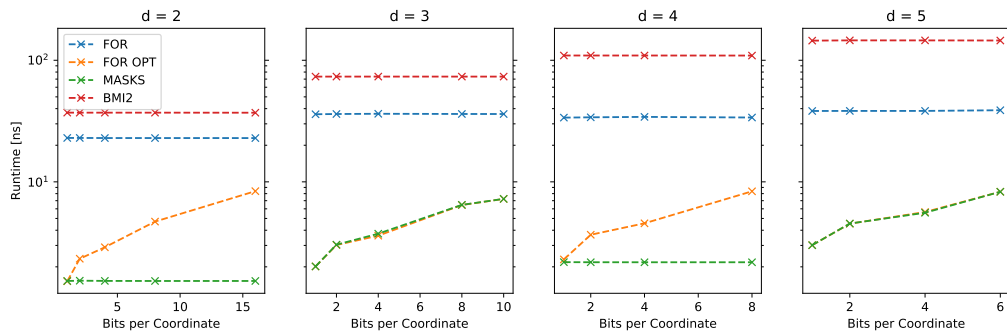


Figure 8.8: Performance of Morton code generation in dimensions 2 to 5 on an AMD processor. Input coordinates are limited to $\lfloor 32/d \rfloor$ bits each, because the result is saved as a 32 bit integer.

and an AMD Ryzen7-2700X (see Figure 8.8). On Intel, BMI2 is consistently the fastest and at least an order of magnitude faster than FOR. Surprisingly, FOR OPT is not monotone in the number of bits per coordinate for dimensions below 5. Inspection of the generated assembly⁵ reveals that the compiler employed SIMD instructions. On AMD, BMI2 is the slowest. Our GIRG generator uses BMI2 if enabled and the loop with early termination (FOR OPT) otherwise.

8.B.4 Avoiding Computationally Expensive Math for RHGs

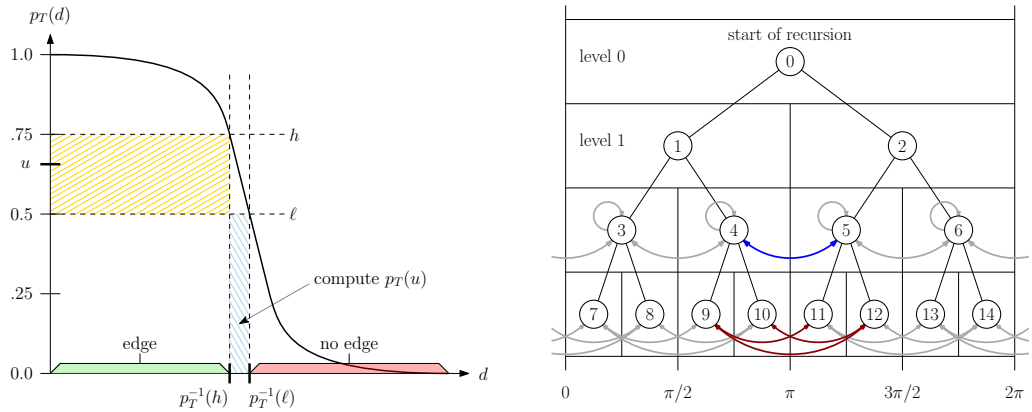
The RHG model requires many computationally expensive mathematical operations. We significantly improve the performance of the generator by avoiding or reusing the results of those operations. The first optimization applies to the threshold variant and the second optimization to the binomial version.

For the threshold model, an edge exists if the distance d is smaller than R . Considering how the hyperbolic distance is defined (Section 8.2.2), reformulating it to $\cosh(d) < \cosh(R)$ avoids the expensive acosh , while $\cosh(R)$ remains constant during execution. Similar to recent *Threshold RHG* generators, we compute intermediate values per vertex such that $\cosh(d)$ can be computed using only multiplication and addition [138, 271].

For the binomial model, evaluating the connection probability $p_T(d)$ from the optimized $\cosh(d)$ is a performance bottleneck and made up half of the total runtime. Evaluating $p_T(d)$ includes an expensive exponential function and cannot avoid the acosh like in the threshold model. We introduce a distance filter (see Figure 8.9a) to reduce the frequency of the operation resulting in a speedup of approximately factor two. The process before was: compute the probability $p_T(d)$, sample a uniform random value $u \in [0, 1]$, and emit an edge if $u < p_T(d)$. The idea of our filter is that we invert the probability function and compute $p_T^{-1}(d)$ in advance for multiple equidistant values between 0 and 1. Each entry $x \mapsto p_T^{-1}(x)$ in the filter represents the distance — or rather proximity — needed for an edge probability of x . During edge sampling, we generate $u \in [0, 1]$ before evaluating p_T . The generated u falls in an interval between two precomputed entries $l \leq u < h$ in our filter. We know that p_T is monotonically decreasing so $p_T^{-1}(l) \geq p_T^{-1}(u) > p_T^{-1}(h)$, meaning the higher the distance the lower

⁵g++8 -std=c++14 -O3 -march=skylake

Figure 8.9: Distance Filter (left) and tasks for parallelization in the 1-dimensional case (right).



(a) Sketch of the distance filter optimization to avoid computationally expensive mathematical operations, providing a x2 speedup.

(b) Visited cell pairs up to level 3. The arrows represent the 8 neighboring cell pairs in level 2 and 12 distant cell pairs in level 3.

the probability and vice versa. Instead of emitting an edge if and only if $u < p_T(d)$, we emit an edge if $p_T^{-1}(h) \geq d$ and skip the edge if $p_T^{-1}(\ell) \leq d$. Only if $p_T(d)$ is in the interval between ℓ and h , the expensive $p_T(d)$ has to be evaluated. Since u is uniformly distributed, the probability to hit the interval where $p_T(d)$ has to be evaluated is $1/(k-1)$, where k is the number of entries in the filter. Our generator uses $k = 100$. Additionally, we avoid the acosh by directly storing $\cosh[p_T^{-1}(x)]$ in the filter.

8.B.5 Parallelization

This section describes how the sampling algorithm can be parallelized focusing on the preprocessing building the geometric data structure (Section 8.3.4) and on the recursion enumerating pairs of grid cells for sampling the edges (Section 8.3.3). The presented approach applies to the GIRG and RHG implementations.

The preprocessing for a weight bucket i computes the containing cell on the insertion level for all vertices in V_i . We optimized the process by processing all weight buckets together. The containing cell for all vertices in V is computed in parallel. We sort vertices by weight bucket first and by cell second. Theoretically, bucket sort results in linear runtime. For the implementation, however, we use a parallel radix sort instead. The vertices V_i of weight bucket i form a contiguous subsequence in V . Moreover, V_i is sorted by cell, allowing parallel computation of the prefix sums for all cells in the insertion level of the weight bucket.

The recursion is executed in parallel and experiments suggest a near optimal scaling when the number of threads is a power of two. Each thread has a local random generator. We use static scheduling to produce deterministic results even for the binomial model. However, the ordering of edges in the edge list varies, because each thread locally buffers generated edges before writing them while locking a mutex. We distinguish two stages of execution. The first stage is to “saw off” the recursion tree at a certain level and collect the omitted recursive calls as *tasks* to execute in stage two. A task is represented by a cell pair from which to pick up the execution later. One thread collects

the tasks by traversing the recursion tree without sampling any edges (omitting lines 1-6 in Algorithm 11). Meanwhile, the other threads process the pairs that the main thread passed through. When all tasks are collected stage two begins. In stage two, the threads pick up the “loose ends” of the cut recursion tree. There are three different types of tasks with varying load. For 1-dimensional geometry, level $\ell > 2$, and assuming a number of threads that is constant in n , the types of tasks are the following. There are 2^ℓ *heavy tasks* given by a neighboring cell pair of the form (A, A) . Their number of recursive calls grows exponentially with each subsequent level implying a load of $\mathcal{O}(n)$. There are 2^ℓ *light tasks* given by a neighboring cell pair of the form $(A, A + 1)$. They produce four recursive calls per subsequent level implying a load of $\mathcal{O}(\log n)$. Finally, there are $3 \cdot 2^{\ell-1}$ *constant tasks* given by a distant cell pair. They invoke no recursive calls at all. The number of distant cell pairs in a level is explained by Figure 8.9b. For each cell B in level $\ell - 1$ with children A and $A + 1$, the distant cell pairs in level ℓ are $(A, A + 2)$, $(A, A + 3)$, $(A + 1, A + 3)$.

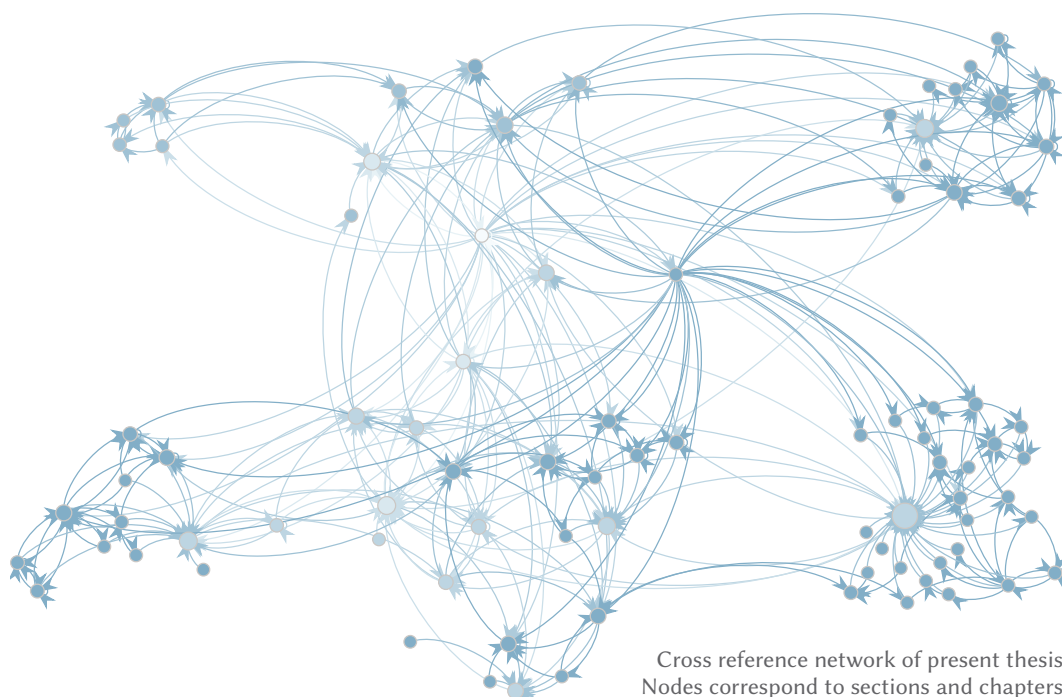
Since heavy tasks dominate the runtime during stage two, we distribute heavy tasks evenly among all threads. This is why the approach scales best when the number of threads is a power of two. The level where we saw off the recursion tree is a tuning parameter of the generator. We choose it, such that there are two heavy tasks per thread to reduce load imbalance if one thread stalls. To apply the same scheduling approach to higher dimensions it suffices to know that the load of tasks remains similar and the number of heavy tasks is $2^{\ell d}$.

Summary

The present thesis considers algorithmic aspects of generating massive random graphs. We develop efficient sampling algorithms for the following commonly used stochastic processes and network models.

- *Barabási-Albert Graphs, Preferential Attachment* [☞ Chapter 3](#)
- *LFR Graphs, Edge Switching, Configuration Model* [☞ Chapter 4](#)
- *Curveball, Global Curveball* [☞ Chapter 5](#)
- *Threshold Random Hyperbolic Graphs* [☞ Chapters 6 and 7](#)
- *Binomial Hyperbolic and Geometric Inhomogenous Random Graphs* [☞ Chapter 8](#)

In this chapter, we summarize our results obtained for various machine models ranging from sequential schemes over I/O-efficient algorithms to parallel solutions, including distributed computing and different types of shared-memory parallelism.



9.1 Preferential Attachment

Chapter 3 is based on [239] and introduces the first two I/O-efficient sampling approaches for random graph models based on preferential attachment. We initially focus on *Barabási-Albert (BA)* graphs to demonstrate the techniques and subsequently discuss additional features such as seed graphs exceeding main memory, vertices with inhomogeneous initial degrees, the inclusion of uniform node sampling, directed graphs, and edges between two randomly chosen vertices.

further applications:
 Section 3.6

The main algorithmic challenge of *BA* lies in its iterative graph growing process that adds one node per time step. Each new node is directly connected to the existing graph by randomly drawing neighbors weighted by their current degrees. Batagelj and Brandes observe for their internal memory generator BB-BA that the underlying dynamic weighted sampling problem can be reduced to uniformly selecting entries from the edge list produced so far.

As BB-BA requires unstructured I/Os, it cannot efficiently produce graphs that do not fit into main memory. We therefore propose the two algorithms MP-BA and TFP-BA:

TFP-BA:
 Section 3.2

- TFP-BA is a simple and easily generalizable sequential generator inspired by BB-BA. Rather than reading from random positions in the edge list, TFP-BA first precomputes all necessary read operations and sorts them by the memory address they read from. As the algorithm produces the edge list monotonously moving from beginning to end, it scans through the sorted read request and forwards the still cached values to the target positions using an I/O-efficient priority queue. We show that this approach causes $\mathcal{O}(\text{scan}(m_0) + \text{sort}(m))$ I/Os, where m_0 is the number of edges in the seed graph and m is the number of edges produced.

TFP-BA outperforms a fast implementation of BB-BA by several orders of magnitude once the graph size exceeds the available RAM by only 2%.

MP-BA:
 Section 3.3

- MP-BA is a highly engineered parallel generator and offloads a number of subtasks onto a general purpose graphics processor. The algorithm implements dynamic weighted sampling using a binary tree T partially stored in external memory. Each node u of the generated graph corresponds to a leaf in T labeled with the degree of u ; any inner node stores the total weight of all leaves contained in its left subtree.

In order to select a neighbor, MP-BA first has to sample a leaf according to the current degree distribution and then increment the leaf's weight to account for the newly gained edge. The key insight is that we can do both in a single top-down traversal from the root to the sampled leaf. This allows us to combine the queries for sampling and updating into a single operation and, in turn, to coalesce queries into batches. We show that MP-BA executes $\mathcal{O}(\text{sort}(n_0 + m))$ I/Os, where n_0 is the number of nodes in the seed graph and m is the number of edges produced.

We use two forms of parallelism: firstly, we cut T at a certain depth to process the subtrees rooted there pleasingly parallel. In order to handle the high volume

of requests near T 's root, we additionally design a PRAM algorithm to process multiple requests to the same tree node in parallel.

MP-BA's implementation executes the latter part on a GPU for maximal throughput. In total, we find that MP-BA is 17.6 times faster than BB-BA for instances fitting in main memory and scales well into the EM regime. Compared to a previous solution on a distributed cluster with 48 dual-socket machines, MP-BA yields competitive results and poses a viable alternative using only a single machine.

9.2 Simple Graphs from Prescribed Degree Sequence

In Chapter 4, we present EM-LFR, a complex pipeline consisting of four novel I/O-efficient algorithms to sample large instances from the *LFR* graph model. Our scalable implementation of EM-LFR can even supply benchmark data for *distributed* community detection schemes [170].

In this section, we focus on *LFR*'s most challenging component, namely the problem of uniformly sampling a simple graph from a prescribed degree sequence. It is a common task in network analysis; e.g., to obtain null models (see Section 1.2.7.1), to perturb existing graphs, or as a building block in graph generators.

The generators considered in the following are Markov-Chain-Monte-Carlo (MCMC) approaches. Even if such solutions are frequently used, we are unaware of general and practically applicable bounds on their mixing times. Thus, we do not claim rigorous bounds on the total work to obtain a uniform sample, and rather derive the complexity of each approach as a function of Markov chain steps carried out.


Even the ranking of the perturbation schemes remains a mostly open problem. To the best of our knowledge, there exist only few analytical results pertaining to the relative performance of the MCMC schemes (e.g., [84] comparing *CB*'s mixing time to *ES*). Phrased as asymptotic inclusions with significant gaps, these findings provide little practical guidance. We, therefore, approach the problem empirically (cf. [282]).

1. *EM-HH with EM-ES*: Both algorithms form an I/O-efficient variant of the *Fixed-Degree-Sequence-Model (FDSM)*. It first deterministically generates a biased simple graph matching a prescribed degree sequence. Then it uses the *Edge Switching (ES)* MCMC approach to obtain a uniform sample.

For the first step, we design an I/O-efficient algorithm inspired by a generator due to Havel and Hakimi [173, 165]. The main algorithmic contribution lies in the efficient usage of a compressed representation of the supplied degree sequence. It groups together all nodes with the same degree and allows efficient queries and updates by EM-HH.

In our analysis, we only account for internal I/Os triggered by EM-HH. This is motivated by the fact that our EM-LFR pipeline moves the algorithm's input and output directly between stages without an additional round trip to disk.¹ We show

¹If the output needs to be streamed to disk, the single scan required dominates the total I/O complexity.

comprehensive survey:
 [Section 2.6](#)

EM-HH:
 [Section 4.4](#)

that our algorithm is I/O-optimal and observe further that EM-HH is I/O-free for many practical settings. More formally, we find that with high probability EM-HH does not need any I/O to generate graphs with $\mathcal{O}(M^{2\gamma})$ edges if an ordered degree sequence from a powerlaw distribution with exponent γ is supplied.

EM-ES:
 Section 4.5

The *Edge Switching* MCMC gives the means to perturb the graph: in each step, the algorithm selects two edges uniformly at random and exchanges their endpoints if the resulting graph remains simple (i.e., if the new edges are neither self-loops nor already contained in the graph). Our I/O-efficient variant EM-ES processes $\Theta(m)$ switches in a single batch. In contrast to $\Theta(m)$ unstructured I/Os of the original formulation, EM-ES causes $\mathcal{O}(\text{scan}(m))$ I/Os for a structured scan of the edge list. Within a batch, several types of dependencies between switches can occur and have to be handled. EM-ES achieves this using *Time Forward Processing* and multiple sort- and scan-passes of edges and switches. We show that EM-ES triggers $\mathcal{O}(\text{sort}(n + m))$ I/Os in total to execute $\Theta(m)$ switches.

EM-CM/ES:
 Section 4.6

2. **EM-CM/ES:** The previous combination of EM-HH followed by EM-ES starts with a highly biased simple graph. EM-CM/ES takes another route by starting with a random but non-simple graph and switches edges until we obtain a simple random graph.

To this end, we implement an I/O-efficient generator for the *Configuration Model* and modify EM-ES to accept non-simple inputs without increasing its I/O complexity. The modified algorithm accepts all switches that neither increase the multiplicity of a given edge nor introduce self-loops. Non-simple edges are also switched more frequently than legal edges to accelerate the repair phase.

empirical performance:
 Section 4.10.7

Observe, however, that it does not suffice to rewire non-simple edges using the presented variant of *ES* as it produces a biased sample [18, 24]. Instead, additional *ES* steps are necessary. Even then, our empirical comparison suggests that EM-CM/ES can converge faster to a uniform sample than the previous method.

3. **EM-CB:** *Curveball (CB)* [321] implements an MCMC process similar to *ES* with the following modification: instead of selecting random edges, *CB* selects two random nodes $u \neq v$, and *trades* their neighborhoods as follows. First, *CB* freezes all edges that either connect u and v themselves or link to neighbors which u and v have in common. Then, the remaining neighbors are randomly shuffled while maintaining the degrees of u and v . A single *CB* trade can therefore inflict “more change” to a graph than a single edge switch; depending on the processed graph, a state in *CB*’s Markov chain may have up to $2^{\Theta(n)}$ neighbors while the degrees in *ES*’s chain are bounded by $\mathcal{O}(n^4)$ [81]. Empirical data suggests that fewer trades are necessary to mix a graph (though each trade may require more work).

CB exposes more data locality than *ES* since all information required to carry out a trade is contained in the two neighborhoods. This is in contrast to *ES*, which requires additional unstructured reads to prevent a switch from introducing multi-edges. Note, however, that an undirected edge is classically stored twice – once for

each endpoint. In this scenario, frequent unstructured updates are necessary and negate the previously mentioned locality benefits. Our I/O-efficient EM-CB thus relies on a dynamic data structure and assigns each edge only to the endpoint that is traded next. EM-CB uses *Time Forward Processing* to ensure that the complete neighborhood of a node is available when needed.

EM-CB:

 [Section 5.4.1](#)

EM-CB works in batches. At the beginning of each batch, it samples² the node pairs to be traded within the batch and organizes them in dedicated indices. These auxiliary data structures are used to address the *TFP* messages and to determine which endpoint of an edge will be traded first. We show that EM-CB issues $\mathcal{O}(r[\text{sort}(n) + \text{sort}(m)])$ I/Os to carry out r global trades (see below).

Since the I/O-efficient auxiliary data structures cause overheads, we investigate another trade-off between I/Os and computation. To this end, we propose IM-CB, a variant of EM-CB. It is intended for small and medium-sized graphs where a certain degree of unstructured accesses to main memory is acceptable.

IM-CB:

 [Section 5.4.2](#)

4. **EM-GCB and EM-PGCB:** We generalize *Global Curveball* (*G-CB*) to undirected graphs. An *undirected global trade* is a sequence of $\lfloor n/2 \rfloor$ single trades such that the neighborhood of each node is traded at most once. We show that the process converges to a uniform distribution and give empirical evidence of its superior performance compared to *CB*.

Undirected Global Curveball:

 [Section 5.3.3](#)

G-CB allows us to eliminate the support data structures of EM-CB and IM-CB. Since each node participates once³ in a global trade, we interpret a global trade as a random permutation of nodes where we trade pairwise adjacent nodes. Our I/O-efficient algorithm EM-GCB for undirected *G-CB* maintains this permutation implicitly using a collision-free and invertible hash function.

EM-GCB:

 [Section 5.4.3](#)

EM-PGCB is a parallel extension of EM-GCB. It subdivides each batch into so-called *microchunks* and executes the trades contained in parallel. The size of each microchunk is chosen such that almost all trades contained are independent; a variant of work stealing is used to resolve rare dependencies. We give experimental evidence that, in some cases, EM-PGCB can achieve the same quality as EM-ES nearly one order of magnitude faster.

EM-PGCB:

 [Section 5.4.4](#)

9.3 Geometrically Embedded Random Graphs

Random Hyperbolic Graphs (*RHG*s) are a popular network model which naturally exhibits many features commonly observed in complex networks. *RHG* assigns each node a position on a two-dimensional hyperbolic disk of radius R . These positions are conveniently expressed in polar coordinates meaning that each point is located in terms of its distance r (radius) to the disk's center and an angular coordinate θ .

²The trade sequence may also be provided as input. In contrast to *G-CB*, arbitrary pairs are supported.

³For simplicity, we assume here that n is even; see Section 5.4.3 for the general case.


Threshold RHG

In the so-called *Threshold RHG* [159] variant, we connect all pairs of points whose hyperbolic distance is smaller than R . Denote the points' coordinates with (r_i, θ_i) and (r_j, θ_j) respectively, then their hyperbolic distance $d(p_i, p_j)$ is defined as

$$\cosh(d(p_i, p_j)) = \cosh(r_i) \cosh(r_j) - \sinh(r_i) \sinh(r_j) \cos(\theta_i - \theta_j). \quad (9.1)$$

Thus, the hyperbolic distance is a function of the relative and absolute positions of both points; the closer a point is to the disk's center, the more neighbors it is expected to have. We obtain a powerlaw degree distribution with a controllable exponent by choosing an appropriate radial density function when randomly placing points.

Binomial RHG

local cohesion:
 *Section 1.2.5*

Binomial RHG extends *Threshold RHG* by adding a positive *temperature* parameter T that affects the local cohesion. In the binomial variant, each pair of nodes $p_i \neq p_j$ is independently connected by an edge with probability $p_T(d(p_i, p_j))$ defined as follows:

$$p_T(d) = \left[\exp\left(\frac{d - R}{2T}\right) + 1 \right]^{-1} \quad (9.2)$$

Binomial RHG contains *Threshold RHG* as p_T becomes a step function for $T \rightarrow 0$.

9.3.1 A Fast and Memory-Efficient Streaming Generator for RHG

Most *RHG* generators (including *HYPERGIRGS* in Chapter 8) work in two phases. In a preprocessing step, they first sample all points and store them in some form of geometric data structure. In the main phase, they query the precomputed data to find the point pairs that imply an edge.

In Chapter 6, we demonstrate that in practice these support structures have a large memory footprint that renders them unsuitable for accelerator hardware with a small dedicated memory. We then present *HYPERGEN*, a streaming generator for *Threshold RHGs* which instead samples the points on demand. It executes a sweep-line algorithm and stores the set of nodes that may still find neighbors in its sweep-line state; we refer to them as candidates.

point sampling:

 *Section 6.2*

Roughly speaking, *HYPERGEN* randomly samples points with non-decreasing angular coordinates.⁴ For each new point, the algorithm identifies all sufficiently close candidates and emits edges to them. The generator then marks the point a candidate itself and advances the sweep-line. *HYPERGEN* stops the sweep-line at additional points, e.g., to prune candidates whose distances to the sweep-line are so large that they cannot find neighbors anymore.

To manage the computational cost of maintaining the sweep state, we use several conservative approximations that do not infringe on the generator's faithful reproduction of *RHG*s. They exploit the distribution of points as well as properties of the hyperbolic distance function. The majority of points can be quickly pruned from the algorithm's state. In contrast, the few points that have small radii stay candidates for a significantly

⁴This is an over-simplification to avoid a detailed discussion of the sweep-line's behavior. For technical reasons discussed in Section 6.2, *HYPERGEN* does not consider the coordinate of point u itself, but instead samples the point of smallest angle at distance R from u and then places u accordingly.

longer period of time. To accommodate the different requirements, HYPERGEN partitions the hyperbolic disk into $\Theta(\log n)$ concentric bands. Each band has its own sweep-line and state which remain synchronized with the states of its adjacent bands.

Observe that, due to the angular periodicity of the hyperbolic disk, points sampled late (i.e., with angles near 2π) can be adjacent to points discovered and pruned much earlier. HYPERGEN accounts for this by restarting the sampling process until all candidates of the first phase are processed. It exploits pseudorandomness to obtain consistent point coordinates in both phases.

We show that HYPERGEN has a memory footprint of $\mathcal{O}([n^{1-\alpha}\bar{d}^\alpha + \log n] \log n)$ with high probability. For realistic average degrees $\bar{d} = o(n/\log^{1/\alpha}(n))$ this is a significant asymptotic reduction over classical approaches.

Parallelization is possible by splitting the disk into segments of equal size. Some care has to be taken to manage the dependencies near the segments' borders. HYPERGEN also significantly accelerates the frequent distance computations by preparing auxiliary values per point. This removes all transcendental functions (here \sinh , \cosh , and \cos) from Eq. (9.1). Refined versions of these techniques carry over to Chapters 7 and 8.

Our implementation is designed with SIMD in mind and is explicitly vectorized. It uses SIMD instructions to compute eight hyperbolic distances simultaneously (which is only possible because we first removed the aforementioned transcendental functions).

In an experimental evaluation, we compare HYPERGEN with four state-of-the-art generators. It is consistently the fastest, reaching a speedup of nearly 30 compared to the second fastest competitor. On commodity hardware, HYPERGEN produces 370 million edges per second for graphs with $10^6 \leq m \leq 10^{12}$ while utilizing less than 600 MB of RAM. We demonstrate nearly linear scalability on an Intel Xeon Phi with 60 processors and 240 hardware threads.

HYPERGEN:
 [Section 6.3](#)

parallelism:
 [Section 6.3.2](#)

precomputation:
 [Section 6.4.1](#)

9.3.2 A Communication-Agnostic Generator for RHG

Funke et al. [139] present *KaGen*, a random graph generator suite for distributed computing including RHG, a communication-agnostic generator for *Threshold RHG*. RHG and HYPERGEN were developed independently at roughly the same time, and share ideas to sample specific subsections of the hyperbolic disk using pseudorandomization. While HYPERGEN uses a monotonous sweep-like motion optimized for memory usage, RHG uses less structured queries. They are supported by a fine-grained partitioning of the hyperbolic space which ingeniously allows random access to any cell.

In Chapter 7 (based on the journal version [138] of [139]), we combine both techniques and improve the load balancing to obtain the communication-agnostic sweep-line generator sRHG which consistently outperforms RHG. We demonstrate its scalability to up to 32,768 cores and produce a graph with $n = 2^{39}$ nodes in less than a minute.

sRHG:
 [Section 7.7.3](#)

9.3.3 GIRG-based Generator

In Chapter 8, we engineer a previously known expected linear time sampling algorithm for GIRGs by Bringmann et al. [70] and adapt it to *Binomial RHGs*. We refer to our

comparison RHG & GIRG:

☞ Section 8.2.3

algorithms as GIRGS and HYPERGIRGS, respectively.⁵ To the best of our knowledge, GIRGS is the first practically efficient generator for the GIRG model. Here, we focus on RHGs since the algorithmic treatment of both models is very similar.

tree-nodes

HYPERGIRGS first samples all points and builds a data structure that can be interpreted as a polar quad-tree. While the structure is similar to the previous state-of-the-art generator by v. Looz et al. [223], differences in details result in a polynomial gap in their runtimes. In the following, we refer to nodes of the quad-tree as *tree-nodes* (to distinguish them from the hyperbolic nodes contained).

Bringmann et al. propose the following neighborhood search which we adapt for HYPERGIRGS. For simplicity, we initially restrict ourselves to *Threshold RHGs*. The generator enumerates all pairs of tree-nodes that may contain point pairs sufficiently close to imply an edge. This is done in a pessimistic and oblivious fashion, i.e., without considering the actual points represented by the tree-nodes. HYPERGIRGS then emits edges by testing all point pairs contained in each previously enumerated pair of tree-nodes. To avoid asymptotically significant overheads, the algorithm pairs tree-nodes as high up in the quad-tree as possible without adding unintended distance computations.

The quad-tree needs to support efficient random access to all points contained within any tree-node at any depth. Similarly to [70], HYPERGIRGS achieves this using z-order space-filling curves [268] to map the tree to memory. This choice allows us to efficiently build and query the quad-tree using Morton codes [251].

edge probability $p_T(d)$:

☞ Eq. (9.2)

general sampling
technique:

☞ Section 2.3.3.4 ff.

In case of *Binomial RHGs* with $T > 0$, any node pair has a positive (yet mostly negligible) probability $p_T(d)$ to be connected. HYPERGIRGS therefore has to consider all tree-node pairs — even those with a tiny connection probability. In the latter case, we bound the connection probability from below and use geometric jumps followed by rejection sampling to prune the search space. We also engineer an exact look-up table-based sampling scheme to reduce the evaluation of transcendental functions during the computation of linking probabilities $p_T(d)$.

HYPERGIRGS processes the tree-node pairs pleasingly parallel. As a special feature, our implementation guarantees reproducibility in the sense that two runs with the same set of parameters and seed values output the same set of edges (though not necessarily in the same order). Compared to existing *RHG* generators (most of which only support *Threshold RHG*), our implementation is the fastest sequential *RHG* generator and competitive for shared-memory parallelism.

Our implementation of GIRGS supports geometric spaces in up to five dimensions. The algorithm's overall structure is similar to HYPERGEN, though we need to replace the quad-tree with a high-dimensional variant and adapt details pertaining to the tree-node pairing and distance computation. In Section 8.B.3 (Appendix), we discuss the implications to Morton codes, and study several implementations including a bit-parallel variant using the BMI2 instruction-set extension for x86 processors [184].

⁵Bringmann et al. already discuss the applicability to *RHG* by showing an asymptotic inclusion of *RHG* in *GIRG*. The models are, however, not identical as we empirically demonstrate in Section 8.5.3. Our modifications, leading to HYPERGIRGS, close this gap.

9.4 Future Research Opportunities

Network models and matching generators remain an open field for future research. Here, we provide a non-exhaustive selection of issues related to Chapters 3 to 8 and refer to Section 2.11 for general remarks on future challenges of scalable graph generation.

9.4.1 Preferential Attachment

Most high-performance preferential attachment schemes (including MP-BA) can produce multi-edges. The issue has typically little practical impact since for most sufficiently large seed graphs⁶ only few multi-edges are emitted and can be easily removed in post-processing. Still, the intermediate presence of multi-edges introduces a bias during sampling which justifies further research into faithful preferential attachment for simple graphs with little performance penalties.


Another line of research pertains to alternative query models such as local sampling of huge random objects [153]. Here, the user may repeatedly inquire about specific local properties while the generator needs to supply consistent answers without sampling the whole object. There are already results in the context of BA graphs. Even et al. [126] propose such an “on-the-fly generator” allowing the user to query the neighborhoods of arbitrary nodes. With high probability the generator answers in time $\mathcal{O}(\log^5 n)$ and increases the size of the internal data structure by $\mathcal{O}(\log^2 n)$ bits while consuming $\mathcal{O}(\log^4 n)$ random bits. Further, the communication-agnostic generator by Sanders and Schulz [294] can consistently answer queries to arbitrary positions of the graph’s edge list in constant expected time and $\mathcal{O}(\log n)$ time with high probability. Due to its use of pseudorandomness, the generator consumes no random bits and requires no internal data structure to track already given answers.


It, hence, seems promising to extend advanced query models to additional network models (cf. [45]), and to identify applications and practically relevant query semantics. This also entails the design of matching scalable generators.

9.4.2 Simple Graphs from Prescribed Degree Sequence

Most MCMC schemes to generate uniformly distributed simple graphs with a prescribed degree sequence lack rigorous mixing time bounds that are practically useful. In general, it is even difficult to predict their relative performance. Experimental campaigns can provide practical guidance and expose structural insights that might eventually lead to analytical results. Our empirical findings of Chapters 4 and 5 are only a first step towards this goal. In on-going research, we are investigating more advanced experimental techniques (e.g., inspired by [51]) and consider additional graph classes to obtain broadly reliable experimental results.

⁶The star graph is a worst-case input as every second sample is expected to yield the central hub. We can, however, first process the seed graph with a slower generator to make it less pathological and then hand it over to a faster scalable implementation.

Pref. Attach. sampling without multi-edges:
 [Section 2.4.2.2](#)

communication-agnostic generator for BA:
 [Section 2.4.2.1](#)

Observe that for some graph classes such as regular graphs or degree sequences following a powerlaw distribution with exponent $\gamma > 2.88$, dedicated processes are available and achieve expected linear total work (e.g., [24, 143, 236]). While the underlying algorithms tend to be quite intricate (e.g., more than 20 carefully balanced switching types in [18, 24]), our preliminary results suggest that they can be made practical [18]. Even then, further research is required to render those approaches truly scalable.

9.4.3 Random Hyperbolic Graphs

The generation of *RHG*s has been heavily investigated and (near) optimal sampling algorithms exist for important models of computation. To the best of our knowledge, our generators `HYPERGEN`, `SRHG`, and `HYPERGIRGS` remain the practically fastest solutions available for shared-memory and distributed parallelism. Still, open problems remain.

`HYPERGEN` and `SRHG` only consider *Threshold RHG* and we are not aware of comparable results for strictly positive temperatures. To obtain a memory-efficient generator for *Binomial RHG*, it seems possible to extend `HYPERGEN` using ideas proposed in [336, Sec. 3.7]. This, however, requires either a new point sampling scheme or novel approaches to handle the longevity of requests in the binomial variant. Further, a combination of *RHG* with the cell-skipping of `HYPERGIRGS` is conceivable even if *RHG*'s current partitioning does not support the merging of cells as done by `HYPERGIRGS`.

Another line of inquiry pertains to the parametrization of *RHG*. For *Threshold RHG*, v. Looz et al. describe the state-of-the-art solution to derive the radius R of the hyperbolic disk in order to match a prescribed density and powerlaw exponent [223]. We are not aware of equally reliable means to estimate the parameters of *Binomial RHG*. However, our *GIRGS* generator can rescale the provided node weights (which correspond to radii in the *RHG* model) to match a required density very accurately; similar techniques seem possible for *Binomial RHG*.

Going even further, one might not only estimate the model's global parameters, but actually learn an embedding (i.e., estimate point positions) of a prescribed graph at scale (see e.g., [50, 144]). Many learning methods involve an optimization process which needs to (partially) generate the graph in each step. If the embedding is sufficiently similar to the point distributions implied by *RHG*, the previously discussed sampling techniques may help to accelerate such tasks.

GIRG scaling:
👉 Section 8.B.2

Bibliography

- [1] M. G. Aartsen et al. The IceProd framework: Distributed data processing for the icecube neutrino observatory. *CoRR*, abs/1311.5904, 2013. arXiv:1311.5904.
- [2] M. G. Aartsen et al. The IceProd framework: Distributed data processing for the icecube neutrino observatory. *J. Parallel Distributed Comput.*, 75:198–211, 2015. doi:10.1016/j.jpdc.2014.08.001.
- [3] E. Abbe. Community detection and stochastic block models. *Found. Trends Commun. Inf. Theory*, 14(1-2):1–162, 2018. doi:10.1561/01000000067.
- [4] D. Achlioptas, A. Coja-Oghlan, M. Hahn-Klimroth, J. Lee, N. Müller, M. Penschuck, and G. Zhou. The random 2-SAT partition function. *CoRR*, abs/2002.03690, 2020. Accepted for Random Structures And Algorithms. arXiv:2002.03690.
- [5] Advanced Micro Devices Inc. *AMD Accelerated Parallel Processing — OpenCL Programming Guide*, nov 2013. Revision 2.7.
- [6] P. Afshani, R. Fagerberg, D. Hammer, R. Jacob, I. Kostitsyna, U. Meyer, M. Penschuck, and N. Sitchinava. Fragile complexity of comparison-based algorithms. In M. A. Bender, O. Svensson, and G. Herman, editors, *European Symp. on Algorithms ESA*, volume 144 of *LIPICs*, pages 2:1–2:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.ESA.2019.2.
- [7] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988. doi:10.1145/48529.48535.
- [8] J. H. Ahrens and U. Dieter. Sequential random sampling. *ACM Trans. Math. Softw.*, 11(2):157–169, 1985. doi:10.1145/214392.214402.
- [9] S. G. Aksoy, E. Purvine, E. C. Sanchez, and M. Halappanavar. A generative graph model for electrical infrastructure networks. *J. Complex Networks*, 7(1):128–162, 2019. doi:10.1093/comnet/cny016.
- [10] M. Alam and K. S. Perumalla. GPU-based parallel algorithm for generating massive scale-free networks using the preferential attachment model. In J. Y. Nie, Z. Obradovic, T. Suzumura, R. Ghosh, R. Nambiar, C. Wang, H. Zang, R. Baeza-Yates, X. Hu, J. Kepner, A. Cuzzocrea, J. Tang, and M. Toyoda, editors, *IEEE Int. Conf. on Big Data Big-Data*, pages 3302–3311. Institute of Electrical and Electronics Engineers IEEE, 2017. doi:10.1109/BigData.2017.8258315.
- [11] M. Alam, K. S. Perumalla, and P. Sanders. Novel parallel algorithms for fast multi-GPU-based generation of massive scale-free networks. *Data Science and Engineering*, 4(1):61–75, 2019. doi:10.1007/s41019-019-0088-6.
- [12] M. M. Alam, M. Khan, and M. V. Marathe. Distributed-memory parallel algorithms for generating massive scale-free networks using preferential attachment model. In W. Gropp and S. Matsuoka, editors, *Int. Conf. for High Performance Computing, Networking, Storage and Analysis SC*, pages 91:1–91:12. Assoc. for Computing Machinery ACM, 2013. doi:10.1145/2503210.2503291.
- [13] M. M. Alam, M. Khan, A. Vullikanti, and M. V. Marathe. An efficient and scalable algorithmic method for generating large: scale random graphs. In J. West and C. M. Pancake, editors, *Int. Conf. for High Performance Computing, Networking, Storage and Analysis SC*, pages 372–383. Institute of Electrical and Electronics Engineers IEEE, 2016. doi:10.1109/SC.2016.31.

- [14] R. Albert and A. L. Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1):47–97, Jan 2002. doi:10.1103/revmodphys.74.47.
- [15] R. Albert, B. DasGupta, and N. Mobasher. Topological implications of negative curvature for biological and social networks. *Phys. Rev. E*, 89:032811, mar 2014. URL: <https://link.aps.org/doi/10.1103/PhysRevE.89.032811>, doi:10.1103/PhysRevE.89.032811.
- [16] R. Aldecoa, C. Orsini, and D. V. Krioukov. Hyperbolic graph generator. *Comput. Phys. Commun.*, 196:492–496, 2015. doi:10.1016/j.cpc.2015.05.028.
- [17] A. D. Alexandrov, M. F. Ionescu, K. E. Schauer, and C. J. Scheiman. LogGP: Incorporating long messages into the LogP model - one step closer towards a realistic model for parallel computation. In C. E. Leiserson, editor, *ACM Symp. on Parallel Algorithms and Architectures SPAA*, pages 95–105. ACM, 1995. doi:10.1145/215399.215427.
- [18] D. Allendorf. Implementation and evaluation of a uniform graph sampling algorithm for prescribed power-law degree sequences. Master’s thesis, Goethe University Frankfurt, Germany, 2020.
- [19] J. Alstott, E. Bullmore, and D. Plenz. powerlaw: A python package for analysis of heavy-tailed distributions. *PLOS ONE*, 9(1):1–11, Jan 2014. doi:10.1371/journal.pone.0085777.
- [20] O. Angel, R. van der Hofstad, and C. Holmgren. Limit laws for self-loops and multiple edges in the configuration model, 2017. arXiv:1603.07172.
- [21] M. J. Appel and R. P. Russo. The connectivity of a graph on uniform points on $[0, 1]^d$. *Statistics & Probability Letters*, 60(4):351–357, 2002.
- [22] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms (extended abstract). In S. G. Akl, F. K. H. A. Dehne, J. R. Sack, and N. Santoro, editors, *Int. Workshop on Algorithms and Data Structures WADS*, volume 955 of *LNCS*, pages 334–345. Springer, 1995. doi:10.1007/3-540-60220-8_74.
- [23] L. Arge. The Buffer Tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003. doi:10.1007/s00453-003-1021-x.
- [24] A. Arman, P. Gao, and N. C. Wormald. Fast uniform generation of random graphs with given degree sequences. In D. Zuckerman, editor, *IEEE Symp. on Foundations of Comp. Science FOCS*, pages 1371–1379. Institute of Electrical and Electronics Engineers IEEE, 2019. doi:10.1109/FOCS.2019.00084.
- [25] J. Atwood, B. F. Ribeiro, and D. Towsley. Efficient network generation under general preferential attachment. In C. W. Chung, A. Z. Broder, K. Shim, and T. Suel, editors, *Int. World Wide Web Conf. WWW*, pages 695–700. Assoc. for Computing Machinery ACM, 2014. doi:10.1145/2567948.2579357.
- [26] K. Azadbakht, N. Bezirgiannis, F. S. d. Boer, and S. Aliakbary. A high-level and scalable approach for generating scale-free graphs using active objects. In S. Ossowski, editor, *ACM Symp. on Appl. Computing*, pages 1244–1250. Assoc. for Computing Machinery ACM, 2016. doi:10.1145/2851613.2851722.
- [27] A. Bacher, O. Bodini, A. Hollender, and J. O. Lumbroso. MergeShuffle: a very fast, parallel random permutation algorithm. In L. Ferrari and M. Vamvakari, editors, *GASCom*, volume 2113 of *CEUR Workshop Proceedings*, pages 43–52. CEUR-WS.org, 2018. URL: <http://ceur-ws.org/Vol-2113/paper3.pdf>.

-
- [28] L. Backstrom, P. Boldi, M. Rosa, J. Ugander, and S. Vigna. Four degrees of separation. In N. S. Contractor, B. Uzzi, M. W. Macy, and W. Nejdl, editors, *Web Science WebSci*, pages 33–42. Assoc. for Computing Machinery ACM, 2012. doi:10.1145/2380718.2380723.
- [29] D. A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner. Benchmarking for graph clustering and partitioning. In *Encyclopedia of Social Network Analysis and Mining*, pages 73–82. Springer, 2014. doi:10.1007/978-1-4614-6170-8_23.
- [30] S. H. Bae and B. Howe. GossipMap: a distributed community detection algorithm for billion-edge directed graphs. In J. Kern and J. S. Vetter, editors, *Int. Conf. for High Performance Computing, Networking, Storage and Analysis SC*, pages 27:1–27:12. Assoc. for Computing Machinery ACM, 2015. doi:10.1145/2807591.2807668.
- [31] J. Baert. Libmorton: C++ Morton encoding/decoding library, 2018. URL: <https://github.com/Forceflow/libmorton>.
- [32] A. L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.
- [33] A. L. Barabási, R. Albert, and H. Jeong. Mean-field theory for scale-free random networks. *Physica A: Statistical Mechanics and its Applications*, 272(1-2):173–187, 1999.
- [34] A. L. Barabási et al. *Network science*. Cambridge university press, 2016.
- [35] V. Batagelj and U. Brandes. Efficient generation of large random networks. *Physical Review E*, 71(3), Mar 2005. doi:10.1103/physreve.71.036113.
- [36] N. Baumann and S. Stiller. Network models. In U. Brandes and T. Erlebach, editors, *Network Analysis: Methodological Foundations [Dagstuhlseminar, 13-16 April 2004]*, volume 3418 of *LNCS*, pages 341–372. Springer, 2004. doi:10.1007/978-3-540-31955-9_13.
- [37] A. Beckmann, R. Dementiev, and J. Singler. Building a parallel pipelined external memory algorithm library. In *IEEE Int. Parallel and Distributed Processing Symp. IPDPS*, pages 1–10. Institute of Electrical and Electronics Engineers IEEE, 2009. doi:10.1109/IPDPS.2009.5161001.
- [38] E. A. Bender and E. R. Canfield. The asymptotic number of labeled graphs with given degree sequences. *J. Comb. Theory, Ser. A*, 24(3):296–307, 1978. doi:10.1016/0097-3165(78)90059-6.
- [39] J. L. Bentley and J. B. Saxe. Generating sorted lists of random numbers. *ACM Trans. Math. Softw.*, 6(3):359–364, 1980. doi:10.1145/355900.355907.
- [40] P. Berenbrink, D. Hammer, D. Kaaser, U. Meyer, M. Penschuck, and H. Tran. Simulating population protocols in sub-constant time per interaction. In *European Symp. on Algorithms ESA*, volume 173 of *LIPIcs*, pages 16:1–16:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ESA.2020.16.
- [41] A. Berger and C. J. Carstens. Smaller universes for sampling graphs with fixed degree sequence, 2018. arXiv:1803.02624.
- [42] P. Bhattacharyya and B. K. Chakrabarti. The mean distance to the n th neighbour in a uniform distribution of random points. *European J. of Physics*, 29(3):639, 2008.
- [43] M. H. Bhuiyan, J. Chen, M. Khan, and M. V. Marathe. Fast parallel algorithms for Edge-Switching to achieve a target visit rate in heterogeneous graphs. In *Int. Conf. on Parallel Processing ICPP*, pages 60–69. Institute of Electrical and Electronics Engineers IEEE, 2014. doi:10.1109/ICPP.2014.15.

- [44] T. Bingmann, M. Axtmann, E. Jöbstl, S. Lamm, H. C. Nguyen, A. Noe, S. Schlag, M. Stumpp, T. Sturm, and P. Sanders. Thrill: High-performance algorithmic distributed batch data processing with C++. In J. Joshi, G. Karypis, L. Liu, X. Hu, R. Ak, Y. Xia, W. Xu, A. H. Sato, S. Rachuri, L. H. Ungar, P. S. Yu, R. Govindaraju, and T. Suzumura, editors, *IEEE Int. Conf. on Big Data BigData*, pages 172–183. Institute of Electrical and Electronics Engineers IEEE, 2016. doi:10.1109/BigData.2016.7840603.
- [45] A. S. Biswas, R. Rubinfeld, and A. Yodpinyanee. Local access to huge random objects through partial sampling. In T. Vidick, editor, *Innovations in Theoretical Comp. Science Conf. ITCS*, volume 151 of *LIPICs*, pages 27:1–27:65. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.ITCS.2020.27.
- [46] A. Blanca and M. Mihail. Efficient generation ε -close to $G(n, p)$ and generalizations. *CoRR*, abs/1204.5834, 2012. URL: <http://arxiv.org/abs/1204.5834>, arXiv:1204.5834.
- [47] T. Bläsius, C. Freiberger, T. Friedrich, M. Katzmann, F. Montenegro-Retana, and M. Thi-effry. Efficient shortest paths in scale-free networks with underlying hyperbolic geometry. In I. Chatzigiannakis, C. Kaklamanis, D. Marx, and D. Sannella, editors, *Int. Colloquium on Automata, Languages, and Programming ICALP*, volume 107 of *LIPICs*, pages 20:1–20:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.ICALP.2018.20.
- [48] T. Bläsius, T. Friedrich, M. Katzmann, U. Meyer, M. Penschuck, and C. Weyand. Efficiently generating geometric inhomogeneous and hyperbolic random graphs. In M. A. Bender, O. Svensson, and G. Herman, editors, *European Symp. on Algorithms ESA*, volume 144 of *LIPICs*, pages 21:1–21:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.ESA.2019.21.
- [49] T. Bläsius, T. Friedrich, A. Krohmer, and S. Laue. Efficient embedding of scale-free graphs in the hyperbolic plane. In P. Sankowski and C. D. Zaroliagis, editors, *European Symp. on Algorithms ESA*, volume 57 of *LIPICs*, pages 16:1–16:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICs.ESA.2016.16.
- [50] T. Bläsius, T. Friedrich, A. Krohmer, and S. Laue. Efficient embedding of scale-free graphs in the hyperbolic plane. *IEEE/ACM Trans. Netw.*, 26(2):920–933, 2018. URL: <http://doi.ieeeComp.org/10.1109/TNET.2018.2810186>, doi:10.1109/TNET.2018.2810186.
- [51] T. Bläsius, T. Friedrich, M. Katzmann, A. Krohmer, and J. Striebel. Towards a systematic evaluation of generative network models. In A. Bonato, P. Pralat, and A. M. Raigorodskii, editors, *Algorithms and Models for the Web Graph*, volume 10836 of *LNCS*, pages 99–114. Springer, 2018. doi:10.1007/978-3-319-92871-5_8.
- [52] G. E. Blelloch. Prefix sums and their applications. Technical report, Citeseer, 1990.
- [53] V. Blondel, J. L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *J. of Statistical Mechanics: Theory and Experiment*, 2008(10), 2008. URL: <http://dx.doi.org/10.1088/1742-5468/2008/10/P10008>.
- [54] M. Bode, N. Fountoulakis, and T. Müller. On the largest component of a hyperbolic model of complex networks. *Electr. J. Comb.*, 22(3):P3.24, 2015. URL: <http://www.combinatorics.org/ojs/index.php/eljc/article/view/v22i3p24>.
- [55] M. Boguñá, M. Kitsak, and D. Krioukov. Cosmological networks. *New J. of Physics*, 16(9):093031, Sep 2014. doi:10.1088/1367-2630/16/9/093031.

-
- [56] M. Boguñá, F. Papadopoulos, and D. Krioukov. Sustaining the internet with hyperbolic mapping. *Nature Communications*, 1(1), Sep 2010. doi:10.1038/ncomms1063.
- [57] M. Boguná, R. Pastor-Satorras, A. Díaz-Guilera, and A. Arenas. Models of social networks based on social distance attachment. *Physical review E*, 70(5):056122, 2004.
- [58] M. Bohr. A 30 year retrospective on Dennard’s MOSFET scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.
- [59] B. Bollobás. *Random Graphs, Second Edition*, volume 73 of *Cambridge Studies in Advanced Math*. Cambridge University Press, 2011. doi:10.1017/CBO9780511814068.
- [60] B. Bollobás, C. Borgs, J. T. Chayes, and O. Riordan. Directed scale-free graphs. In *ACM-SIAM Symp. on Discrete Algorithms SODA*, pages 132–139. ACM-SIAM, 2003. URL: <http://dl.acm.org/citation.cfm?id=644108.644133>.
- [61] B. Bollobás, O. Riordan, J. Spencer, and G. E. Tusnády. The degree sequence of a scale-free random graph process. *Random Struct. Algorithms*, 18(3):279–290, 2001. doi:10.1002/rsa.1009.
- [62] A. Bonato. A survey of models of the web graph. In A. López-Ortiz and A. M. Hamel, editors, *Workshop on Combinatorial and Algorithmic Aspects of Networking CAAN*, volume 3405 of *LNCS*, pages 159–172. Springer, 2004. doi:10.1007/11527954_16.
- [63] S. P. Borgatti, K. M. Carley, and D. Krackhardt. On the robustness of centrality measures under conditions of imperfect data. *Soc. Networks*, 28(2):124–136, 2006. doi:10.1016/j.socnet.2005.05.001.
- [64] S. P. Borgatti and M. G. Everett. Models of core/periphery structures. *Soc. Networks*, 21(4):375–395, 2000. doi:10.1016/S0378-8733(99)00019-2.
- [65] U. Brandes and M. Mader. A quantitative comparison of stress-minimization approaches for offline dynamic graph drawing. In M. J. v. Kreveld and B. Speckmann, editors, *Int. Symp. on Graph Drawing GD*, volume 7034 of *LNCS*, pages 99–110. Springer, 2011. doi:10.1007/978-3-642-25878-7_11.
- [66] R. A. Bridges, J. P. Collins, E. M. Ferragut, J. A. Laska, and B. D. Sullivan. A multi-level anomaly detection algorithm for time-varying graph data with interactive visualization. *Social Netw. Analys. Mining*, 6(1):99:1–99:14, 2016. doi:10.1007/s13278-016-0409-y.
- [67] K. Bringmann and T. Friedrich. Exact and efficient generation of geometric random variates and random graphs. In F. V. Fomin, R. Freivalds, M. Z. Kwiatkowska, and D. Peleg, editors, *Automata, Languages, and Programming – Int. Colloquium ICALP, Proceedings, Part I*, volume 7965 of *LNCS*, pages 267–278. Springer, 2013. doi:10.1007/978-3-642-39206-1_23.
- [68] K. Bringmann, R. Keusch, and J. Lengler. Geometric inhomogeneous random graphs. *CoRR*, abs/1511.00576, 2015. URL: <http://arxiv.org/abs/1511.00576>, arXiv:1511.00576.
- [69] K. Bringmann, R. Keusch, and J. Lengler. Sampling geometric inhomogeneous random graphs in linear time. In K. Pruhs and C. Sohler, editors, *European Symp. on Algorithms ESA*, volume 87 of *LIPICs*, pages 20:1–20:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.ESA.2017.20.
- [70] K. Bringmann, R. Keusch, and J. Lengler. Geometric inhomogeneous random graphs. *Theor. Comput. Sci.*, 760:35–54, 2019. doi:10.1016/j.tcs.2018.08.014.
- [71] G. Brinkmann. Program Fullgen—a program for generating nonisomorphic fullerenes. see <http://cs.anu.edu.au/bdm/plantri>, 2011.

- [72] G. Brinkmann and B. D. McKay. Fast generation of planar graphs. *MATCH Commun. Math. Comput. Chem*, 58(2), 2007.
- [73] T. Britton, M. Deijfen, and F. Liljeros. A weighted configuration model and inhomogeneous epidemics. *J. of Statistical Physics*, 145(5):1368–1384, Sep 2011. doi:10.1007/s10955-011-0343-3.
- [74] A. D. Broido and A. Clauset. Scale-free networks are rare. *Nature Communications*, 10(1), Mar 2019. doi:10.1038/s41467-019-08746-5.
- [75] V. Buchhold, P. Sanders, and D. Wagner. Efficient calculation of microscopic travel demand data with low calibration effort. In F. B. Kashani, G. Trajcevski, R. H. Güting, L. Kulik, and S. D. Newsam, editors, *ACM SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems SIGSPATIAL*, pages 379–388. Assoc. for Computing Machinery ACM, 2019. doi:10.1145/3347146.3359361.
- [76] F. Buckley and F. Harary. *Distance in graphs*. Addison Wesley, 1990.
- [77] N. Buzun, A. Korshunov, V. Avanesov, I. Filonenko, I. Kozlov, D. Turdakov, and H. Kim. EgoLP: Fast and distributed community detection in billion-node social networks. In Z. H. Zhou, W. Wang, R. Kumar, H. Toivonen, J. Pei, J. Z. Huang, and X. Wu, editors, *IEEE Int. Conf. on Data Mining Workshops ICDM*, pages 533–540. Institute of Electrical and Electronics Engineers IEEE, 2014. doi:10.1109/ICDMW.2014.158.
- [78] T. T. Cao, A. Nanjappa, M. Gao, and T. S. Tan. A GPU accelerated algorithm for 3d delaunay triangulation. In J. Keyser and P. V. Sander, editors, *Symp. on Interactive 3D Graphics and Games I3D*, pages 47–54. Assoc. for Computing Machinery ACM, 2014. doi:10.1145/2556700.2556710.
- [79] C. J. Carstens. Proof of uniform sampling of binary matrices with fixed row sums and column sums for the fast Curveball algorithm. *Physical Review E*, 91:042812, 2015.
- [80] C. J. Carstens. *Topology of Complex Networks: Models and Analysis*. PhD thesis, RMIT University, 2016.
- [81] C. J. Carstens, A. Berger, and G. Strona. Curveball: a new generation of sampling algorithms for graphs with fixed degree sequence. *CoRR*, abs/1609.05137, 2016. URL: <http://arxiv.org/abs/1609.05137>, arXiv:1609.05137.
- [82] C. J. Carstens, M. Hamann, U. Meyer, M. Penschuck, H. Tran, and D. Wagner. Parallel and I/O-efficient randomisation of massive networks using Global Curveball trades. In Y. Azar, H. Bast, and G. Herman, editors, *European Symp. on Algorithms ESA*, volume 112 of *LIPICs*, pages 11:1–11:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.ESA.2018.11.
- [83] C. J. Carstens and K. J. Horadam. Switching edges to randomize networks: what goes wrong and how to fix it. *J. Complex Networks*, 5(3):337–351, 2017. doi:10.1093/comnet/cnw027.
- [84] C. J. Carstens and P. Kleer. Comparing the Switch and Curveball Markov Chains for sampling binary matrices with fixed marginals. *CoRR*, abs/1709.07290, 2017. URL: <http://arxiv.org/abs/1709.07290>, arXiv:1709.07290.
- [85] L. Carter and M. N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979. doi:10.1016/0022-0000(79)90044-8.

-
- [86] J. M. Cebrian, L. Natvig, and M. Jahre. Scalability analysis of AVX-512 extensions. *J. Supercomput.*, 76(3):2082–2097, 2020. doi:10.1007/s11227-019-02840-7.
- [87] D. Chakrabarti and C. Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Comput. Surv.*, 38(1):2, 2006. doi:10.1145/1132952.1132954.
- [88] S. Chatterjee and P. Diaconis. Estimating and understanding exponential random graph models. *The Annals of Statistics*, 41(5):2428–2461, Oct 2013. doi:10.1214/13-aos1155.
- [89] V. Chauhan, A. Gutfraind, and I. Safro. Multiscale planar graph generation. *Appl. Network Science*, 4(1):46:1–46:28, 2019. doi:10.1007/s41109-019-0142-3.
- [90] J. Chen and I. Safro. Algebraic distance on graphs. *SIAM J. Scientific Computing*, 33(6):3468–3490, 2011. doi:10.1137/090775087.
- [91] Y. J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In K. L. Clarkson, editor, *ACM-SIAM Symp. on Discrete Algorithms SODA*, pages 139–149. ACM-SIAM, 1995. URL: <http://dl.acm.org/citation.cfm?id=313651.313681>.
- [92] F. Chung and L. Lu. The average distances in random graphs with given expected degrees. *Proceedings of the National Academy of Sciences*, 99(25):15879–15882, Dec 2002. doi:10.1073/pnas.252631999.
- [93] F. Chung and L. Lu. Connected components in random graphs with given expected degree sequences. *Annals of Combinatorics*, 6(2):125–145, Nov 2002. doi:10.1007/pl00012580.
- [94] V. Chvátal and P. L. Hammer. Aggregation of inequalities in integer programming. In *Studies in Integer Programming*, pages 145–162. Elsevier, 1977. doi:10.1016/s0167-5060(08)70731-3.
- [95] K. Chykhradze, A. Korshunov, N. Buzun, R. Pastukhov, N. N. Kuzyurin, D. Turdakov, and H. Kim. Distributed generation of billion-node social graphs with overlapping community structure. In P. Contucci, R. Menezes, A. Omicini, and J. Poncela-Casasnovas, editors, *Workshop on Complex Networks CompleNet 2014*, volume 549 of *Studies in Computational Intelligence*, pages 199–208. Springer, 2014. doi:10.1007/978-3-319-05401-8_19.
- [96] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009. doi:10.1137/070710111.
- [97] G. W. Cobb and Y. P. Chen. An application of Markov Chain Monte Carlo to community ecology. *The American Mathematical Monthly*, 110(4):265–288, 2003. URL: <http://www.jstor.org/stable/3647877>.
- [98] T. F. Consortium, R. PMII, R. C. (DGT), et al. A promoter-level mammalian expression atlas. *Nat.*, 507(7493):462–470, 2014. doi:10.1038/nature13182.
- [99] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, E. E. Santos, K. E. Schauer, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Commun. ACM*, 39(11):78–85, 1996. doi:10.1145/240455.240477.
- [100] H. A. David and H. N. Nagaraja. Order statistics. *Encyclopedia of Statistical Sciences*, 2004.
- [101] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008. URL: <http://doi.acm.org/10.1145/1327452.1327492>, doi:10.1145/1327452.1327492.

- [102] R. Dementiev, L. Kettner, and P. Sanders. STXXL: standard template library for XXL data sets. *Softw. Pract. Exp.*, 38(6):589–637, 2008. doi:10.1002/spe.844.
- [103] A. Denise, M. Vasconcellos, and D. J. Welsh. The random planar graph. In *Congressus numerantium*, 1996.
- [104] R. H. Dennard, F. H. Gaensslen, H. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE J. of Solid-State Circuits*, 9(5):256–268, 1974.
- [105] L. Devroye. *Non-Uniform Random Variate Generation*. Springer, 1986. doi:10.1007/978-1-4613-8643-8.
- [106] L. Devroye and P. Kruszewski. The botanical beauty of random binary trees. In F. J. Brandenburg, editor, *Int. Symp. on Graph Drawing GD*, volume 1027 of *LNCS*, pages 166–177. Springer, 1995. doi:10.1007/BFb0021801.
- [107] P. Diaconis, S. P. Holmes, and S. Janson. Threshold graph limits and random threshold graphs. *Internet Math.*, 5(3):267–320, 2008. doi:10.1080/15427951.2008.10129166.
- [108] P. Dinklage, J. Ellert, J. Fischer, D. Köppl, and M. Penschuck. Bidirectional text compression in external memory. In M. A. Bender, O. Svensson, and G. Herman, editors, *European Symp. on Algorithms ESA, LIPIcs*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ESA.2019.41.
- [109] S. N. Dorogovtsev and J. F. Mendes. *Evolution of networks: From biological nets to the Internet and WWW*. OUP Oxford, 2013.
- [110] S. N. Dorogovtsev and J. F. F. Mendes. Evolution of networks. *Advances in Physics*, 51(4):1079–1187, Jun 2002. doi:10.1080/00018730110112519.
- [111] S. N. Dorogovtsev, J. F. F. Mendes, and A. N. Samukhin. Anomalous percolation properties of growing networks. *Phys. Rev. E*, 64:066110, Nov 2001.
- [112] M. Drobyshvskiy and D. Turdakov. Random graph modeling: A survey of the concepts. *ACM Comput. Surv.*, 52(6):131:1–131:36, 2020. doi:10.1145/3369782.
- [113] N. Durak, T. G. Kolda, A. Pinar, and C. Seshadhri. A scalable null model for directed graphs matching all degree distributions: In, out, and reciprocal. In *IEEE Network Science Workshop NSW*, pages 23–30. Institute of Electrical and Electronics Engineers IEEE, 2013. doi:10.1109/NSW.2013.6609190.
- [114] R. A. Dwyer. Higher-dimensional Voronoi diagrams in linear expected time. *Discret. Comput. Geom.*, 6:343–367, 1991. doi:10.1007/BF02574694.
- [115] H. Edelsbrunner. *Voronoi Diagrams*, pages 293–333. Springer, 1987.
- [116] S. Edunov, D. Logothetis, C. Wang, A. Ching, and M. Kabiljo. Generating synthetic social graphs with Darwini. In *IEEE Int. Conf. on Distributed Computing Systems ICDCS*, pages 567–577. Institute of Electrical and Electronics Engineers IEEE, 2018. doi:10.1109/ICDCS.2018.00062.
- [117] F. Eggenberger and G. Pólya. Über die Statistik verketteter Vorgänge. *ZAMM-J. of Appl. Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik und Mechanik*, 3(4):279–289, 1923.
- [118] R. B. Eggleton and D. A. Holton. Simple and multigraphic realizations of degree sequences. In *ACCM'80, Lecture Notes in Math.*, pages 155–172. Springer, 1980. URL: <http://dx.doi.org/10.1007/BFb0091817>.

-
- [119] O. El-Daghar, E. Lundberg, and R. A. Bridges. EGBTER: capturing degree distribution, clustering coefficients, and community structure in a single random graph model. In U. Brandes, C. Reddy, and A. Tagarelli, editors, *Int. Conf. on Advances in Social Networks Analysis and Mining ASONAM*, pages 282–289. Institute of Electrical and Electronics Engineers IEEE, 2018. doi:10.1109/ASONAM.2018.8508598.
- [120] S. Emmons, S. G. Kobourov, M. Gallant, and K. Börner. Analysis of network clustering algorithms and cluster quality metrics at scale. *PLoS ONE*, 11(7):1–18, Jul 2016. URL: <http://dx.doi.org/10.1371/journal.pone.0159161>.
- [121] P. Erdős and A. Rényi. On random graphs I. *Publicationes Mathematicae Debrecen*, 1959.
- [122] P. Erdős and A. Rényi. On the evolution of random graphs. *Mathematical Institute of the Hungarian Academy of Sciences*, 5(1), 1960.
- [123] P. L. Erdős, C. S. Greenhill, T. R. Mezei, I. Miklós, D. Soltész, and L. Soukup. The mixing time of the swap (switch) markov chains: a unified approach. *CoRR*, abs/1903.06600, 2019. URL: <http://arxiv.org/abs/1903.06600>, arXiv:1903.06600.
- [124] A. V. Esquivel and M. Rosvall. Comparing network covers using mutual information, 2012. arXiv:1202.0425.
- [125] L. Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii academiae scientiarum Petropolitanae*, pages 128–140, 1741.
- [126] G. Even, R. Levi, M. Medina, and A. Rosén. Sublinear random access generators for preferential attachment graphs. In I. Chatzigiannakis, P. Indyk, F. Kuhn, and A. Muscholl, editors, *Automata, Languages, and Programming – Int. Colloquium ICALP*, volume 80 of *LIPICs*, pages 6:1–6:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.ICALP.2017.6.
- [127] C. T. Fan, M. E. Muller, and I. Rezucha. Development of sampling plans by using sequential (item by item) selection techniques and digital comp. *J. of the American Statistical Assoc.*, 57(298), 1962.
- [128] R. A. S. Fisher and F. Yates. *Statistical tables for biological, agricultural, and medical research*. Oliver and Boyd, 1963.
- [129] M. J. Flynn. Some comp. organizations and their effectiveness. *IEEE Transactions on Comp.*, C-21(9):948–960, 1972.
- [130] A. Fog. Instruction tables. version 2020-10-11. URL: https://www.agner.org/optimize/instruction_tables.pdf.
- [131] S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75–174, Feb 2010. doi:10.1016/j.physrep.2009.11.002.
- [132] S. Fortunato and M. Barthélemy. Resolution limit in community detection. *PNAS*, 104(1):36–41, 2007. URL: <https://doi.org/10.1073/pnas.0605965104>.
- [133] S. Fortunato and D. Hric. Community detection in networks: A user guide. *Physics Reports*, 659:1–44, Nov 2016. doi:10.1016/j.physrep.2016.09.002.
- [134] T. Friedrich and A. Krohmer. On the diameter of hyperbolic random graphs. In M. M. Halldórsson, K. Iwama, N. Kobayashi, and B. Speckmann, editors, *Automata, Languages, and Programming – Int. Colloquium ICALP, Proceedings, Part II*, volume 9135 of *LNCS*, pages 614–625. Springer, 2015. doi:10.1007/978-3-662-47666-6_49.

- [135] T. Friedrich and A. Krophmer. On the diameter of hyperbolic random graphs. *SIAM J. Discret. Math.*, 32(2):1314–1334, 2018. doi:10.1137/17M1123961.
- [136] T. Friedrich. From graph theory to network science: The natural emergence of hyperbolicity (tutorial). In *Int. Symp. on Theoretical Aspects of Comp. Science STACS*, pages 5:1–5:9, 2019. doi:10.4230/LIPIcs.STACS.2019.5.
- [137] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *IEEE Symp. on Foundations of Comp. Science FOCS*, pages 285–298. IEEE Comp. Society, 1999. doi:10.1109/SFFCS.1999.814600.
- [138] D. Funke, S. Lamm, U. Meyer, M. Penschuck, P. Sanders, C. Schulz, D. Strash, and M. v. Looz. Communication-free massively distributed graph generation. *J. Parallel Distributed Comput.*, 131:200–217, 2019. doi:10.1016/j.jpdc.2019.03.011.
- [139] D. Funke, S. Lamm, P. Sanders, C. Schulz, D. Strash, and M. v. Looz. Communication-free massively distributed graph generation. In *IEEE Int. Parallel and Distributed Processing Symp. IPDPS*, pages 336–347. Institute of Electrical and Electronics Engineers IEEE, 2018. doi:10.1109/IPDPS.2018.00043.
- [140] D. Funke and P. Sanders. Parallel d -d Delaunay triangulations in shared and distributed memory. In S. P. Fekete and V. Ramachandran, editors, *Workshop on Algorithm Engineering and Experiments ALENEX*, pages 207–217. Society for Industrial and App. Math. SIAM, 2017. doi:10.1137/1.9781611974768.17.
- [141] É. Fusy. Uniform random sampling of planar graphs in linear time. *Random Struct. Algorithms*, 35(4):464–522, 2009. doi:10.1002/rsa.20275.
- [142] E. Galin, A. Peytavie, E. Guérin, and B. Benes. Authoring hierarchical road networks. *Comput. Graph. Forum*, 30(7):2021–2030, 2011. doi:10.1111/j.1467-8659.2011.02055.x.
- [143] P. Gao and N. C. Wormald. Uniform generation of random regular graphs. *SIAM J. Comput.*, 46(4):1395–1427, 2017. doi:10.1137/15M1052779.
- [144] G. García-Pérez, A. Allard, M. Á. Serrano, and M. Boguñá. Mercator: uncovering faithful hyperbolic embeddings of complex networks. *New J. of Physics*, 21(12):123033, dec 2019. doi:10.1088/1367-2630/ab57d2.
- [145] D. Garlaschelli. The weighted random graph model. *New J. of Physics*, 11(7):073005, Jul 2009. doi:10.1088/1367-2630/11/7/073005.
- [146] M. N. Garofalakis, J. Gehrke, and R. Rastogi, editors. *Data Stream Management - Processing High-Speed Data Streams*. Data-Centric Systems and Applications. Springer, 2016. doi:10.1007/978-3-540-28608-0.
- [147] O. Gebhard, M. Hahn-Klimroth, O. Parczyk, M. Penschuck, M. Rolvien, J. Scarlett, and N. Tan. Near optimal sparsity-constrained group testing: improved bounds and algorithms. *CoRR*, abs/2004.11860, 2020. URL: <https://arxiv.org/abs/2004.11860>, arXiv:2004.11860.
- [148] E. N. Gilbert. Random graphs. *The Annals of Mathematical Statistics*, 30(4):1141–1144, Dec 1959. doi:10.1214/aoms/1177706098.
- [149] E. N. Gilbert. Random plane networks. *J. of the Society for Industrial and App. Math.*, 9(4):533–543, Dec 1961. doi:10.1137/0109045.
- [150] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, Jun 2002. doi:10.1073/pnas.122653799.

-
- [151] C. Gkantsidis, M. Mihail, and E. W. Zegura. The Markov Chain simulation method for generating connected power law random graphs. In R. E. Ladner, editor, *Workshop on Algorithm Engineering and Experiments*, pages 16–25. Society for Industrial and App. Math. SIAM, 2003.
- [152] A. Goldenberg, A. X. Zheng, S. E. Fienberg, and E. M. Airoldi. A survey of statistical network models. *Foundations and Trends in Machine Learning*, 2(2):129–233, 2009. doi:10.1561/2200000005.
- [153] O. Goldreich, S. Goldwasser, and A. Nussboim. On the implementation of huge random objects. *SIAM J. Comput.*, 39(7):2761–2822, 2010. doi:10.1137/080722771.
- [154] N. J. Gotelli and G. R. Graves. *Null models in ecology*. Smithsonian Institution, 1996.
- [155] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. of App. Math.*, 17(2):416–429, 1969.
- [156] C. S. Greenhill. A polynomial bound on the mixing time of a markov chain for sampling regular directed graphs. *Electr. J. Comb.*, 18(1), 2011. URL: http://www.combinatorics.org/Volume_18/Abstracts/v18i1p234.html.
- [157] C. S. Greenhill. The switch markov chain for sampling irregular graphs (extended abstract). In P. Indyk, editor, *ACM-SIAM Symp. on Discrete Algorithms SODA*, pages 1564–1572. Society for Industrial and App. Math. SIAM, 2015. doi:10.1137/1.9781611973730.103.
- [158] C. S. Greenhill and M. Sfragara. The Switch Markov Chain for sampling irregular graphs and digraphs. *Theor. Comput. Sci.*, 719:1–20, 2018. doi:10.1016/j.tcs.2017.11.010.
- [159] L. Gugelmann, K. Panagiotou, and U. Peter. Random hyperbolic graphs: Degree sequence and clustering - (extended abstract). In A. Czumaj, K. Mehlhorn, A. M. Pitts, and R. Wattenhofer, editors, *Int. Colloquium on Automata, Languages, and Programming ICALP*, volume 7392 of *LNCS*, pages 573–585. Springer, 2012. doi:10.1007/978-3-642-31585-5_51.
- [160] P. Gupta and P. R. Kumar. The capacity of wireless networks. *IEEE Trans. Inf. Theory*, 46(2):388–404, 2000. doi:10.1109/18.825799.
- [161] A. Gutfraind, I. Safro, and L. A. Meyers. Multiscale network generation. In *Int. Conf. on Information Fusion FUSION*, pages 158–165. Institute of Electrical and Electronics Engineers IEEE, 2015. URL: <http://ieeexplore.ieee.org/document/7266557/>.
- [162] J. Hackl and B. T. Adey. Generation of spatially embedded random networks to model complex transportation networks. In *Int. Probabilistic Workshop*, pages 217–230. Springer, Nov 2016. doi:10.1007/978-3-319-47886-9_15.
- [163] A. Hagberg, D. Schult, and P. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Python in Science Conf. SciPy*, pages 11–15, Pasadena, CA, 2008.
- [164] W. W. Hager, J. T. Hungerford, and I. Safro. A multilevel bilinear programming algorithm for the vertex separator problem. *Comp. Opt. and Appl.*, 69(1):189–223, 2018. doi:10.1007/s10589-017-9945-2.
- [165] S. L. Hakimi. On realizability of a set of integers as degrees of the vertices of a linear graph. i. *J. of the Society for Industrial and App. Math.*, 10(3):496–506, Sep 1962. doi:10.1137/0110037.
- [166] M. Hamann, U. Meyer, M. Penschuck, H. Tran, and D. Wagner. I/O-efficient generation of massive graphs following the LFR benchmark. *CoRR*, abs/1604.08738, 2017.

- [167] M. Hamann, U. Meyer, M. Penschuck, H. Tran, and D. Wagner. I/O-efficient generation of massive graphs following the LFR benchmark. *ACM J. of Experimental Algorithmics*, 23, 2018. doi:10.1145/3230743.
- [168] M. Hamann, U. Meyer, M. Penschuck, and D. Wagner. I/O-efficient generation of massive graphs following the LFR benchmark. In S. P. Fekete and V. Ramachandran, editors, *Workshop on Algorithm Engineering and Experiments ALENEX*, pages 58–72. Society for Industrial and App. Math. SIAM, 2017. doi:10.1137/1.9781611974768.5.
- [169] M. Hamann, B. Strasser, D. Wagner, and T. Zeitz. Simple distributed graph clustering using modularity and Map Equation. *CoRR*, abs/1710.09605, 2017. URL: <http://arxiv.org/abs/1710.09605>, arXiv:1710.09605.
- [170] M. Hamann, B. Strasser, D. Wagner, and T. Zeitz. Distributed graph clustering using modularity and Map Equation. In M. Aldinucci, L. Padovani, and M. Torquati, editors, *Euro-Par 2018: Int. Conf. on Parallel and Distributed Computing*, volume 11014 of *LNCS*, pages 688–702. Springer, 2018. doi:10.1007/978-3-319-96983-1_49.
- [171] P. L. Hammer and B. Simeone. The splittance of a graph. *Combinatorica*, 1(3):275–284, 1981. doi:10.1007/BF02579333.
- [172] S. Harenberg, G. Bello, L. Gjeltema, S. Ranshous, J. Harlalka, R. Seay, K. Padmanabhan, and N. Samatova. Community detection in large-scale networks: a survey and empirical evaluation. *Wiley Interdisciplinary Reviews: Computational Statistics*, 6(6):426–439, Jul 2014. doi:10.1002/wics.1319.
- [173] V. Havel. Poznámka o existenci konečných grafů. *Časopis pro pěstování matematiky*, 080(4), 1955.
- [174] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Int. Conf. on Distributed Computing Systems ICDCS*, pages 522–529. IEEE Comp. Society, 2003. doi:10.1109/ICDCS.2003.1203503.
- [175] S. Hert and M. Seel. *dd* convex hulls and Delaunay triangulations. In *CGAL 4.7 User and Reference Manual*. CGAL, 2015.
- [176] P. W. Holland, K. B. Laskey, and S. Leinhardt. Stochastic blockmodels: First steps. *Social Networks*, 5(2):109–137, Jun 1983. doi:10.1016/0378-8733(83)90021-7.
- [177] P. Holme and B. J. Kim. Growing scale-free networks with tunable clustering. *Physical Review E*, 65(2), Jan 2002. doi:10.1103/physreve.65.026107.
- [178] M. Holtgrewe. A scalable coarsening phase for a multi-level graph partitioning algorithm. Master’s thesis, University of Karlsruhe, 2009.
- [179] M. Holtgrewe, P. Sanders, and C. Schulz. Engineering a scalable high quality graph partitioner. In *IEEE Int. Parallel and Distributed Processing Symp. IPDPS*, pages 1–12. Institute of Electrical and Electronics Engineers IEEE, 2010. doi:10.1109/IPDPS.2010.5470485.
- [180] L. Hubert and P. Arabie. Comparing partitions. *J. of Classification*, 2(1):193–218, 1985. URL: <http://dx.doi.org/10.1007/BF01908075>.
- [181] L. Hübschle-Schneider and P. Sanders. Linear work generation of R-MAT graphs. *CoRR*, abs/1905.03525, 2019. arXiv:1905.03525.

-
- [182] L. Hübschle-Schneider and P. Sanders. Parallel weighted random sampling. In M. A. Bender, O. Svensson, and G. Herman, editors, *European Symp. on Algorithms ESA*, volume 144 of *LIPICs*, pages 59:1–59:24. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.ESA.2019.59.
- [183] M. L. Huson and A. Sen. Broadcast scheduling algorithms for radio networks. In *Proceedings of MILCOM '95*, volume 2, pages 647–651 vol.2, nov 1995. doi:10.1109/MILCOM.1995.483546.
- [184] Intel Corporation. *Intel®64 and IA-32 Architectures – Software Developer’s Manual – Volume 2*, 2019.
- [185] F. Iorio, M. Bernardo-Faura, A. Gobbi, T. Cokelaer, G. Jurman, and J. Saez-Rodriguez. Efficient randomization of biological networks while preserving functional characterization of individual nodes. *BMC Bioinform.*, 17:542:1–542:14, 2016. doi:10.1186/s12859-016-1402-1.
- [186] S. Itzkovitz, R. Milo, N. Kashtan, G. Ziv, and U. Alon. Subgraphs in random networks. *Physical Review E*, 68(2), Aug 2003. doi:10.1103/physreve.68.026127.
- [187] J. JáJá. *An Introduction to Parallel Algorithms*. Addison Wesley, 1992.
- [188] X. Jia. Wireless networks and random geometric graphs. In *Int. Symp. on Parallel Architectures, Algorithms, and Networks I-SPAN*, pages 575–580. Institute of Electrical and Electronics Engineers IEEE, 2004. doi:10.1109/ISPAN.2004.1300540.
- [189] M. Kaiser. Mean clustering coefficients: the role of isolated nodes and leafs on clustering measures for small-world networks. *New J. of Physics*, 10(8), 2008. URL: <http://dx.doi.org/10.1088/1367-2630/10/8/083042>.
- [190] T. Kawamoto and M. Rosvall. Estimating the resolution limit of the map equation in community detection. *Physical Review E*, 91:012809, 2015. URL: <https://dx.doi.org/10.1103/PhysRevE.91.012809>.
- [191] P. Kennedy. Amazon EC2 cloud compute instances benchmarked w/ rackspace, 2013. (accessed on 10 October 2019).
- [192] H. J. Kim, Y. Lee, B. Kahng, and I. m. Kim. Weighted scale-free network in financial correlations. *J. of the Physical Society of Japan*, 71(9), 2002.
- [193] J. M. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. The web as a graph: Measurements, models, and methods. In T. Asano, H. Imai, D. T. Lee, S. I. Nakano, and T. Tokuyama, editors, *Computing and Combinatorics COCOON*, volume 1627 of *LNCS*, pages 1–17. Springer, 1999. doi:10.1007/3-540-48686-0_1.
- [194] R. Kleinberg. Geographic routing using hyperbolic space. In *IEEE INFOCOM 2007 - 26th IEEE Int. Conf. on Comp. Communications*, pages 1902–1909, 2007.
- [195] D. E. Knuth. *The Art of Comp. Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison Wesley, 1981.
- [196] T. G. Kolda, A. Pinar, T. D. Plantenga, and C. Seshadhri. A scalable generative graph model with community structure. *SIAM J. Scientific Computing*, 36(5), 2014. doi:10.1137/130914218.
- [197] P. Krapivsky, G. Rodgers, and S. Redner. Degree distributions of growing networks. *Physical Review Letters*, 86(23):5401, 2001.

- [198] M. Kretz. *Extending C++ for explicit data-parallel programming via SIMD vector types*. PhD thesis, Goethe University Frankfurt am Main, 2015. URL: <http://publikationen.uni-frankfurt.de/frontdoor/index/index/docId/38415>.
- [199] M. Kretz and V. Lindenstruth. Vc: A C++ library for explicit vectorization. *Softw. Pract. Exp.*, 42(11):1409–1430, 2012. doi:10.1002/spe.1149.
- [200] D. Krioukov, F. Papadopoulos, M. Kitsak, A. Vahdat, and M. Boguñá. Hyperbolic geometry of complex networks. *Physical Review E*, 82(3), Sep 2010. doi:10.1103/physreve.82.036106.
- [201] P. R. Kumar, M. J. Wainwright, and R. Zecchina. *Mathematical Foundations of Complex Networked Information Systems: Politecnico Di Torino, Verrès, Italy 2009*, volume 2141. Springer, 2015.
- [202] R. Kumar, J. Novak, and A. Tomkins. Structure and evolution of online social networks. In T. Eliassi-Rad, L. H. Ungar, M. Craven, and D. Gunopulos, editors, *ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 611–617. Assoc. for Computing Machinery ACM, 2006. doi:10.1145/1150402.1150476.
- [203] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal. Random graph models for the web graph. In *IEEE Symp. on Foundations of Comp. Science FOCS*, pages 57–65. Institute of Electrical and Electronics Engineers IEEE, 2000. doi:10.1109/SFCS.2000.892065.
- [204] S. Kumar. Co-authorship networks: a review of the literature. *Aslib J. Inf. Manag.*, 67(1):55–73, 2015. doi:10.1108/AJIM-09-2014-0116.
- [205] K. Kurihara, Y. Kameya, and T. Sato. A frequency-based stochastic blockmodel. *Workshop on Information-Based Induction Sciences*, 1(1):N2, 2006.
- [206] S. Lamm. Communication efficient algorithms for generating massive networks. Master’s thesis, Karlsruhe Institute of Technology, 2017. doi:10.5445/IR/1000068617.
- [207] L. Lamport and N. A. Lynch. Distributed computing: Models and methods. In J. van Leeuwen, editor, *Handbook of Theoretical Comp. Science, Volume B: Formal Models and Semantics*, pages 1157–1199. Elsevier and MIT Press, 1990. doi:10.1016/b978-0-444-88074-1.50023-8.
- [208] A. Lancichinetti and S. Fortunato. Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities. *Physical Review E*, 80(1), Jul 2009. doi:10.1103/physreve.80.016118.
- [209] A. Lancichinetti and S. Fortunato. Community detection algorithms: A comparative analysis. *Physical Review E*, 80(5), Nov 2009. URL: <http://link.aps.org/doi/10.1103/PhysRevE.80.056117>.
- [210] A. Lancichinetti, S. Fortunato, and F. Radicchi. Benchmark graphs for testing community detection algorithms. *Physical Review E*, 78(4), Oct 2008. doi:10.1103/physreve.78.046110.
- [211] A. Lancichinetti, F. Radicchi, J. J. Ramasco, and S. Fortunato. Finding statistically significant communities in networks. *PLoS ONE*, 6(4):1–18, Apr 2011. URL: <http://dx.doi.org/10.1371/journal.pone.0018961>.
- [212] C. Lee and D. J. Wilkinson. A review of stochastic block models and extensions for graph clustering. *Appl. Network Science*, 4(1):122, 2019. doi:10.1007/s41109-019-0232-2.

-
- [213] J. Leskovec, D. Chakrabarti, J. M. Kleinberg, and C. Faloutsos. Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication. In A. Jorge, L. Torgo, P. Brazdil, R. Camacho, and J. Gama, editors, *European Conf. on Principles and Practice of Knowledge Discovery in Databases PKDD*, volume 3721 of *LNCS*, pages 133–145. Springer, 2005. doi:10.1007/11564126_17.
- [214] J. Leskovec, D. Chakrabarti, J. M. Kleinberg, C. Faloutsos, and Z. Ghahramani. Kronecker graphs: An approach to modeling networks. *J. Mach. Learn. Res.*, 11:985–1042, 2010. URL: <https://dl.acm.org/citation.cfm?id=1756039>.
- [215] J. Leskovec, J. M. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In R. Grossman, R. J. Bayardo, and K. P. Bennett, editors, *ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 177–187. Assoc. for Computing Machinery ACM, 2005. doi:10.1145/1081870.1081893.
- [216] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. <http://snap.stanford.edu/data>, Jun 2014.
- [217] D. A. Levin, Y. Peres, and E. L. Wilmer. *Markov chains and mixing times*. American Mathematical Society AMS, Providence, Rhode Island, 2009.
- [218] L. Li, D. Alderson, J. C. Doyle, and W. Willinger. Towards a theory of scale-free graphs: Definition, properties, and implications. *Internet Math.*, 2(4):431–523, 2005. URL: <https://projecteuclid.org/443/euclid.im/1150477667>.
- [219] E. Lindholm, J. Nickolls, S. F. Oberman, and J. Montrym. NVIDIA tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008. doi:10.1109/MM.2008.31.
- [220] S. Lo. Parallel Delaunay triangulation in three dimensions. *Comput.Methods in Appl.Mech.Eng.*, 237-240:88–106, 2012.
- [221] Y. C. Lo, C. T. Li, and S. D. Lin. Parallelizing preferential attachment models for generating large-scale social networks that cannot fit into memory. In *Int. Conf. on Privacy, Security, Risk and Trust PASSAT, and Int. Confernece on Social Computing SocialCom*, pages 229–238. Institute of Electrical and Electronics Engineers IEEE, 2012. doi:10.1109/SocialCom-PASSAT.2012.28.
- [222] M. v. Looz and H. Meyerhenke. Querying probabilistic neighborhoods in spatial data sets efficiently. In V. Mäkinen, S. J. Puglisi, and L. Salmela, editors, *Int. Workshop on Combinatorial Algorithms IWOCA*, volume 9843 of *LNCS*, pages 449–460. Springer, 2016. doi:10.1007/978-3-319-44543-4_35.
- [223] M. v. Looz, H. Meyerhenke, and R. Prutkin. Generating random hyperbolic graphs in subquadratic time. In K. M. Elbassioni and K. Makino, editors, *Algorithms and Computation - Int. Symp. ISAAC*, volume 9472 of *LNCS*, pages 467–478. Springer, 2015. doi:10.1007/978-3-662-48971-0_40.
- [224] M. v. Looz, M. S. Özdayi, S. Laue, and H. Meyerhenke. Generating massive complex networks with hyperbolic geometry faster in practice. In *IEEE High Performance Extreme Computing Conf. HPEC*, pages 1–6. Institute of Electrical and Electronics Engineers IEEE, 2016. doi:10.1109/HPEC.2016.7761644.
- [225] A. Lumsdaine, D. P. Gregor, B. Hendrickson, and J. W. Berry. Challenges in parallel graph processing. *Parallel Process. Lett.*, 17(1):5–20, 2007. doi:10.1142/S0129626407002843.
- [226] D. Lusher, J. Koskinen, and G. Robins. *Exponential random graph models for social networks*. Cambridge University Press, 2013.

- [227] N. V. R. Mahadev and U. N. Peled. *Threshold Graphs and Related Topics*. Elsevier, 1995.
- [228] P. Mahadevan, D. V. Krioukov, K. R. Fall, and A. Vahdat. Systematic topology analysis and generation using degree correlations. In L. Rizzo, T. E. Anderson, and N. McKeown, editors, *ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Comp. Communications*, pages 135–146. Assoc. for Computing Machinery ACM, 2006. doi:10.1145/1159913.1159930.
- [229] M. Mahdian and Y. Xu. Stochastic Kronecker graphs. In A. Bonato and F. R. K. Chung, editors, *Int. Workshop on Algorithms and Models for the Web Graph WAW*, volume 4863 of *LNCS*, pages 179–186. Springer, 2007. doi:10.1007/978-3-540-77004-6_14.
- [230] A. Maheshwari and N. Zeh. A survey of techniques for designing I/O-efficient algorithms. In U. Meyer, P. Sanders, and J. F. Sibeyn, editors, *Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar, March 10-14, 2002]*, volume 2625 of *LNCS*, pages 36–61. Springer, 2002. doi:10.1007/3-540-36574-5_3.
- [231] P. Massart. *Concentration inequalities and model selection*, volume 6. Springer, 2007.
- [232] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.
- [233] C. J. H. McDiarmid. On the method of bounded differences. *Surveys in Combinatorics*, pages 148–188, 1989.
- [234] C. C. McGeoch. *A Guide to Experimental Algorithmics*. Cambridge University Press, 2012. URL: <http://www.cambridge.org/us/academic/subjects/Comp.-science/algorithmics-complexity-Comp.-algebra-and-computational-g/guide-experimental-algorithmics>.
- [235] A. McGregor. Graph stream algorithms: a survey. *SIGMOD Rec.*, 43(1):9–20, 2014. doi:10.1145/2627692.2627694.
- [236] B. D. McKay and N. C. Wormald. Uniform generation of random regular graphs of moderate degree. *J. Algorithms*, 11(1):52–67, 1990. doi:10.1016/0196-6774(90)90029-E.
- [237] S. Meinert and D. Wagner. An experimental study on generating planar graphs. In M. J. Atallah, X. Y. Li, and B. Zhu, editors, *Joint Int. Conf. on Frontiers in Algorithmics and Algorithmic Aspects in Information and Management FAW-AAIM*, volume 6681 of *LNCS*, pages 375–387. Springer, 2011. doi:10.1007/978-3-642-21204-8_39.
- [238] R. Meusel, S. Vigna, O. Lehmborg, and C. Bizer. Graph structure in the web - revisited: a trick of the heavy tail. In C. W. Chung, A. Z. Broder, K. Shim, and T. Suel, editors, *Int. World Wide Web Conf. WWW*, pages 427–432. Assoc. for Computing Machinery ACM, 2014. doi:10.1145/2567948.2576928.
- [239] U. Meyer and M. Penschuck. Generating massive scale-free networks under resource constraints. In M. T. Goodrich and M. Mitzenmacher, editors, *Workshop on Algorithm Engineering and Experiments ALENEX*, pages 39–52. Society for Industrial and App. Math. SIAM, 2016. doi:10.1137/1.9781611974317.4.
- [240] U. Meyer and M. Penschuck. Large-scale graph generation and big data: An overview on recent results. *Bulletin of the EATCS*, 122, 2017. URL: <http://eatcs.org/beatcs/index.php/beatcs/article/view/494>.

-
- [241] U. Meyer and M. Penschuck. Large-scale graph generation: Recent results of the SPP 1736 – Part II. *it - Information Technology*, 2020.
- [242] U. Meyer, P. Sanders, and J. F. Sibeyn, editors. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*. Springer, 2003. doi:10.1007/3-540-36574-5.
- [243] S. Milgram. The small world problem. *Psychology today*, 2(1):60–67, 1967.
- [244] J. C. Miller and A. A. Hagberg. Efficient generation of networks with given expected degrees. In A. M. Frieze, P. Horn, and P. Pralat, editors, *Algorithms and Models for the Web Graph – Int. Workshop WAW*, volume 6732 of *LNCS*, pages 115–126. Springer, 2011. doi:10.1007/978-3-642-21286-4_10.
- [245] R. Milo. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594):824–827, Oct 2002. doi:10.1126/science.298.5594.824.
- [246] R. Milo, N. Kashtan, S. Itzkovitz, M. Newman, and U. Alon. On the uniform generation of random graphs with prescribed degree sequences, 2003. arXiv:cond-mat/0312028.
- [247] M. Molloy and B. A. Reed. A critical point for random graphs with a given degree sequence. *Random Struct. Algorithms*, 6(2/3):161–180, 1995. doi:10.1002/rsa.3240060204.
- [248] G. E. Moore et al. Cramming more components onto integrated circuits, 1965.
- [249] S. Moreno, J. J. P. III, and J. Neville. Scalable and exact sampling method for probabilistic generative graph models. *Data Min. Knowl. Discov.*, 32(6):1561–1596, 2018. doi:10.1007/s10618-018-0566-x.
- [250] S. Moreno, J. J. P. III, J. Neville, and S. Kirshner. A scalable method for exact sampling from Kronecker family models. In R. Kumar, H. Toivonen, J. Pei, J. Z. Huang, and X. Wu, editors, *IEEE Int. Conf. on Data Mining ICDM*, pages 440–449. Institute of Electrical and Electronics Engineers IEEE, 2014. doi:10.1109/ICDM.2014.148.
- [251] G. M. Morton. A comp. oriented geodetic data base and a new technique in file sequencing. Technical report, Int. Business Machines Company New York, 1966. URL: <https://domino.research.ibm.com/library/cyberdig.nsf/0/0dabf9473b9c86d48525779800566a39?OpenDocument>.
- [252] R. Motwani and P. Raghavan. *Randomized algorithms*. Chapman & Hall/CRC, 2010.
- [253] T. Munzner. Exploring large graphs in 3D hyperbolic space. *IEEE Comp. Graphics and Applications*, 18(4):18–23, 1998. doi:10.1109/38.689657.
- [254] S. Muthukrishnan and G. Pandurangan. Thresholding random geometric graph properties motivated by ad hoc sensor networks. *J. Comput. Syst. Sci.*, 76(7):686–696, 2010. doi:10.1016/j.jcss.2010.01.002.
- [255] S. A. Myers, A. Sharma, P. Gupta, and J. J. Lin. Information network or social network?: the structure of the twitter follow graph. In C. W. Chung, A. Z. Broder, K. Shim, and T. Suel, editors, *Int. World Wide Web Conf. WWW*, pages 493–498. Assoc. for Computing Machinery ACM, 2014. doi:10.1145/2567948.2576939.
- [256] T. Müller and M. Steps. The diameter of KPKVB random graphs. *CoRR*, abs/1707.09555, 2017. URL: <http://arxiv.org/abs/1707.09555>, arXiv:1707.09555.
- [257] F. Nake. Das doppelte Bild. In *Digitale Form*, volume 3,2 of *Bildwelten des Wissens: Kunsthistorisches Jahrbuch für Bildkritik*. De Gruyter, 2006.

- [258] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003. doi:10.1137/S003614450342480.
- [259] M. E. J. Newman. Analysis of weighted networks. *Physical Review E*, 70(5), Nov 2004. doi:10.1103/physreve.70.056131.
- [260] M. E. J. Newman. *Networks: An Introduction*. Oxford University Press, 2010. doi:10.1093/ACPROF:OSO/9780199206650.001.0001.
- [261] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(026113):1–16, 2004. URL: <http://link.aps.org/abstract/PRE/v69/e026113>.
- [262] M. E. J. Newman, S. H. Strogatz, and D. J. Watts. Random graphs with arbitrary degree distributions and their applications. *Physical Review E*, 64(2), Jul 2001. doi:10.1103/physreve.64.026118.
- [263] M. Newman. Power laws, pareto distributions and Zipf’s law. *Contemporary Physics*, 46(5):323–351, 2005. doi:10.1080/00107510500052444.
- [264] S. Nobari, X. Lu, P. Karras, and S. Bressan. Fast random graph generation. In A. Ailamaki, S. Amer-Yahia, J. M. Patel, T. Risch, P. Senellart, and J. Stoyanovich, editors, *Int. Conf. on Extending Database Technology EDBT*, pages 331–342. Assoc. for Computing Machinery ACM, 2011. doi:10.1145/1951365.1951406.
- [265] NVIDIA Corp. CUDA C Programming Guide, 2015. URL: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.
- [266] NVIDIA Corp. CURAND LIBRARY, Mar 2015. URL: http://docs.nvidia.com/cuda/pdf/CURAND_Library.pdf.
- [267] OpenMP Architecture Review Board. OpenMP application program interface version 5.0, 2018. URL: <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf>.
- [268] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In D. J. Rosenkrantz and R. Fagin, editors, *ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 181–190. Assoc. for Computing Machinery ACM, 1984. doi:10.1145/588011.588037.
- [269] E. Parsonage and M. Roughan. Fast generation of spatially embedded random networks. *IEEE Trans. Network Science and Engineering*, 4(2):112–119, 2017. doi:10.1109/TNSE.2017.2681700.
- [270] M. D. Penrose. *Random Geometric Graphs*. Oxford University Press, 2003. URL: <http://www.maths.bath.ac.uk/~masmdp/rgg.html>, doi:10.1093/acprof:oso/9780198506263.001.0001.
- [271] M. Penschuck. Generating practical random hyperbolic graphs in near-linear time and with sub-linear memory. In C. S. Iliopoulos, S. P. Pissis, S. J. Puglisi, and R. Raman, editors, *Int. Symp. on Experimental Algorithms SEA*, volume 75 of *LIPICs*, pages 26:1–26:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.SEA.2017.26.
- [272] M. Penschuck, U. Brandes, M. Hamann, S. Lamm, U. Meyer, I. Safro, P. Sanders, and C. Schulz. Recent advances in scalable network generation. *CoRR*, abs/2003.00736, 2020. arXiv:2003.00736.
- [273] K. Popper. *The Logic of Scientific Discovery*. Hutchinson, 1959.

-
- [274] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C, 2nd Edition*. Cambridge University Press, 1992. URL: <http://www.nr.com/>.
- [275] D. d. S. Price. A general theory of bibliometric and other cumulative advantage processes. *JASIS*, 27(5):292–306, 1976. doi:10.1002/asi.4630270505.
- [276] D. J. D. S. Price. Networks of scientific papers. *Science*, 149(3683):510–515, 1965. URL: <http://www.jstor.org/stable/1716232>.
- [277] N. Przulj, D. G. Corneil, and I. Jurisica. Modeling interactome: scale-free or geometric? *Bioinform.*, 20(18):3508–3515, 2004. doi:10.1093/bioinformatics/bth436.
- [278] X. Que, F. Checconi, F. Petrini, T. Wang, and W. Yu. Lightning-fast community detection in social media: A scalable implementation of the Louvain algorithm. Technical report, Auburn University, 2013. Tech. Rep. AU-CSSE-PASL/13-TR01.
- [279] A. R. Rao, R. Jana, and S. Bandyopadhyay. A Markov Chain Monte Carlo method for generating random $(0, 1)$ -matrices with given marginals. *Sankhyā: The Indian J. Statistics, Series A*, 1996.
- [280] J. Ray, A. Pinar, and C. Seshadhri. Are we there yet? when to stop a markov chain while generating random graphs. In A. Bonato and J. C. M. Janssen, editors, *Int. Workshop on Algorithms and Models for the Web Graph WAW*, volume 7323 of LNCS, pages 153–164. Springer, 2012. doi:10.1007/978-3-642-30541-2_12.
- [281] J. Ray, A. Pinar, and C. Seshadhri. A stopping criterion for markov chains when generating independent random graphs. *J. Complex Networks*, 3(2):204–220, 2015. doi:10.1093/comnet/cnu041.
- [282] S. Rechner. *Markov Chain Monte Carlo algorithms for the uniform sampling of combinatorial objects*. PhD thesis, Martin-Luther-Universität Halle-Wittenberg, 2018. URL: <http://dx.doi.org/10.25673/2241>.
- [283] C. Robert and G. Casella. *Monte Carlo statistical methods*. Springer, 2013.
- [284] R. W. Robinson and N. C. Wormald. Almost all regular graphs are hamiltonian. *Random Struct. Algorithms*, 5(2):363–374, 1994. doi:10.1002/rsa.3240050209.
- [285] A. S. Rodionov and H. Choo. On generating random network structures: Trees. In P. M. A. Sloot, D. Abramson, A. V. Bogdanov, J. J. Dongarra, A. Y. Zomaya, and Y. E. Gorbachev, editors, *Computational Science - ICCS 2003, Int. Conf.*, volume 2658 of LNCS, pages 879–887. Springer, 2003. doi:10.1007/3-540-44862-4_95.
- [286] D. Ron, I. Safro, and A. Brandt. Relaxation-based coarsening and multiscale graph organization. *Multiscale Model. Simul.*, 9(1):407–423, 2011. doi:10.1137/100791142.
- [287] M. Rosvall, D. Axelsson, and C. T. Bergstrom. The map equation. *The European Physical J. Special Topics*, 178(1):13–23, 2009. URL: <http://dx.doi.org/10.1140/epjst/e2010-01179-1>.
- [288] I. Safro, P. Sanders, and C. Schulz. Advanced coarsening schemes for graph partitioning. *ACM J. of Experimental Algorithmics*, 19(1), 2014. doi:10.1145/2670338.
- [289] R. Salfi. A long-period random number generator with application to permutations. In *COMPSTAT*, 1974.
- [290] P. Sanders. Random permutations on distributed, external and hierarchical memory. *Inf. Process. Lett.*, 67(6):305–309, 1998. doi:10.1016/S0020-0190(98)00127-6.

- [291] P. Sanders. Fast priority queues for cached memory. *ACM J. of Experimental Algorithmics*, 5:7, 2000. doi:10.1145/351827.384249.
- [292] P. Sanders, S. Lamm, L. Hübschle-Schneider, E. Schrade, and C. Dachsbacher. Efficient parallel random sampling - vectorized, cache-efficient, and online. *ACM Trans. Math. Softw.*, 44(3):29:1–29:14, 2018. doi:10.1145/3157734.
- [293] P. Sanders, S. Schlag, and I. Müller. Communication efficient algorithms for fundamental big data problems. In X. Hu, T. Y. Lin, V. V. Raghavan, B. W. Wah, R. Baeza-Yates, G. C. Fox, C. Shahabi, M. Smith, Q. Yang, R. Ghani, W. Fan, R. Lempel, and R. Nambiar, editors, *IEEE Int. Conf. on Big Data*, pages 15–23. Institute of Electrical and Electronics Engineers IEEE, 2013. doi:10.1109/BigData.2013.6691549.
- [294] P. Sanders and C. Schulz. Scalable generation of scale-free graphs. *Inf. Process. Lett.*, 116(7):489–491, 2016. doi:10.1016/j.ipl.2016.02.004.
- [295] P. Sanders, K. Mehlhorn, M. Dietzfelbinger, and R. Dementiev. *Sequential and Parallel Algorithms and Data Structures - The Basic Toolbox*. Springer, 2019. doi:10.1007/978-3-030-25209-0.
- [296] W. E. Schlauch, E. A. Horvát, and K. A. Zweig. Different flavors of randomness: comparing random graph models with fixed degree sequences. *Social Netw. Analys. Mining*, 5(1):36:1–36:14, 2015. doi:10.1007/s13278-015-0267-z.
- [297] W. E. Schlauch and K. A. Zweig. Influence of the null-model on motif detection. In J. Pei, F. Silvestri, and J. Tang, editors, *IEEE/ACM Int. Conf. on Advances in Social Networks Analysis and Mining ASONAM*, pages 514–519. Assoc. for Computing Machinery ACM, 2015. doi:10.1145/2808797.2809400.
- [298] Schloss Dagstuhl - Leibniz Center for Informatics. DBLP: Compute science bibliography. August 2020 Snapshot. URL: <https://dblp.uni-trier.de/>.
- [299] C. Schulz, A. Nocaj, M. El-Assady, S. Frey, M. Hlawatsch, M. Hund, G. K. Karch, R. Netzel, C. Schätzle, M. Butt, D. A. Keim, T. Ertl, U. Brandes, and D. Weiskopf. Generative data models for validation and evaluation of visualization techniques. In M. Sedlmair, P. Isenberg, T. Isenberg, N. Mahyar, and H. Lam, editors, *Workshop on Beyond Time and Errors on Novel Evaluation Methods for Visualization BELIV*, pages 112–124. Assoc. for Computing Machinery ACM, 2016. doi:10.1145/2993901.2993907.
- [300] M. . Serrano and M. Boguñá. Weighted configuration model. In *AIP Conf. Proceedings*, volume 776. AIP, 2005.
- [301] C. Seshadhri, T. G. Kolda, and A. Pinar. Community structure and scale-free collections of erdős-rényi graphs. *Physical Review E*, 85(5), May 2012. doi:10.1103/physreve.85.056109.
- [302] C. Seshadhri, A. Pinar, and T. G. Kolda. A hitchhiker’s guide to choosing parameters of stochastic Kronecker graphs. *CoRR*, abs/1102.5046, 2011. URL: <http://arxiv.org/abs/1102.5046>, arXiv:1102.5046.
- [303] Y. Shavitt and T. Tankel. Hyperbolic embedding of internet graph for distance estimation and overlay construction. *IEEE/ACM Trans. Netw.*, 16(1):25–36, 2008. URL: <http://doi.acm.org/10.1145/1373452.1373455>, doi:10.1145/1373452.1373455.
- [304] S. S. Shen-Orr, R. Milo, S. Mangan, and U. Alon. Network motifs in the transcriptional regulation network of *Escherichia coli*. *Nature Genetics*, 31(1):64–68, Apr 2002. doi:10.1038/ng881.

-
- [305] J. R. Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Comput. Geom.*, 22(1-3):21–74, 2002. doi:10.1016/S0925-7721(01)00047-5.
- [306] A. Shine and D. Kempe. Generative graph models based on Laplacian spectra? In L. Liu, R. W. White, A. Mantrach, F. Silvestri, J. J. McAuley, R. Baeza-Yates, and L. Zia, editors, *The World Wide Web Conf. WWW*, pages 1691–1701. Assoc. for Computing Machinery ACM, 2019. doi:10.1145/3308558.3313631.
- [307] J. Shun, Y. Gu, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Sequential random permutation, list contraction and tree contraction are highly parallel. In P. Indyk, editor, *ACM-SIAM Symp. on Discrete Algorithms SODA*, pages 431–448. Society for Industrial and App. Math. SIAM, 2015. doi:10.1137/1.9781611973730.30.
- [308] F. Simini, M. C. González, A. Maritan, and A. L. Barabási. A universal model for mobility and migration patterns. *Nature*, 484(7392), 2012.
- [309] G. Simmel. *Soziologie: Untersuchungen über die Formen der Vergesellschaftung*. Duncker & Humblot, 1908.
- [310] R. Sitzenfrei, M. Möderl, and W. Rauch. Automatic generation of water distribution systems based on GIS data. *Environ. Model. Softw.*, 47:138–147, 2013. doi:10.1016/j.envsoft.2013.05.006.
- [311] G. M. Slota, J. W. Berry, S. D. Hammond, S. L. Olivier, C. A. Phillips, and S. Rajamanickam. Scalable generation of graphs for benchmarking HPC community-detection algorithms. In M. Taufer, P. Balaji, and A. J. Peña, editors, *Int. Conf. for High Performance Computing, Networking, Storage and Analysis SC*, pages 73:1–73:14. Assoc. for Computing Machinery ACM, 2019. doi:10.1145/3295500.3356206.
- [312] T. A. B. Snijders. Statistical models for social networks. *Annual Review of Sociology*, 37, 2011.
- [313] E. Stadlober. Ratio of uniforms as a convenient method for sampling from classical discrete distributions. In E. A. MacNair, K. J. Musselman, and P. Heidelberger, editors, *Winter Simulation Conf.*, pages 484–489. Assoc. for Computing Machinery ACM, 1989. doi:10.1145/76738.76801.
- [314] E. Stadlober. The ratio of uniforms approach for generating discrete random variates. *J. Computational and App. Math.*, 31(1), 1990.
- [315] E. Stadlober and H. Zechner. The patchwork rejection technique for sampling from unimodal distributions. *ACM Trans. Model. Comput. Simul.*, 9(1):59–80, 1999. doi:10.1145/301677.301685.
- [316] C. L. Staudt, A. Sazonovs, and H. Meyerhenke. NetworKit: A tool suite for large-scale complex network analysis. *Network Science*, 4(4):508–530, 2016. doi:10.1017/nws.2016.20.
- [317] M. Stephan and J. Docter. Jülich Supercomputing Centre. JUQUEEN: IBM Blue Gene/Q Supercomp. System at the Jülich Supercomputing Centre. *J. of large-scale research facilities*, 2015, 1, A1. <http://dx.doi.org/10.17815/jlsrf-1-18>.
- [318] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Prémillieu, A. Reid, A. Rico, and P. Walker. The ARM scalable vector extension. *IEEE Micro*, 37(2):26–39, 2017. doi:10.1109/MM.2017.35.

- [319] A. Stivala, G. Robins, and A. Lomi. Exponential random graph model parameter estimation for very large directed networks. *PLOS ONE*, 15(1):e0227804, Jan 2020. doi:10.1371/journal.pone.0227804.
- [320] S. H. Strogatz. Exploring complex networks. *Nature*, 410(6825):268, 2001.
- [321] G. Strona, D. Nappo, F. Boccacci, S. Fattorini, and J. San-Miguel-Ayanz. A fast and unbiased procedure to randomize ecological binary matrices with fixed row and column totals. *Nature Communications*, 5(1), Jun 2014. doi:10.1038/ncomms5114.
- [322] A. Stukowski. Structure identification methods for atomistic simulations of crystalline materials. *Modelling and Simulation in Materials Science and Engineering*, 20(4):045021, 2012.
- [323] L. Tabourier, C. Roth, and J. Cointet. Generating constrained random graphs using multiple edge switches. *ACM J. Exp. Algorithmics*, 16, 2011. doi:10.1145/1963190.2063515.
- [324] The Lemur Project. ClueWeb12 Web Graph, Nov 2013. <http://www.lemurproject.org/clueweb12/webgraph.php>.
- [325] C. Thomassen. Kuratowski’s theorem. *J. of Graph Theory*, 5(3):225–241, 1981. doi:10.1002/jgt.3190050304.
- [326] J. Travers and S. Milgram. An experimental study of the small world problem. In S. Leinhardt, editor, *Social Networks*, pages 179–197. Academic Press, 1977. URL: <http://www.sciencedirect.com/science/article/pii/B9780124424500500183>, doi:https://doi.org/10.1016/B978-0-12-442450-0.50018-3.
- [327] E. Turro et al. Whole-genome sequencing of patients with rare diseases in a national health system. *Nat.*, 583(7814):96–102, 2020. doi:10.1038/s41586-020-2434-2.
- [328] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the Facebook social graph. *CoRR*, abs/1111.4503, 2011. URL: <http://arxiv.org/abs/1111.4503>, arXiv:1111.4503.
- [329] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990. doi:10.1145/79173.79181.
- [330] D. E. Vengroff and J. S. Vitter. Supporting i/o-efficient scientific computation in TPIE. In *IEEE SPDP*, pages 74–77. IEEE, 1995. doi:10.1109/SPDP.1995.530667.
- [331] N. D. Verhelst. An efficient MCMC algorithm to sample binary matrices with fixed marginals. *Psychometrika*, 73(4):705–728, 2008.
- [332] F. Viger and M. Latapy. Efficient and simple generation of random simple connected graphs with prescribed degree sequence. *J. Complex Networks*, 4(1):15–37, 2016. doi:10.1093/comnet/cnv013.
- [333] J. S. Vitter. Faster methods for random sampling. *Commun. ACM*, 27(7):703–718, 1984. doi:10.1145/358105.893.
- [334] J. S. Vitter. An efficient algorithm for sequential random sampling. *ACM Trans. Math. Softw.*, 13(1):58–67, 1987. doi:10.1145/23002.23003.
- [335] J. S. Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Comp. Science*, 2(4):305–474, 2006. doi:10.1561/04000000014.
- [336] M. von Looz. *High-Performance Graph Algorithms*. PhD thesis, KIT – Karlsruhe Institute of Technology, 2018.

-
- [337] J. Von Neumann. Various techniques used in connection with random digits. In *Monte Carlo Method*, volume 12, pages 36–38. National Bureau of Standards, 1951.
- [338] M. D. Vose. A linear algorithm for generating random numbers with a given distribution. *IEEE Trans. Software Eng.*, 17(9):972–975, 1991. doi:10.1109/32.92917.
- [339] A. J. Walker. An efficient method for generating discrete random variables with general distributions. *ACM Trans. Math. Softw.*, 3(3):253–256, 1977. doi:10.1145/355744.355749.
- [340] C. Walshaw. A multilevel algorithm for force-directed graph drawing. In J. Marks, editor, *Int. Symp. on Graph Drawing GD*, volume 1984 of *LNCS*, pages 171–182. Springer, 2000. doi:10.1007/3-540-44541-2_17.
- [341] Z. Wang, R. J. Thomas, and A. Scaglione. Generating random topology power grids. In *Hawaii Int. Int. Conf. on Systems Science HICSS*, page 183. Institute of Electrical and Electronics Engineers IEEE, 2008. doi:10.1109/HICSS.2008.182.
- [342] D. J. Watts and S. H. Strogatz. Collective dynamics of small-world networks. *Nature*, 393(6684):440–442, Jun 1998. doi:10.1038/30918.
- [343] D. J. Watts, P. S. Dodds, and M. E. Newman. Identity and search in social networks. *science*, 296(5571):1302–1305, 2002.
- [344] B. M. Waxman. Routing of multipoint connections. *IEEE J. Sel. Areas Commun.*, 6(9):1617–1622, 1988. doi:10.1109/49.12889.
- [345] N. C. Wormald. Models of random regular graphs. *London Mathematical Society Lecture Note Series*, 1999.
- [346] S. Xie, A. Kirillov, R. B. Girshick, and K. He. Exploring randomly wired neural networks for image recognition. In *IEEE/CVF Int. Conf. on Comp. Vision ICCV*, pages 1284–1293. Institute of Electrical and Electronics Engineers IEEE, 2019. doi:10.1109/ICCV.2019.00137.
- [347] J. Yang and J. Leskovec. Community-affiliation graph model for overlapping network community detection. In M. J. Zaki, A. Siebes, J. X. Yu, B. Goethals, G. I. Webb, and X. Wu, editors, *IEEE Int. Conf. on Data Mining ICDM*, pages 1170–1175. Institute of Electrical and Electronics Engineers IEEE, 2012. doi:10.1109/ICDM.2012.139.
- [348] J. Yang and J. Leskovec. Structure and overlaps of ground-truth communities in networks. *ACM TIST*, 5(2):26:1–26:35, 2014. doi:10.1145/2594454.
- [349] A. Yoo and K. W. Henderson. Parallel generation of massive scale-free graphs. *CoRR*, abs/1003.3684, 2010. URL: <http://arxiv.org/abs/1003.3684>, arXiv:1003.3684.
- [350] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016. URL: <http://doi.acm.org/10.1145/2934664>, doi:10.1145/2934664.
- [351] J. Zeng and H. Yu. A study of graph partitioning schemes for parallel graph community detection. *Parallel Comput.*, 58:131–139, 2016. doi:10.1016/j.parco.2016.05.008.
- [352] L. Zhang, M. Small, and K. Judd. Exactly scale-free scale-free networks. *CoRR*, abs/1309.0961, 2013. URL: <http://arxiv.org/abs/1309.0961>, arXiv:1309.0961.
- [353] J. Y. Zhao. Expand and contract: Sampling graphs with given degrees and other combinatorial families. *CoRR*, abs/1308.6627, 2013. URL: <http://arxiv.org/abs/1308.6627>, arXiv:1308.6627.

