Uniform Sampling of Simple Graphs with Prescribed Degrees and Experimental Algorithms for External Memory

Dissertation

zur Erlangung des Doktorgrades

der Naturwissenschaften

vorgelegt beim Fachbereich 12

der Johann Wolfgang Goethe–Universität

in Frankfurt am Main

von

Hung The Tran

aus Wernigerode

Frankfurt (2022)

(D 30)

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

Dekan: Prof. Dr. M. Möller

Gutachter: Prof. Dr. M. Hoefer, Prof. Dr. U. Meyer

Datum der Disputation: 09.05.2023

Acknowledgements

First and foremost, I would like to thank Ulrich Meyer for granting me a position in his working group, guiding my research and being very understanding in times when I was not particularly healthy. I would also like to thank Martin Hoefer for agreeing to review this thesis.

Thanks to all my coauthors, especially Manuel, who sparked my interest in the field of algorithm engineering. I really enjoyed the time we have worked together, the discussions we had and the opportunities to learn from – and with you.

Furthermore, I would like to thank all my coworkers and office mates, especially Alex, Arlene, Daniel and David – thank you for the countless conversations and the many lunches we shared.

Finally, I would like to thank my family and friends – who provided me with the right amount of distractions and support in my endeavors. Though, I especially thank my father, who –even in hard times– has always put me first.

Hung Tran
21. November 2022

Deutsche Zusammenfassung

Die Emergenz digitaler Netzwerke ist auf die ständige Entwicklung und Transformation neuer Informationstechnologien zurückzuführen. Dieser Strukturwandel führt zu äußerst komplexen Systemen in vielen verschiedenen Lebensbereichen. Es besteht daher verstärkt die Notwendigkeit, die zugrunde liegenden wesentlichen Eigenschaften von realen Netzwerken zu untersuchen und zu verstehen. In diesem Zusammenhang wird die Netzwerkanalyse als Mittel für die Untersuchung von Netzwerken herangezogen und stellt beobachtete Strukturen mithilfe mathematischer Modelle dar. Hierbei, werden in der Regel parametrisierbare Zufallsgraphen verwendet, um eine systematische experimentelle Evaluation von Algorithmen und Datenstrukturen zu ermöglichen.

Angesichts der zunehmenden Menge an Informationen, sind viele Aspekte der Netzwerkanalyse datengesteuert und zur Interpretation auf effiziente Algorithmen angewiesen. Algorithmische Lösungen müssen daher sowohl die strukturellen Eigenschaften der Eingabe als auch die Besonderheiten der zugrunde liegenden Maschinen, die sie ausführen, sorgfältig berücksichtigen. Die Generierung und Analyse massiver Netzwerke ist dementsprechend eine anspruchsvolle Aufgabe für sich.

Die vorliegende Arbeit bietet daher algorithmische Lösungen für die Generierung und Analyse massiver Graphen. Zu diesem Zweck entwickeln wir Algorithmen für das Generieren von Graphen mit vorgegebenen Knotengraden, die Berechnung von Zusammenhangskomponenten massiver Graphen und zertifizierende Grapherkennung für Instanzen, die die Größe des Hauptspeichers überschreiten. Unsere Algorithmen und Implementierungen sind praktisch effizient für verschiedene Maschinenmodelle und bieten sequentielle, Shared-Memory parallele und/oder I/O-effiziente Lösungen.

Maschinenmodelle:
☞ *Abschnitt 1.2*

Struktureller Aufbau

Wir führen zunächst das *External Memory Model* (*Emm*) ein und fassen die Resultate dieser Arbeit nachfolgend zusammen. Der Hauptteil (Kapitel 2 bis 7) ist hierbei zweigeteilt: Kapitel 2 bis 5 stellen den ersten Teil dar und befassen sich mit dem Problem der Generierung von uniform zufälligen Graphen mit vorgegebenen Knotengraden; Kapitel 6 und 7 stellen den zweiten Teil dar und befassen sich mit Algorithmen zur Analyse und Verarbeitung von massiven Graphen.

External Memory Model

Wir verwenden das anerkannte *External Memory Model* (*Emm*) von Aggarwal und Vitter [1]. Das Modell abstrahiert Speicherhierarchien moderner Computer in ein theoretisches Framework. Es modelliert eine Speicherhierarchie aus zwei Schichten, dem schnellen Internspeicher, welches bis zu M Datenelemente umfasst, sowie dem langsamen Externspeicher unbegrenzter Größe. Die Ein- und Ausgabe wird im Externspeicher gespeichert, und Daten werden zwischen den Speicherschichten mittels sogenannter I/Os übertragen, wobei jeweils ein Block von B aufeinanderfolgenden Datenelementen verschoben wird. Im *Emm* werden Berechnungen nur mit Daten im Internspeicher durchgeführt, und die Performanz eines Algorithmus wird in der Anzahl der durchgeführten I/Os gemessen.

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

- Lesen und Schreiben von n Werten eines zusammenhängenden Bereichs benötigt $\mathrm{scan}(n) = \Theta(\frac{n}{B})$ I/Os.

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

- Vergleichsbasiertes Sortieren eines zusammenhängenden Bereichs mit n Werten benötigt $\mathrm{sort}(n) = \Theta(\frac{n}{B}\log_{M/B}(\frac{n}{B}))$ I/Os.
- Prioritätswarteschlangen können n Einfüge- und Löschoperationen in $\mathrm{sort}(n)$ I/Os ausführen [14, 13].

Für praktische Werte von n, B und M ist das Sortieren vergleichbar mit nur wenigen Scandurchläufen und wird gegenüber zufälligen I/Os stark bevorzugt. Hierbei stellt die Sortierkomplexität eine untere Schranke für viele nicht-triviale algorithmische Probleme im Externspeicher dar [1, 133].

Edge Switching und Global Edge Switching (Kapitel 2 und 3)

LFR: ☞ *Abschnitt 2.3*

Kapitel 2 stellt EM-LFR vor, eine komplexe Pipeline mehrerer I/O-effizienter Subroutinen zur Generierung großer *LFR* Graphinstanzen, die die Größe des Hauptspeichers überschreiten. In EM-LFR besteht die größte Herausforderung darin, einen simplen Graphen mit vorgegebener Gradsequenz zu generieren. Um dies zu realisieren, folgt EM-LFR dem *Fixed-Degree-Sequence-Model* (*FDSM*) und unterteilt die Generierung in zwei Schritte auf. Zuerst, wird deterministisch ein simpler Graph mit der vorgegebenen Gradsequenz erzeugt. Anschließend wird der Graph mittels *Edge Switching* (*ES*) pertubiert, wobei die Knotengrade unverändert bleiben. In der ursprünglichen Formulierung von EM-LFR werden die Algorithmen EM-HH und EM-ES verwendet um diesen Prozess zu realisieren. Hierbei ist EM-HH ein I/O-effizienter Generator nach Havel und Hakimi [95, 89] und EM-ES eine I/O-effiziente Implementierung von *ES*. Da die von EM-HH produzierten Graphen einen starken Bias aufweisen, sind viele Perturbationsschritte durch *Edge Switching* nötig. Wir betrachten daher eine alternative Art der Generierung mittels des *Configuration Models* (*CM*) und passen die Pipeline entsprechend an:

EM-HH und EM-ES: ☞ *Abschnitte 2.4 und 2.5*

EM-CM/ES: ☞ *Abschnitt 2.6*

- *EM-CM/ES:* Wir stellen EM-CM/ES als Alternative zu der ursprünglich vorgeschlagenen Kombination von EM-HH und EM-ES vor. Hierfür implementieren wir einen I/O-effizienten Generator für das *Configuration Model* und substituieren EM-HH im ersten Schritt. Da der generierte Graph allerdings in der Regel nicht simpel ist, passen wir EM-ES an, um Multi-Graphen verarbeiten zu können.

  Um zu einem simplen Graphen zu gelangen, akzeptiert der Algorithmus alle *ES* Tausche, die weder Multi-Kanten noch Eigenschleifen erzeugen. Zur Beschleunigung dieses Prozesses, werden illegale Kanten gezielt häufiger in den *ES* Tauschen anvisiert.

  Die so erhaltenen Graphinstanzen haben jedoch weiterhin einen Bias [5, 17] und müssen daher weiter mit EM-ES randomisiert werden. Unsere Experimente zeigen, auch wenn dieser Ansatz aufwändiger ist, dass EM-CM/ES schneller zu einem uniformen Sample konvergieren kann als durch die zuvor vorgeschlagene Kombination von EM-HH und EM-ES.

Kapitel 3 betrachtet Shared-Memory parallele Ansätze zur Randomisierung simpler Graphen. Wir implementieren einen einfachen parallelen Algorithmus für *Edge Switching Markov Chain* (*ES-MC*), der aufgrund der in *ES-MC* auftretenden Abhängigkeiten nicht gut skalieren kann. Um dieses Problem zu umgehen und mehr Parallelität zu ermöglichen, schlagen wir *Global Edge Switching Markov Chain* (*G-ES-MC*), eine Variante von *ES-MC* mit einfacheren Abhängigkeiten, vor. Ähnlich wie bei *ES-MC* zeigen wir, dass auch *G-ES-MC* zu einer uniformen Verteilung konvergiert. Unsere Experimente deuten an, dass *G-ES-MC* höchstens die gleiche Anzahl an Tauschen benötigt wie *ES-MC* und zeigen außerdem die Effizienz und Skalierbarkeit unserer implementierten Algorithmen.

- *RobinES and GlobalES:* Wir präsentieren RobinES und GlobalES als sequentielle Lösungen von *ES-MC* und *G-ES-MC*. Unsere Implementierungen verwenden Hash-Sets, die Kanteneinfügung, -löschung und Existenzabfragen in erwarteter konstanter Zeit unterstützen, weitere Hilfsdatenstrukturen für das zufällige Ziehen von Kanten und Prefetching um die zufälligen I/Os auf den Hauptspeicher zu beschleunigen.

  ES-MC und G-ES-MC: ☞ *Abschnitt 3.2*

  In einem Vergleich mit *NetworKit* [171] und *Gengraph* [181] zeigen wir, dass unsere Implementierungen 15-50 Mal schneller als *NetworKit* und 5-10 Mal schneller als *Gengraph* sind. Für große Graphen schneidet GlobalES besser ab als RobinES, da das Permutieren der Kanten effizienter ist als das mehrfache Ziehen einzelner zufälliger Kanten. RobinES hingegen zeigte eine bessere Performanz für kleine Graphen.

- *EagerES and SteadyGlobalES:* Wir stellen EagerES, eine vereinfachte Parallelisierung von *ES-MC*, als Performancebaseline unserer parallelen Algorithmen vor. Der Algorithmus verwendet ein paralleles Hash-Set und setzt nur eine implizite Synchronisationsstrategie ein, bei der jeder Prozessor seine *ES* Tausche unabhängig ausführt. Dies führt jedoch zu einer Ausführungsreihenfolge, die vom Prozess-Scheduler abhängig ist, und bildet daher *ES-MC* nicht vollkommen exakt ab.

  EagerES: ☞ *Abschnitt 3.4.1*

  Für *G-ES-MC* entwickeln wir SteadyGlobalES, eine parallele Umsetzung, die uniform zufällige *ES* Globaltausche verarbeitet. Hierbei sind *ES* Globaltausche so konzipiert, dass sie im Vergleich zu *ES-MC* einfachere Abhängigkeiten vorweisen, die wir in zwei Typen unterteilen: Lösch- und Einfüge-Abhängigkeiten. Ein *ES* Globaltausch wird dann in mehreren Runden parallel verarbeitet. Trotz der Komplexität des Algorithmus zeigen unsere Experimente, dass SteadyGlobalES nur höchstens einen Faktor 2 langsamer als EagerES ist, obwohl das *G-ES-MC* exakt abgebildet wird.

  SteadyGlobalES: ☞ *Abschnitt 3.4.2*

Curveball und Global Curveball (Kapitel 4)

Kapitel 4 präsentiert eine Reihe von Algorithmen für einen alternativen Randomisierungsprozess, nämlich *Curveball* (*CB*) [174, 46] und *Global Curveball* (*G-CB*) [47, 48], vor. *CB* geht ähnlich wie *ES* vor, wählt aber stattdessen zwei zufällige Knoten $u \neq v$ und führt einen *CB* Tausch aus, bei dem die Nachbarschaften von u und v randomisiert werden. Dazu sammelt *CB* zunächst alle nicht-gemeinsamen Nachbarn beider beteiligten Knoten, entfernt u und v aus dieser Liste, und verteilt sie zufällig neu ohne dabei die Knotengrade zu verändern. Da die Nachbarschaften beider Knoten im Ganzen berücksichtigt werden können, führt ein einzelner *CB* Tausch potentiell zu einer größeren Veränderung im Graphen im Vergleich zu einem *ES* Schritt.

Während *CB* die partizipierenden Knoten uniform zieht, fasst *G-CB* mehrere *CB* Tausche in einen Superschritt zusammen. In diesem sogenannten Globaltausch ist jeder Knoten genau einmal in einem *CB* Tausch beteiligt[1]. Wir erweitern *G-CB* auf ungerichtete Graphen und zeigen, dass auch dieser Prozess zur uniformen Verteilung konvergiert. Darüber hinaus deuten unsere Experimente an, dass *G-CB* besser abschneidet als *CB*. Mit weiteren Experimenten zeigen wir die Effizienz und Skalierbarkeit unserer Algorithmen, auch für Instanzen, die die Größe des Hauptspeichers überschreiten.

EM-CB:
☞ *Abschnitt 4.4.1*

TFP:
☞ *Abschnitt 4.2.2*

- *EM-CB und IM-CB:* Wir präsentieren EM-CB, einen I/O-effizienten sequentiellen Algorithmus für *CB*. Da Änderungen in den Nachbarschaften der beteiligten Knoten durch einen *CB* Tausch ebenfalls berücksichtigt werden müssen, verzichtet EM-CB auf eine statische Graphdatenstruktur bei der gegebenenfalls unstrukturierte Zugriffe durchgeführt werden. Durch die Anwendung von *Time Forward Processing* (*TFP*) umgehen wir diese unstrukturierten Zugriffsmuster und verwalten den Graphen stattdessen dynamisch. Dazu interpretieren wir jeden *CB* Tausch als einen Zeitpunkt in der Berechnung des Algorithmus. Zu jedem *CB* Tausch werden dann nur die Nachbarschaften beider Tauschpartner benötigt, die mittels *TFP* bereitgestellt werden. Um dies zu realisieren, führt EM-CB *CB* Tausche in Batches durch, wobei für jeden Batch alle Tauschpartner zufällig gezogen und anschließend in Hilfsdatenstrukturen verwaltet werden. Diese werden dann genutzt um *TFP* Nachrichten entsprechend weiterzuleiten.

  IM-CB:
  ☞ *Abschnitt 4.4.2*

  Für den Fall, dass die Speicherzugriffe nicht der limitierender Faktor sind, schlagen wir IM-CB als schnellere Alternative zu EM-CB vor. Durch den Verzicht auf den für *TFP* benötigten Datenstrukturen und Verwendung einer klassischen Adjazenzvektor-Repräsentation, akzeptiert IM-CB die unstrukturierten Zugriffe und zeichnet sich daher besonders bei kleinen und mittelgroßen Graphen aus.

EM-GCB:
☞ *Abschnitt 4.4.3*

- *EM-GCB und EM-PGCB:* EM-CB ist unser I/O-effizienter Algorithmus für ungerichtetes *G-CB*. Durch das Ausnutzen der zusätzlichen Struktur von Globaltauschen können wir auf die Hilfsdatenstrukturen von EM-CB und IM-CB verzichten. Genauer gesagt, interpretieren wir einen Globaltausch als eine zufällige Permutation

[1]Der Einfachheit halber nehmen wir an, dass die Anzahl der Knoten gerade ist; für den allgemeinen Fall siehe Abschnitt 4.4.3.

der Knoten und repräsentieren diesen implizit durch Verwendung geeigneter Hash-Funktionen.

Weiteres Engineering führt zu EM-PGCB, einer parallelen Erweiterung von EM-GCB. Um parallele Verarbeitung zu ermöglichen, unterteilen wir die Globaltausche in noch kleinere sogenannte *Makrochunks*, die einzeln im Hauptspeicher gehalten werden. Diese Makrochunks werden dann ebenfalls in kleinere *Mikrochunks* unterteilt, deren Größe so gewählt wird, dass fast alle *CB* Tausche unabhängig voneinander parallel ausgeführt werden können. In den seltenen Fällen, in denen Abhängigkeiten auftreten, greifen wir auf Work Stealing zurück, um unnötige Wartezeiten zu vermeiden. Wir weisen experimentell nach, dass in einigen Fällen, EM-PGCB im Vergleich zu EM-ES um fast eine Größenordnung schneller ist und eine vergleichbare Randomisierungsqualität erreicht.

EM-PGCB:
☞ Abschnitt 4.4.4

Uniformes Generieren von Power-law Graphen (Kapitel 5)

Während in Kapitel 2 bis 4 Markow-Ketten basierende Monte-Carlo Verfahren (MCMC) zur Generierung simpler Graphen mit vorgegebener Gradsequenz dargelegt werden behandelt Kapitel 5 einen exakten uniformen Generator.

Wir stellen den Inc-Powerlaw Algorithmus in seiner vollständigen Beschreibung vor. Inc-Powerlaw verbessert den Pld Algorithmus [78] durch Anwendung von *incremental relaxation*, einer kürzlich entwickelten Technik von Arman et al. [17]. An den Stellen, bei denen incremental relaxation verwendet wird, bestimmen wir die Reihenfolge, in der die relevanten Graphsubstrukturen relaxiert werden sollen, wie sie zu zählen sind und bestimmen geeignete untere Schranken. In unserer Untersuchung haben wir festgestellt, dass Inc-Powerlaw in der usrprünglichen Formulierung mittels incremental relaxation zu viele Runs verwarf und daher keine lineare Laufzeit aufwies. Um dieses Problem zu lösen, führen wir weitere Switchings zum Algorithmus ein und weisen nach, dass die Rejection-Wahrscheinlichkeit hinreichend klein ist.

Um unsere Erkenntnisse zu verifizieren, entwickeln wir eine Inc-Powerlaw Implementierung, die anschließend parallelisiert wird. In unserer empirischen Studie stellen wir fest, dass Inc-Powerlaw bei kleinen Durchschnittsgraden sehr effizient ist und bei größeren Durchschnittsgraden höhere Konstanten aufweist. Außerdem bestätigen wir empirisch die lineare erwartete Laufzeit von Inc-Powerlaw.

- *Inc-Powerlaw:* Inc-Powerlaw erzeugt zunächst einen Zufallsgraph nach dem *Configuration Model* und wandelt illegale Strukturen in legale um. Dazu werden mehr als 20 verschiedene Switchings verwendet. Um Uniformität zu gewährleisten, wird Rejection-Sampling verwendet, so dass alle generierten Graphen uniform in ihrer jeweiligen Klasse sind. Für jedes zufällig gezogene Switching wird gewürfelt ob es verworfen (f-rejection oder b-rejection) wird und der Algorithmus dadurch neugestartet wird.

Inc-Powerlaw:
☞ Abschnitt 5.2

Wir stellen alle notwendigen Voraussetzungen für die Wiederherstellung der linearen Laufzeit von Inc-Powerlaw vor. In Phase 4 führt die Hinzunahme von incremental relaxation zur Verringerung des Rechenaufwands, die mit der Erhöhung der b-rejection Wahrscheinlichkeit einhergeht. Dies führte zu intolerabel vielen Rejections des Algorithmus. Wir lösen dieses Problem durch Hinzufügen von drei Booster-Switchings (t_a, t_b und t_c zu dem ursprünglich verwendeten t-Switching). Alle vier Switchings erzeugen die sogenannte *triplet*-Struktur und möglicherweise zusätzliche Kanten in Abhängigkeit vom gezogenen Switching. Durch Berechnung der entsprechenden Konstanten und unteren Schranken zeigen wir, dass die Wahrscheinlichkeit für eine b-rejection in Phase 4 dann $o(1)$ ist.

In ähnlicher Weise wird in Phase 5 von Inc-Powerlaw die Wahrscheinlichkeit einer b-rejection durch Hinzunahme von incremental relaxation beeinträchtigt. Hier fügen wir Booster-Switchings (type-III, type-IV, type-V, type-VI und type-VII Switchings) hinzu, um eine *doublet*-Struktur zu erzeugen und die Wahrscheinlichkeit einer Rejection zu verringern. Auf analoge Weise beweisen wir anschließend, dass die Wahrscheinlichkeit für eine b-rejection erneut $o(1)$ ist.

Parallelisierung: ☞ *Abschnitt 5.4.2*

- *Intra-Run und Inter-Run Parallelisierung:* In unserer experimentellen Auswertung zeigen wir, dass das Sampling des anfänglichen Multi-Graphen und die Konstruktion der geeigneten Datenstrukturen die dominierenden Faktoren im Algorithmus sind. Unsere Implementierung inkorporiert daher Parallelisierungsstrategien, um die Auswirkungen des oben genannten Flaschenhalses zu verringern. Wir betrachten zwei orthogonale Strategien: Intra-Run und Inter-Run.

  Während Intra-Run die Konstruktion des Multi-Graphen und seiner repräsentativen Datenstrukturen direkt parallelisiert, startet Inter-Run mehrere Runs und akzeptiert den ersten akzeptierenden Run. Wir verwenden Synchronisation, um einen Bias zugunsten von schnelleren Runs zu vermeiden: alle Prozessoren weisen ihren Runs global eindeutige Indizes zu und der akzeptierende Run mit kleinstem Index wird zurückgegeben.

Zusammenhangskomponenten im Externspeicher (Kapitel 6)

Kapitel 6 präsentiert eine empirische Untersuchung des Problems Zusammenhangskomponenten (*CC*) im Externspeicher zu finden. Wir betrachten mehrere Algorithmen, die entweder theoretisch effizient sind oder praktisch vielversprechend erscheinen. Unsere Experimente werden dann auf einer Vielzahl von verschiedenen Graphklassen durchgeführt, darunter populäre Zufallsmodelle wie Gilbert Graphen, *Random Geometric Graph*s (*RGG*s) und *Random Hyperbolic Graphs* (*RHG*s).

Borůvka: ☞ *Abschnitt 6.3*

- *Borůvka and Randomized-Borůvka:* Wir liefern eine Implementierung einer für das Externspeicher ausgelegten Variante des Borůvka Algorithmus [51] als Vergleichspunkt für weitere Externspeicher-Algorithmen. Zur Berechnung der Zusammenhangskomponenten führt der Algorithmus wiederholt *Borůvka steps* aus, die die Größe der Knotenmenge des Graphen reduziert. Der angepasste Algorithmus weist jedoch eine erhöhte Komplexität auf, die unter anderem die

Verwendung zusätzlicher Datenstrukturen erfordert. Dies führt jedoch zu einer impraktikablen Performanz in realen Anwendungen.

  Um den Effekt der zusätzlichen Datenstrukturen zu verringern, schlagen wir daher den ähnlichen aber weniger komplizierten Algorithmus RANDOMIZED-BORŮVKA vor. RANDOMIZED-BORŮVKA funktioniert im Wesentlichen wie BORŮVKA, erzeugt aber kleinere, leicht-handhabbare Subgraphen, die effizienter kontrahiert werden können. Beide Algorithmen können jedoch nicht mit den anderen Kandidaten konkurrieren.

RANDOMIZED-BORŮVKA: ☞ *Abschnitt 6.3*

- *SIBEYN:* SIBEYN ist ein Algorithmus zur Berechnung von minimalen Spannwäldern (*MSF*), der weitere Optimierungsmöglichkeiten aufweist, wenn lediglich Zusammenhangskomponenten berechnet werden müssen. Die Simplizität von SIBEYN ermöglichte eine entsprechende Umsetzung, die zu unserer hocheffizienten Implementierung führt. Der Algorithmus lässt dazu wiederholt Knoten eine beliebige inzidente Kante wählen, die anschließend kontrahiert wird. Dieser Prozess wird durch *Time Forward Processing* (*TFP*) realisiert und erzeugt dabei in der Praxis vergleichsweise kleine I/O-Volumen.

  In unseren Experimenten betrachten wir verschiedene Strategien der Kantenauswahl und eine Vielzahl von Möglichkeiten, die Kontraktionen durchzuführen. Dazu zählt das etwaige Verwenden von Prioritätswarteschlangen oder Buckets.

SIBEYN: ☞ *Abschnitt 6.3*

- *KARGER-KLEIN-TARJAN:* Wir betrachten den bekannten Algorithmus von Karger, Klein und Tarjan [108] für das *MSF*-Problem in einem generellen Framework. Auch für den Externspeicher existiert eine entsprechende Variante; die wiederum auf das Finden von Zusammenhangskomponenten übertragen werden kann. Unser Framework geht rekursiv vor und erfordert einige Subroutinen: das Reduzieren der Knotenmenge durch Kontraktionen, die Berechnung von zufälligen Stichproben der Kanten und die Zusammenführung rekursiv berechneter Teillösungen des *CC*-Problems. Aufgrund der erhöhten Variabilität sind wir in der Lage eine große Bandbreite an Parameterkombinationen für KARGER-KLEIN-TARJAN in Betracht zu ziehen. Dazu zählt unter anderem das Verwenden von BORŮVKA, RANDOMIZED-BORŮVKA und SIBEYN zur Kontraktion des Graphen und adaptives Samplen der Kanten.

  Obwohl KARGER-KLEIN-TARJAN der theoretisch effizienteste Algorithmus ist, ist für eine praktische Implementierung viel Tuning nötig. Dies ist zum Teil auf die rekursive Natur des Algorithmus und die zusätzlich benötigten Hilfsdatenstrukturen zurückzuführen.

KARGER-KLEIN-TARJAN: ☞ *Abschnitt 6.3*

In unseren Experimenten haben wir festgestellt, dass SIBEYN aufgrund seiner Simplizität eine gute Wahl ist. Die Implementierung erfordert im Wesentlichen nur die Verwendung einer einzigen Externspeicher-Prioritätswarteschlange. Daher kann ein konkurrierender Algorithmus nur wenige Operationen durchführen, bevor er gegen SIBEYN verliert. KARGER-KLEIN-TARJAN ist dennoch ein kompetitiver Algorithmus; mit den richtigen Subroutinen und Parametern bietet der Algorithmus eine robuste und vergleichbar gute Lösung.

Zertifizierende Grapherkennung im Externspeicher (Kapitel 7)

Kapitel 7 stellt I/O-effiziente zertifizierende Algorithmen zur Erkennung von verschiedenen Graphklassen vor. Für einen Graphen mit n Knoten und m Kanten benötigen unsere Algorithmen $\mathcal{O}(\text{sort}(n+m))$ I/Os im worst-case oder mit hoher Wahrscheinlichkeit für bipartite und Bipartite-Chain-Graphen. Im Falle der Zugehörigkeit zur Graphklasse, wird ein YES-Zertifikat zurückgegeben, das die Graphklasse charakterisiert. Im Gegensatz dazu wird im Fall der Nicht-Zugehörigkeit ein $\mathcal{O}(1)$ großes NO-Zertifikat für alle Graphklassen bis auf den bipartiten Fall zurückgegeben. Wir passen die für den Internspeicher ausgelegten Algorithmen von Heggernes und Kratsch [96] an das Externspeicher an. Dazu verwenden wir Standardtechniken wie das *Time Forward Processing* und Euler-Tour Berechnungen. Für alle Graphklasesen nutzen wir ihre wichtigsten strukturellen Eigenschaften aus, um I/O-effiziente Algorithmen zu entwickeln.

Zertifizieren von Split-Graphen: ☞ *Abschnitt 7.2.1*

- *Split-Graphen:* Split-Graphen sind Graphen, deren Knotenmenge in (K, I) partitioniert werden können, sodass K eine Clique und I eine unabhängige Menge ist. Wir nutzen folgende weitere Erkenntnisse aus: (i) die maximale Clique besteht aus Knoten mit den höchsten Knotengraden; und (ii) jede nicht-abnehmende Knotengradordnung eines Split-Graphen bildet eine sogenannte *pefect elimination ordering*. Die Berechnung dieser Ordnung und das entsprechende Relabeling des Graphen erleichtert weitere notwendige Subroutinen des Algorithmus, unter anderem, ob K tatsächlich eine Clique und I tatsächlich eine unabhängige Menge ist.

Zertifizieren von Threshold-Graphen: ☞ *Abschnitt 7.2.2*

- *Threshold-Graphen:* Threshold-Graphen sind Split-Graphen mit der weiteren Eigenschaft, dass die unabhängige Menge I eine sogenannte *nested neighborhood ordering* hat. Sie sind zusätzlich durch den folgenden Graphgenerierungsprozess gekennzeichnet: füge wiederholt universelle oder isolierte Knoten zu einem ursprünglich leeren Graphen hinzu. Durch das Relabeling können wir diese Eigenschaft in umgekehrter Reihenfolge I/O-effizient überprüfen, indem wir wiederholt universelle und isolierte Knoten aus dem Eingabegraphen entfernen.

Zertifizieren von Trivially-Perfect-Graphen: ☞ *Abschnitt 7.2.3*

- *Trivially-Perfect-Graphen:* Trivially-Perfect-Graphen sind Graphen bei denen die Größe der unabhängigen Menge jedes induzierten Subgraphens mit der Anzahl nicht-erweiterbaren Cliquen übereinstimmt. Ähnlich wie bei Split-Graphen nutzen wir, dass Trivially-Perfect-Graphen eine besondere Eigenschaft in ihren Knotengraden haben: Jede nicht-zunehmende Knotengradordnung ist eine sogenannte *universal-in-a-component-ordering*. Mit *Time Forward Processing* übersetzen wir das iterative Labelingschema von Heggernes und Kratsch [96] und verifizieren diese Eigenschaft I/O-effizient.

Zertifizieren von bipartiten Graphen sowie Bipartite-Chain-Graphen: ☞ *Abschnitt 7.2.4*

- *Bipartite Graphen und Bipartite-Chain-Graphen:* Bipartite-Chain-Graphen sind bipartite Graphen in denen die Partitionen eine nested neighbood ordering auf-

weisen. Wir entwickeln daher zunächst einen I/O-effizienten Zertifizierungsalgorithmus für die Erkennung von bipartiten Graphen. Anstelle von Graphtraversierungsalgorithmen verwenden wir Spannwälder und Stapelverarbeitung um einen I/O-effizienten Algorithmus für bipartite Graphen zu realisieren. Durch die Kombination dieses und des entwickelten Algorithmus für Threshold-Graphen präsentieren wir außerdem einen Algorithmus für den Fall von Bipartite-Chain-Graphen, der ebenfalls $\mathcal{O}(\mathrm{sort}(n+m))$ I/Os mit hoher Wahrscheinlichkeit benötigt.

Contents

1 Introduction

The emergence of digital networks is rooted in the ever on-going advancement and transformation of new information technologies. This structural change gives rise to deeply complex systems in many different areas in life. As such, a necessity to study and understand the underlying key properties of real-world networks persists. In this context, network analysis is regarded as a methodological perspective for the study of networks and depicts the observed entities using mathematical models. Commonly, random graphs are used to provide an adjustable and controllable means to obtain synthetic data to enable systematic experimental evaluations of algorithms and data structures.

Nowadays, with the explosion of data, many aspects of network analysis are data-driven and rely on efficient algorithms to make sense of the information at hand. Algorithmic solutions, therefore, need to carefully consider both the structural properties of the input and the peculiarities of the underlying machines that execute them. The generation and analysis of massive networks is therefore a challenging task in and of itself.

The present thesis, thus, provides algorithmic solutions to generate and analyze graphs at scale. To this end, we develop algorithms for the sampling of graphs with prescribed degrees, the computation of connected components of massive graphs and certifying graph recognition for instances exceeding the size of main memory. Our algorithms and implementations are practically efficient for various machine models providing sequential, shared-memory parallel and I/O-efficient solutions.

1.1 Motivation

Networks have an exceptional impact on science, technology, and society [22]. They lie at the heart of complex systems that are used to effectively model many aspects of human life. In order to make sense of these systems and understand their intricate features, a careful systematic investigation and interpretation is crucial. This fueled the emergence of a new scientific discipline called *network science* which concerns itself with the study of relational data [40]. Its methodological approach, almost inherently, is interdisciplinary and applicable to many other fields.

In this domain, *network analysis* emerged from the study of graphs and provides a collection of techniques assisting in the investigation of networks. It provides a means to analyze and depict the relations of the entities in question. There are many insights that can be gained; among these, global metrics that quantify some information of the network as a whole, importance features of the contained entities and many more. With the ever growing volume of network data, however, processing networks efficiently becomes increasingly difficult. As such, designing and implementing efficient algorithms for larger scales is of pivotal importance to enable the analysis of massive networks. This, in part, requires algorithmic solutions to carefully consider both the structural properties of the given network and the peculiarities of the underlying machines that execute them. To further enhance these considerations and improve performance predictions for the designed algorithms, *network models* are used to generate supplemental synthetic data that can be incorporated into experimental campaigns.

In this context, random graphs commonly serve as a model for networks. They are subject to many design decisions in an attempt to depict and reproduce the observed features of real-world networks. To this end, many popular models are parameterized and provide a certain degree of flexibility to enable a large variety of generatable instances. This, therefore, motivates the development of efficient algorithms for the generation of random graphs at scale.

1.1.1 Goals

This thesis studies algorithmic aspects of network analysis, in particular for large processing scales. While network analysis expresses itself in many directions and can be considered through multiple different lenses, we aim to provide algorithmic means to study and understand networks at hand. In addition to designing and implementing tools for the examination of any given network, we also aim to develop methods to reliably generate comparable and controllable synthetic network data which are typically rooted in random graph models.

To this end, we study *Connected Components* and certifying graph recognition algorithms, and consider the generation of uniform simple graphs with prescribed degrees as a popular synthetic random graph model. In order to cover a variety of applications, our algorithms are designed according to different models of computation providing sequential, shared-memory parallel and/or I/O-efficient solutions.

Comprehensive Summary: ☞ *Chapter 8*

1.1.2 Outline

We organize the present thesis as follows:

- Section 1.2 provides the foundations of this thesis, discusses *algorithm engineering* as our design methodology and presents the used machine models thereafter.
- Sections 1.3 to 1.5 briefly describe the context of the covered topics of this thesis.
- Chapters 2 to 7 constitute the main part of the present thesis. We provide a brief overview in Section 1.6.

  The main part is divided into two larger sections: Chapters 2 to 5 present the first part and deal with the problem of uniformly sampling simple graphs with prescribed degrees; Chapters 6 and 7 present the second part and deal with experimental algorithms for external memory.
- Chapter 8 summarizes the results of the main part of the thesis and concludes with further directions for future research.

1.2 Algorithm Engineering

This dissertation adopts the *algorithm engineering* methodology [161, 162] which addresses the apparent gap between algorithm theory and practical applications. Algorithm engineering proposes a feedback loop of design, analysis, implementation and experiments combining algorithm theory and experimental algorithmics.

Aiming at practicality, applications motivate the use of abstract models to provide a simple but adequate representation of real machines. This enables sufficient tractability to perform theoretical analyses and derive provable performance guarantees of the algorithms. Following implementations are then evaluated using systematic experiments to provide new insights that potentially inspire incremental improvements in both implementation and algorithm design.

As any problem is generally specified by its application domain, one-fits-all algorithmic solutions do not suffice and may require the use of more advanced models of computation (Sections 1.2.1 and 1.2.2). This can be illustrated by the classical *unit-cost Random-Access Machine* (*unit-cost RAM*) model which, depending on the application, fails to capture the complexity of real machines. It assumes a single processor containing multiple registers, which can perform elementary operations in one unit of cost. Due to its basic formulation, its main strengths are its simplicity and universal applicability. The model, however, clearly does not reflect real costs of operations, as it assumes an equal cost for memory accesses and basic arithmetic instructions. This contradicts the fact that modern machines have a *memory hierarchy* to provide cost-efficient trade-offs in size and speed leading to non-trivial penalties.

unit-cost RAM

Additionally, the *unit-cost RAM*, by design, does not incorporate advanced features of modern computers and therefore fails to benefit from natural optimization opportunities such as parallelism. To accommodate both of these features, we introduce two related advanced models of computation, the *External Memory Model* and the *Parallel Random-Access Machine*.

1.2.1 External Memory Model

External Memory Model

We use the commonly accepted *External Memory Model* (*EMM*) by Aggarwal and Vitter [1], a theoretical framework that abstracts memory hierarchies to an idealized setting. The model consists of a memory hierarchy of two layers, the fast internal memory which can hold up to M data items and the slow external memory of unbounded size. The input and output are stored in external memory and data is moved between layers using so-called I/Os, each of which moves a block of B consecutive items at a time. In the *EMM*, computation is only performed on data in internal memory, and an algorithm's performance is measured in the number of I/Os it performs. Common basic primitives and their I/O-complexities are:

M: internal memory size

B: block size

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

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

- Reading or writing n contiguous items requires $\mathrm{scan}(n) = \Theta(\frac{n}{B})$ I/Os.
- Comparison-based sorting of n items requires $\mathrm{sort}(n) = \Theta(\frac{n}{B}\log_{M/B}(\frac{n}{B}))$ I/Os.
- Pushing and removing n items from a priority-queue requires $\mathrm{sort}(n)$ I/Os [14, 13].

For all realistic values of n, B, and M sorting is comparable to only very few scanning rounds and is strongly preferred over random I/Os. Its complexity constitutes a lower bound for many non-trivial algorithmic tasks in external memory [1, 133].

The actual implementation of external memory algorithms is quite demanding however. There exist, for this reason, two open-source libraries that efficiently implement external memory algorithms and data structures for general purpose use, namely *STXXL* [60] and *TPIE* [179, 16].

1.2.2 Parallel Random-Access Machine

In the advent of substantial improvements in multiprocessing systems and multi-core machines, parallel algorithms gain an increasing amount of traction both in theory and practice. In this context, the most commonly used parallel machine model, the *Parallel Random-Access Machine* (*PRAM*) [103], is an abstract model to capture shared-memory parallel systems.

PRAM

The model consists of P sequential *processing units* (PUs), which each have a small number of local registers, and a *globally shared memory*. Communication between PUs is not modeled explicitly but can be achieved by utilizing the shared memory. Typically, PUs are clocked synchronously, meaning computation proceeds in synchronized fashion. In this case, however, concurrent read and write conflicts can occur, and are only solved by using a *PRAM* subtype that explicitly designates which concurrent access patterns are allowed and how to resolve them.

1.3 Uniform Sampling of Simple Graphs with Prescribed Degrees

Network science provides tools and guidelines for the modeling, analysis and understanding of complex systems. To this end, many network models have been proposed to represent real-world systems and capture their respective features. In this context, the use of null models became increasingly popular as they reveal and quantify network properties of the studied system at hand. Roughly speaking, null models consist of a collection of networks that match an observed network in some selected structural aspects, while being random in others. By using null models, observed networks can then be compared within a reasonable frame of reference.

Depending on the application and its requirements, it can be varyingly difficult to realize the null models in question. As such, an appropriately chosen null-model has to strike a balance between capturing structural properties and computational tractability. In this context, a commonly accepted method is to consider random simple graphs with matching degrees, as it captures non-trivial properties while still enabling potentially efficient realizations. It is therefore no coincidence that the generation of such graphs is a classical problem in theoretical computer science.

A common solution is to use Markov-Chain-Monte-Carlo (MCMC) methods, a notable example for this is the *Fixed-Degree-Sequence-Model*. It first generates a biased deterministic graph using the Havel-Hakimi algorithm which is then perturbed using an *Edge Switching* (*ES*) Markov chain process. The process repeatedly performs so-called *ES* switches where the incident nodes of two uniformly at random selected edges are exchanged – skipping any illegal switch that produces self-loops or multi-edges. It converges to a uniform simple graph with matching degrees if sufficiently many steps are performed. Its rate of convergence is quantified by the Markov chain's mixing time but practical upper bounds, despite intensive research, remain elusive. Though, in practice, usually a small multiple of the number of edges suffices.

Fixed-Degree-Sequence-Model:
☞ *Chapters 2 and 3*

A structurally similar and more recent process is *Curveball* (*CB*). In each step, the process proceeds by performing a *CB* trade that shuffles the neighborhoods of two nodes while fixing their degrees. As the neighborhoods may be considered in their entirety, each step of this process can inflict larger changes as compared to *ES* that may be reflected in the mixing times [46, 174, 180]. *Global Curveball* (*G-CB*) is a variant of *CB* that groups multiple *CB* trades into a single superstep. Each superstep consists of $n/2$ *CB* trades and targets each node exactly once allowing for benefits in the implementation and its analysis.

Besides MCMC based approaches that generate an approximately uniform graph, there exist exact uniform generators. They are often based on the *Configuration Model* and use switching mechanisms to exchange illegal structures to legal counterparts. Their use cases, however, are either limited by restrictions in the degrees or specified for certain types of graph classes.

This thesis therefore presents a variety of efficient algorithms for *ES*, *CB* and *G-CB* for various machine models. To accomodate parallel settings, we additionally explore further variants of *ES* that may enable better parallelism. Apart from straight-forward

Efficient MCMC Algorithms:
☞ *Chapters 2 to 4*

performance measures, we additionally incorporate the mixing time of the considered Markov chains in our experiments. Toward exact uniform generators, we consider and implement the recently developed Inc-Powerlaw algorithm [17] for the exact uniform generation of power-law graphs.

Exact Uniform Power-law Generator: ☞ *Chapter 5*

1.4 Connected Components in External Memory

The study of graphs majorly comprises many areas of both computer science and mathematics. Their use cases, however, certainly exceeds the scope of both these fields. In applications, graphs lend themselves as a convenient tool for the modeling of many complex structures and phenomena especially when entities and their relationships are the points of interest. Natural examples include but are not limited to social or communication networks. As such, vast amounts of research has been conducted for the study and development of algorithms for the analysis of graphs. This, in part, includes the examination and computation of certain structural properties. In this context, simple features that may be of interest are *connectivity* (is the underlying graph connected) or the actual computation of the *Connected Components* (*CC*).

For small scales, this task is trivially solvable by applying standard traversal algorithms like breadth-first or depth-first search. Both algorithms take linear time in the *unit-cost Ram* and compute further properties that go beyond the *CC* of the given graph. In the same vein, computing the *Minimum Spanning Forest* (*MSF*) may be regarded as a related problem, as any spanning tree constitutes a connected component of a graph. While there exists a randomized expected linear time algorithm for the *MSF* problem by Karger, Klein, and Tarjan [108], it is a long standing open problem whether the same complexity can be achieved deterministically. In the deterministic case, the problem can be optimally solved by an algorithm described by Pettie and Ramachandran [151] but the exact upper bound for this algorithm is not known. The closest upper bound that is known is $\mathcal{O}(m\alpha(m,n))$ [50] where α is the inverse Ackermann function.

Karger-Klein-Tarjan: ☞ *Section 6.3*

Large graphs that cannot be kept in main memory, however, do not lend themselves to the same algorithmic treatment. In the presence of a multi-level memory hierarchy, data transfers typically become the dominating factor in an algorithm's execution time. Hence, straight-forward applications of internal memory algorithms to this large-scale setting are prohibitively costly and unusable in practice. To alleviate this apparent gap, many algorithms for the *External Memory Model* have been designed. In this context, even simple graph problems become significantly harder to solve efficiently. This can already be seen, when considering the aforementioned problem of computing breadth-first search traversals. By utilizing structural properties of the iteratively computed breadth-first search levels, Munagala and Ranade [142] provide a $\mathcal{O}(n + \text{sort}(m))$ I/Os algorithm, that later, Mehlhorn and Meyer [129] improve upon by adding clustering as a preprocessing step to achieve an $\mathcal{O}\left(\sqrt{n(n+m)/B} + \text{sort}(n+m)\right)$ I/Os algorithm for undirected graphs. Buchsbaum et al. [42] propose a different algorithm that is applicable to directed graphs and relies on utilizing I/O-efficient buffered repository trees and I/O-efficient priority-queues (e.g. [13, 99]) instead. All of these algorithms, however, are very involved and therefore not suited to solve the *CC* problem in practice.

Instead of relying on graph traversals, algorithms that are designed for the *MSF* problem seem more promising. Deterministically, algorithms that are based on Borůvka's algorithm provide the best upper bounds. In its classical formulation, Borůvka's algorithm repeatedly performs so-called Borůvka steps in which the algorithm contracts nodes along their lightest incident edges. A Borůvka step, considered as a whole, therefore selects a subset of the edges and contracts the corresponding subgraphs in a subsequent step, reducing the overall number of nodes. When implemented as is, this leads to a simple $\mathcal{O}(\log(n/M)\,\mathrm{sort}(m))$ I/Os algorithm that can be further optimized to $\mathcal{O}(\log\log(nB/m)\,\mathrm{sort}(m))$ I/Os as shown by Arge et al. [15]. Despite the theoretically efficient I/O-complexity, implementations are, to the best of our knowledge, basically non-existent due to the high constants that would occur in an actual implementation. Therefore, a simpler algorithm by Sibeyn et al. [61] that uses an easier node contraction scheme based on *Time Forward Processing* [123] is favorable for use in practice, even though it has a worse I/O-complexity of $\mathcal{O}(\log(n/M)\,\mathrm{sort}(m))$ I/Os.

Borůvka:
☞ *Section 6.3*

Sibeyn:
☞ *Section 6.3*

Similar to Borůvka's algorithm, Karger-Klein-Tarjan's algorithm can also be considered for the external memory setting. Its translation leads to a randomized $\mathcal{O}(\mathrm{sort}(m))$ I/Os *MSF* algorithm which again may incur high constants in an actual implementation. However, solving the *CC* problem potentially enables significant optimizations to all aforementioned algorithms; this seems especially true for Karger-Klein-Tarjan. Therefore, in practice, it is not clear which algorithm to use in which circumstances and whether there is a clear-cut winner.

This thesis therefore presents an extensive experimental study for the *Connected Components* problem in the external memory setting. We provide efficient implementations for most available algorithms and execute them on a variety of graph classes. With these results, we shed light on how to efficiently perform *CC* computations for networks at scale.

Experimental Study:
☞ *Chapter 6*

1.5 Certifying Algorithms

The theory of algorithms strongly relies on mathematics and logical reasoning for the design and analysis of algorithms. Proving correctness and estimating the running time of algorithms for all possible inputs is undoubtedly an essential and necessary task. However, providing actual implementations of algorithms depends on a variety of additional factors. An often overlooked aspect in this regard is the raw complexity of the algorithms at hand, which can significantly increase when considering more advanced models of computation. While an algorithm may have been proven to be correct, its implementation may contain errors that can be very subtle and difficult to notice. Therefore, a lot of effort in software engineering is required to capture any errors that come with the implementation of algorithms that are used in supposedly reliable practical applications.

This motivates the study of *certifying algorithms* in an attempt to remove the aforementioned uncertainties. While there is no guarantee that any piece of software is bug-free, designing algorithms with software reliability in mind may already alleviate the issue significantly. To this end, a certifying algorithm is an algorithm that produces with each output a *certificate* that the returned output has not been compromised, i.e., evidence that can be used to authenticate the correctness of the answer. In order to validate the algorithm's output, the certificate is inspected using an *authentication algorithm* that takes the input, output, and certificate and returns whether the output is correct. As the use of the authentication algorithm is indispensable, it should be relatively simple to allow for a bug-free implementation.

To illustrate the effectiveness of certifying algorithms, we present a common toy example. Consider the problem of recognizing whether a given graph is bipartite, i.e., if there exists a bipartition of its nodes such that all edges have an endpoint in both partitions. A regular non-certifying algorithm will either return whether the graph is bipartite or not. At this point, however, the user has to simply trust the algorithm's execution as no more information is provided. In this context, a certifying algorithm will additionally return an appropriate certificate that proves its output to be correct. For our toy example, a certificate for bipartiteness is the actual bipartition of the nodes and contrary, in the case of non-bipartiteness an odd-length cycle. Verifying the algorithm's output now is straight-forward using the input and the provided certificate.

Naturally, certifying algorithms appear in a variety of different areas including graph recognition, combinatorial optimization, computational geometry, linear algebra and many more. Some of these subfields provide not only algorithms in theory but also lead to actual implementations; a prominent example for this is the *LEDA* project [130, 131] that provides implementations of many certifying algorithms. We note, however, that most of the present literature is concerned with algorithms for the *unit-cost* RAM and that to the best of our knowledge, external memory certifying algorithms have not been addressed in a structured way. Considering that practical applications that require the use of external memory generally have longer execution times, it is needless to say that certifying algorithms for larger processing scales are even more justified.

External Memory Certifying Graph Recognition: ☞ *Chapter 7*

This thesis aims to pave the first steps in this direction by considering external memory certifying graph recognition algorithms. We provide I/O-efficient algorithms and implementations that recognize whether a graph belongs to either of the considered graph classes and thus enable new means to classify networks at scale.

1.6 Articles Included in the Present Thesis

Comprehensive Summary: ☞ *Chapter 8*

This section contains a brief overview of the papers in Chapters 2 to 7 selected for the present thesis. The remaining chapters discuss algorithmic and engineering contributions of practical algorithms to uniformly sample simple graphs with prescribed degrees and further novel algorithms and considerations for the external memory setting with a focus on network analysis.

Edge Switching and Global Edge Switching

Chapters 2 and 3, based on our articles [90, 6], are concerned, in part, with Markov-Chain-Monte-Carlo (MCMC) processes for the randomization of graphs, namely *Edge Switching* (*ES*) and *Global Edge Switching* (*GES*). We provide algorithms for various machine models including sequential schemes over I/O-efficient algorithms to shared-memory parallel solutions. In Chapter 2 we present EM-LFR, a complex assembly of several I/O-efficient subroutines to generate large graphs following the *LFR* community detection benchmarks. Its most involving component is EM-ES, an I/O-efficient implementation of the *Edge Switching* process. In Chapter 3 we consider parallel implementations of *Edge Switching*, and in doing so develop *Global Edge Switching*, an alternative Markov chain process which exposes less dependencies in its execution allowing for improved parallelization.

[90] 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 2*

[6] D. Allendorf, U. Meyer, M. Penschuck, and H. Tran. Parallel global edge switching for the uniform sampling of simple graphs with prescribed degrees. In *IEEE Int. Parallel and Distributed Processing Symp. IPDPS*, pages 269–279. IEEE, 2022. doi:10.1109/IPDPS53621.2022.00034 . ☞ *Chapter 3*

Curveball and Global Curveball

Chapter 4, based on our article [48], presents a variety of algorithms realizing the *Curveball* (*CB*) and *Global Curveball* (*G-CB*) MCMC randomization process. Due to the nature of *CB* type processes, more data locality can be exposed leading to practical I/O-efficient algorithms that in some cases can outperform *ES* type processes. Additionally, since *G-CB* operates slightly differently in comparison to *CB*, further algorithmic optimizations are possible.

[48] 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 . ☞ *Chapter 4*

Exactly Uniform Sampling of Power-law Graphs

Chapter 5, based on our article [9], presents Inc-Powerlaw an exactly uniform sampler for power-law graphs with a degree exponent $\gamma \gtrapprox 2.88$. It highlights all necessary adjustments applied to the Pld algorithm [78] by adding incremental relaxation [17]. In particular, additional so-called *booster switchings* are introduced to ensure Inc-Powerlaw's expected linear runtime.

☞ *Chapter 5* [9] D. Allendorf, U. Meyer, M. Penschuck, H. Tran, and N. Wormald. Engineering uniform sampling of graphs with a prescribed power-law degree sequence. In C. A. Phillips and B. Speckmann, editors, *Proceedings of the Symp. on Algorithm Engineering and Experiments ALENEX*, pages 27–40. Society for Industrial and App. Math. SIAM, 2022. doi:10.1137/1.9781611977042.3 .

Connected Components in External Memory

Chapter 6, based on our article [41], provides an extensive experimental study of several external memory algorithms for the *Connected Components* problem. It provides tuned implementations of the considered algorithms and performs a comparison for a multitude of different graph classes.

☞ *Chapter 6* [41] G. S. Brodal, R. Fagerberg, D. Hammer, U. Meyer, M. Penschuck, and H. Tran. An experimental study of external memory algorithms for connected components. In D. Coudert and E. Natale, editors, *Int. Symp. on Experimental Algorithms SEA*, volume 190 of *LIPIcs*, pages 23:1–23:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.SEA.2021.23 .

Certifying Graph Recognition in External Memory

Chapter 7 is based on our article [134], and presents several I/O-efficient certifying recognition algorithms for a variety of graph classes. The algorithms incur $\mathcal{O}(\text{sort}(n+m))$ I/Os for a graph with n nodes and m edges for the recognition of split, threshold and trivially perfect graphs in the worst-case and with high probability for bipartite and bipartite chain graphs.

☞ *Chapter 7* [134] U. Meyer, H. Tran, and K. Tsakalidis. Certifying induced subgraphs in large graphs. In C. Lin, B. M. T. Lin, and G. Liotta, editors, *WALCOM: Algorithms and Computation - 17th Int. Conference and Workshops, WALCOM 2023, Hsinchu, Taiwan, March 22-24, 2023, Proceedings*, volume 13973 of *Lecture Notes in Computer Science*, pages 229–241. Springer, 2023. doi:10.1007/978-3-031-27051-2_20 .

Publications Not Included

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

[28] 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 .

[7] D. Allendorf, U. Meyer, M. Penschuck, and H. Tran. Parallel and I/O-efficient algorithms for non-linear preferential attachment. In G. Navarro and J. Shun, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2023, Florence, Italy, January 22-23, 2023*, pages 65–76. SIAM, 2023. doi:10.1137/1.9781611977561.ch6 .

[8] D. Allendorf, U. Meyer, M. Penschuck, and H. Tran. Parallel global edge switching for the uniform sampling of simple graphs with prescribed degrees. *J. Parallel Distributed Comput.*, 174:118–129, 2023. doi:10.1016/j.jpdc.2022.12.010 .

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

joint work with M. Hamann, U. Meyer, M. Penschuck, and 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 [90] extending [91]:

[91] 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 .

[90] 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

The journal version adds new algorithmic ideas related to the *Configuration Model* based on work of Manuel Penschuck and myself. I do *not* claim authorship for the preceding work in [91].

2.1 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 [69, 94, 71] 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 [19, 71]. The *LFR* benchmark [116, 114] has become a standard benchmark for such experimental studies, both for disjoint and for overlapping communities [63].

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 [43, 185]. Since such algorithms typically use hierarchical input representations, quality results of small benchmarks may not be generalizable to larger instances [63, 92]. 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 [20, 43]. Instead, computationally inexpensive random graph models such as *R-MAT* are used [152] 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 [92].

The distributed *CKB* benchmark [54] 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.

2.1.1 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., [137]), 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 2.5). Each edge switch can be seen as a transition in a Markov chain. This Markov chain is irreducible [62], symmetric and aperiodic [81] 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 [86]. 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 [137] suggest that $100m$ steps should be more than enough where m is the number of edges. Further results for random connected graphs [81] 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 [154].

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* [164]. In this context the question arises whether edge switches starting from the Configuration Model can be used to uniformly sample simple graphs at random.

2.1.2 Our Contribution

We introduce EM-LFR[1], 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 2.3. As illustrated in Figure 2.1, EM-LFR consists of several algorithmic building blocks which we discuss in Section 2.7. Here, the focus lies on *FDSM* consisting of (i) generating a deterministic graph from a prescribed degree sequence (cf. EM-HH, Section 2.4) and (ii) randomizing this graph using random edge switches (cf. EM-ES, Section 2.5). 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 2.6, 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

[1]The 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.

2.2 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.

2.2.1 Notation

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

Graphs and degree sequences. A graph $G = (V, E)$ has $n = |V|$ sequentially numbered nodes $V = \{v_1, \ldots, v_n\}$ 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 $\forall v_i \in V\colon\ \deg(v_i) = d_i$. 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. 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 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 2.A (Appendix) for a summary of commonly used definitions.

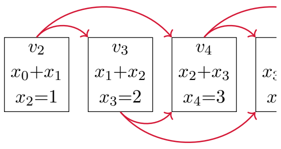
2.2.2 External Memory Model

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 [1] to

EM: external memory

Algorithm 1: Compute Fibonacci numbers using *Time Forward Processing*

```
PQ.push((key = 2, val = 0), (key = 2, val = 1))        // Send base cases $x_0$ & $x_1$ to $v_2$
for $i \leftarrow 2, \dots, n$ do
    sum ← 0
    while PQ.min.key == $i$ do
        sum ← sum + PQ.removeMin().val                 // Receive all messages for $x_i$
    print($x_i$ = sum)
    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 [1, 133].

IM: internal memory
M: *main memory size*
B: *block size*
scan
sort

2.2.3 TFP: Time Forward Processing

Time Forward Processing (*TFP*) is a generic technique to manage data dependencies in external memory algorithms [123]. 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 1).

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 [14, 160], *TFP* incurs $\mathcal{O}(\text{sort}(k))$ I/Os, where k is the number of messages sent.

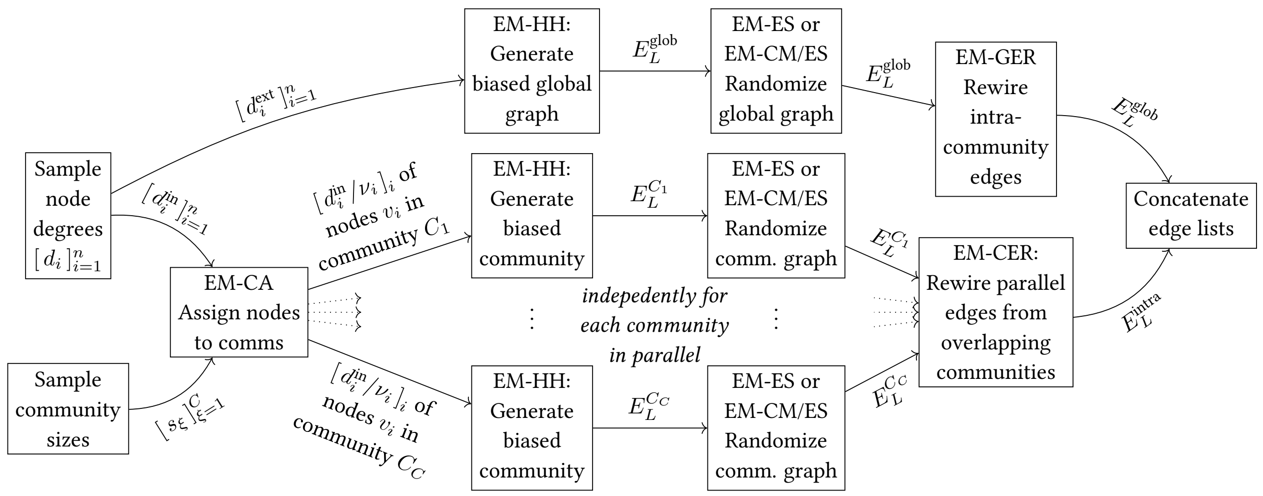

Sample node degrees $[d_i]_{i=1}^n$
Sample community sizes
$[s_\xi]_{\xi=1}^C$
$[d_i^{\text{ext}}]_{i=1}^n$
$[d_i^{\text{in}}]_{i=1}^n$
EM-CA Assign nodes to comms
$[d_i^{\text{in}}/\nu_i]_i$ of nodes v_i in community C_1
$[d_i^{\text{in}}/\nu_i]_i$ of nodes v_i in community C_C
EM-HH: Generate biased global graph
EM-HH: Generate biased community
E_L^{glob}
$E_L^{C_1}$
$E_L^{C_C}$
EM-ES or EM-CM/ES Randomize global graph
EM-ES or EM-CM/ES Randomize comm. graph
indepedently for each community in parallel
EM-GER Rewire intra-community edges
EM-CER: Rewire parallel edges from overlapping communities
E_L^{intra}
Concatenate edge lists

Figure 2.1: The EM-LFR pipeline: After randomly sampling the node degrees and community sizes, nodes are assigned into suited communities by EM-CA (Section 2.7). The global (inter-community) graph and each community graph is then generated independently by first materializing biased graphs using EM-HH (Section 2.4) followed by a randomization using EM-ES or EM-CM/ES (Sections 2.5 and 2.6). 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 2.8.1). Similarly, two overlapping communities can have identical edges which are rewired by EM-CER (Section 2.8.2).

2.3 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 2.1 for an overview).

The *LFR* benchmark [116] 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 [114] 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 2.A (Appendix) and are fully supported by EM-LFR.

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

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]

μ: mixing parameter

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.

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

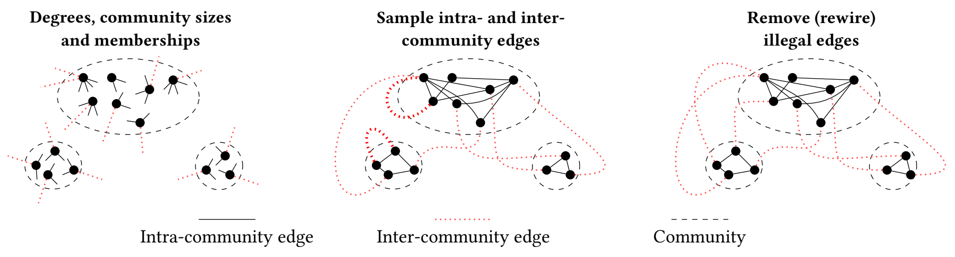

Degrees, community sizes and memberships
Sample intra- and inter-community edges
Remove (rewire) illegal edges
Intra-community edge
Inter-community edge
Community

Figure 2.2: **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 2.2, 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.

2.4 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 2.1) to first obtain a legal but biased graph following $\mathcal{D}$ and then randomizes $G_\mathcal{D}$ in a subsequent step.

graphical degree sequence

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 [95, 89]: 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 $\{\, \{v_1, v_{n-i}\} \mid 0 \le i < d_1 \,\}$. Then obtain an updated sequence $\mathcal{D}'$ by

[3]EM-LFR directly generates a monotonic degree sequence by first sampling a monotonic uniform sequence [27, 183] 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.

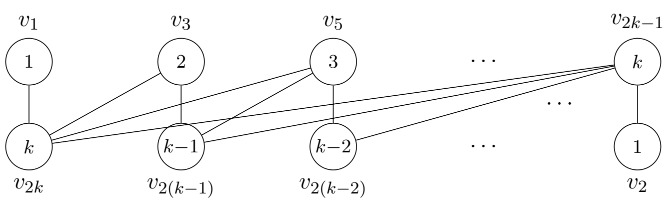


Figure 2.3: Graph with $\mathcal{D}_k = (1, 1, 2, 2, \ldots, k, k)$ maximizes EM-HH's memory consumption asymptotically as $D(\mathcal{D}_k) = k = \Theta(n)$. Node are labeled with their degrees.

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 [171, 181]. 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 2.6).

2.4.1 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. $\delta_j < \delta_{j+1} \;\; \forall 1 \leq j < |L|$

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

[4]This variant is due to [89]; in [95], the node of maximal degree is picked and connected.

[5]While 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.

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

Remark 2.1. 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 2.3, Lemma 2.2 gives a sub-linear bound on $D(\mathcal{D})$ if $\mathcal{D}$ is sampled from a powerlaw distribution. ◀

Lemma 2.2. Let $\mathcal{D}$ be a degree sequence of n nodes sampled from $\textsc{Pld}\,([1,n),\gamma)$. Then, there are $\mathcal{O}\big(n^{1/\gamma}\big)$ unique degrees $D(\mathcal{D}) = |\{d_i \mid 1 \le i \le n\}|$ whp.. ◀

Proof. Consider random variables $(X_1,\dots,X_n)$ sampled i.i.d. from $\textsc{Pld}\,([1,n),\gamma)$ as an unordered degree sequence. Fix an index $1{\le}j{\le}n$. 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:

$$\begin{aligned}
\mathbb{P}[X_j > n^{1/\gamma}] &= \sum_{i=n^{1/\gamma}+1}^{n-1} \mathbb{P}[X_j = i] = \frac{\sum_{i=n^{1/\gamma}+1}^{n-1} i^{-\gamma}}{\sum_{i=1}^{n-1} i^{-\gamma}} \overset{(i)}{=} \frac{\sum_{i=n^{1/\gamma}+1}^{n-1} i^{-\gamma}}{\zeta(\gamma) - \sum_{i=n}^{\infty} i^{-\gamma}} \\
&\overset{(ii)}{\le} \frac{\int_{n^{1/\gamma}}^{n-1} x^{-\gamma}\,\mathrm{d}x}{\zeta(\gamma) - \int_{n}^{\infty} x^{-\gamma}\,\mathrm{d}x} = \frac{\frac{1}{1-\gamma}\left[(n{-}1)^{1-\gamma} - n^{1/\gamma}/n\right]}{\zeta(\gamma) + \frac{1}{1-\gamma} n^{1-\gamma}} \\
&= \frac{n^{1/\gamma}/n - (n-1)^{1-\gamma}}{(\gamma-1)\zeta(\gamma) - n^{1-\gamma}} = \mathcal{O}\Big(n^{1/\gamma}/n\Big),
\end{aligned}$$

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

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 $p = \mathcal{O}\big(n^{1/\gamma}/n\big)$. Thus, the expected number of high degrees is $\mathbb{E}[\sum_{i=1}^{n} Y_i] = \sum_{i=1}^{n} \mathbb{P}[X_i{>}n^{1/\gamma}] = \mathcal{O}\big(n^{1/\gamma}\big)$. Chernoff's inequality gives an exponentially decreasing bound on the tail distribution of the sum which thus holds with high probability. □

Remark 2.3. Experiments in Section 2.10.2 indicate that the hidden constants in Lemma 2.2 are small for realistic γ. ◀

[6] A 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.

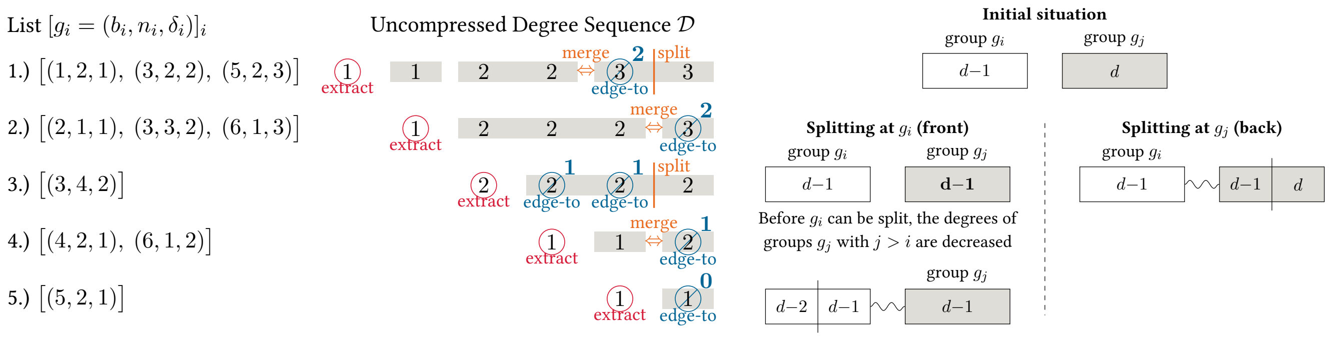

List $[g_i = (b_i, n_i, \delta_i)]_i$
Uncompressed Degree Sequence $\mathcal{D}$
1.) $[(1,2,1),\ (3,2,2),\ (5,2,3)]$
2.) $[(2,1,1),\ (3,3,2),\ (6,1,3)]$
3.) $[(3,4,2)]$
4.) $[(4,2,1),\ (6,1,2)]$
5.) $[(5,2,1)]$
extract
merge
split
edge-to
Initial situation
group g_i
group g_j
$d{-}1$
d
Splitting at g_i (front)
Splitting at g_j (back)
Before g_i can be split, the degrees of groups g_j with $j > i$ are decreased
$d{-}2$

Figure 2.4: **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 2.4. Graphs with $m = \mathcal{O}\big(M^{2\gamma}\big)$ edges and a powerlaw degree distribution are processed without I/O whp.. ◀

Proof. Due to Lemma 2.2 the number of unique degrees $D(\mathcal{D})$ is bounded by $\mathcal{O}\big(n^{1/\gamma}\big)$ 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 $m = \mathcal{O}\big(M^{2\gamma}\big)$ many edges with high probability. □

2.4.2 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 2.4 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 $g_j, \ldots, g_{|L|}$ 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 2.4 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 2.4 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 [89]. 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.

2.4.3 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 $(\mathtt{I1})$ enforces a timely activation.

Lemma 2.5. The optimized EM-HH needs $\mathcal{O}(\mathrm{scan}(D(\mathcal{D})))$ I/Os if L is an EM list. ◀

Proof. An external memory list requires $\mathcal{O}(\mathrm{scan}(k))$ I/Os to execute any sequence of k sequential read, insertion, and deletion requests to adjacent positions (i.e. if no seeking is necessary) [123]. 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 $\mathcal{O}(\mathrm{scan}(D(\mathcal{D})))$ 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

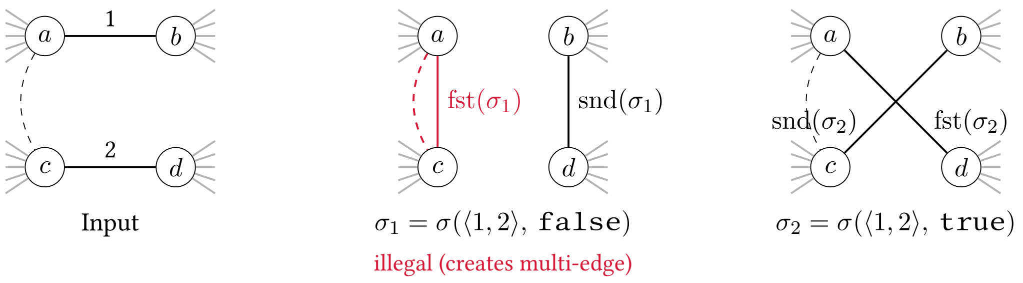

1
2
a
b
c
d
Input
fst(σ_1)
snd(σ_1)
$\sigma_1 = \sigma(\langle 1,2\rangle,\ \texttt{false})$
illegal (creates multi-edge)
snd(σ_2)
fst(σ_2)
$\sigma_2 = \sigma(\langle 1,2\rangle,\ \texttt{true})$

Figure 2.5: *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 2.4). As a result splitting can at most double L's size. □

2.5 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 2.1), and additionally functions as a building block to rewire illegal edges (cf Sections 2.6 and 2.8). As discussed in Section 2.10.6, 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 2.5, 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

$$\text{fst}(\sigma(\langle a,b\rangle,\ d)) := \begin{cases} \{\alpha_1,\beta_1\} & \text{if } d = \textsc{false} \\ \{\alpha_1,\beta_2\} & \text{if } d = \textsc{true} \end{cases}, \quad \text{and}$$

$$\text{snd}(\sigma(\langle a,b\rangle,\ d)) := \begin{cases} \{\alpha_2,\beta_2\} & \text{if } d = \textsc{false} \\ \{\alpha_2,\beta_1\} & \text{if } d = \textsc{true} \end{cases},$$

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 [181]). 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 2.6, 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.

2.5.1 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 2.5.6 contains the algorithmic modifications necessary to account for dependencies.

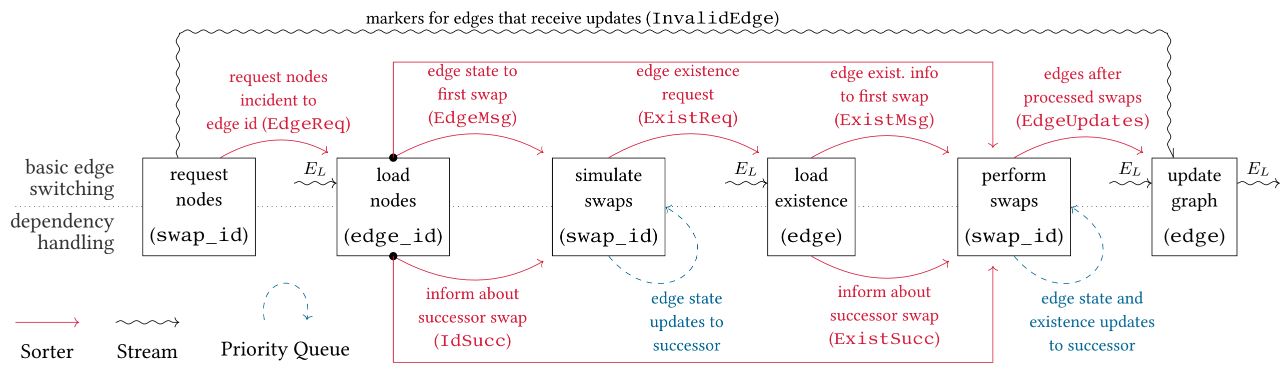

markers for edges that receive updates (InvalidEdge)
request nodes incident to edge id (EdgeReq)
edge state to first swap (EdgeMsg)
edge existence request (ExistReq)
edge exist. info to first swap (ExistMsg)
edges after processed swaps (EdgeUpdates)
basic edge switching
dependency handling
request nodes (swap_id)
E_L
load nodes (edge_id)
simulate swaps (swap_id)
E_L
load existence (edge)
perform swaps (swap_id)
E_L
update graph (edge)
E_L
inform about successor swap (IdSucc)
edge state updates to successor
inform about successor swap (ExistSucc)
edge state and existence updates to successor
Sorter
Stream
Priority Queue

Figure 2.6: 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 2.5.1 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 sorter[7] to later be processed in the order dictated by the data source of the receiving end. EM-ES uses the following phases:

2.5.2 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 EDGE_REQ$(a, s, 0)$ and 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 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 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 2.5.5. Since both phases produce only a constant amount of data per input

[7]The 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 [24].

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

2.5.3 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 $\textsc{edge\_msg}(s, 0, e_a)$ and $\textsc{edge\_msg}(s, 1, e_b)$ in EdgeMsg. This information suffices to compute the switched edges $\mathrm{fst}(\sigma)$ and $\mathrm{snd}(\sigma)$, but not to test for multi-edges.

It remains to check whether the switched edges already exist; we push the existence requests $\textsc{exist\_req}(\mathrm{fst}(\sigma), s)$ and $\textsc{exist\_req}(\mathrm{snd}(\sigma), s)$ 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 $\textsc{exist\_msg}(s, e)$ is pushed into the sorter ExistMsg. Both phases hence incur a total of $\mathcal{O}(\mathrm{sort}(r) + \mathrm{scan}(m))$ I/Os.

2.5.4 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 $\mathrm{fst}(\sigma)$ and $\mathrm{snd}(\sigma)$ 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 $\mathrm{fst}(\sigma)$ and $\mathrm{snd}(\sigma)$ into the sorter EdgeUpdates, otherwise we propagate the unaltered source edges e_a and e_b. This phase requires $\mathcal{O}(\mathrm{sort}(r))$ I/Os.

2.5.5 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 $\mathcal{O}(\mathrm{scan}(m))$ 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.

2.5.6 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.

2.5.7 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 $\mathrm{fst}(\sigma_1)$ and $\mathrm{snd}(\sigma_1)$ 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 EXIST_REQ$(\mathrm{fst}(\sigma_1), s_1)$ and EXIST_REQ$(\mathrm{snd}(\sigma_1), s_1)$ into sorter EdgeReq, 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 EXIST_REQ$(e_a, s_1,$ MAY_CHANGE$)$ and EXIST_REQ$(e_b, s_1,$ 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 `exist_succ`(s_1, e, s_2) into the sorter ExistSucc, 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.

2.5.8 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 EDGE_REQ(e_i, s_1, p_1) and 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 ID_SUCC(s_1, p_1, s_2, p_2) pushed into the sorter IdSucc.

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 EdgeUpdates sorter but rather forwarded to the successor in a *TFP* fashion. This way, every invalidated edge id receives exactly one update in EdgeUpdates and the merging remains correct.

2.5.9 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

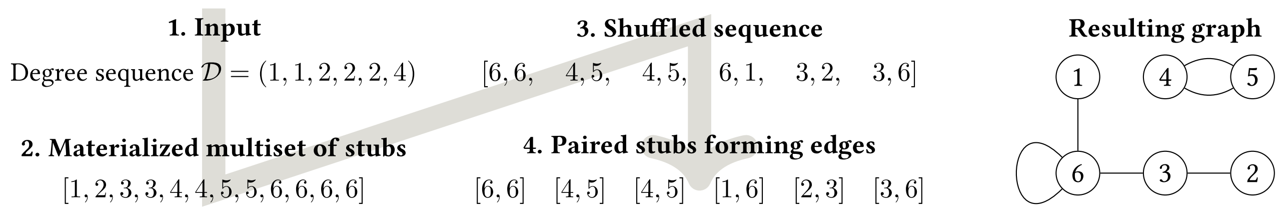

1. Input
Degree sequence $\mathcal{D} = (1, 1, 2, 2, 2, 4)$
2. Materialized multiset of stubs
$[1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 6, 6]$
3. Shuffled sequence
$[6, 6, \quad 4, 5, \quad 4, 5, \quad 6, 1, \quad 3, 2, \quad 3, 6]$
4. Paired stubs forming edges
$[6, 6] \quad [4, 5] \quad [4, 5] \quad [1, 6] \quad [2, 3] \quad [3, 6]$
Resulting graph
1
4
5
6
3
2

Figure 2.7: 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 [141]. 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 2.10.3 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..

2.6 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 2.6.2), this still results in a biased realization of the degree sequence requiring additional edge switching randomization steps.

2.6.1 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 [113]. 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-edge[8]. Finally, we scan over the shuffled sequence and match pairs of adjacent half-edges into edges.

As illustrated in Figure 2.7, the Configuration Model gives rise to self-loops and multi-edges which then need to be rewired, cf. section 2.6.2. Consequently, the rewiring process depends on the number of introduced illegal edges. In the following lemma, we bound their number from above.

Lemma 2.6. Let $\mathcal{D}$ be drawn from $\textsc{Pld}\left([a,b),2\right)$. The expected number of self-loops and multi-edges are bound by:

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

Proof. For an arbitrary degree sequence $\mathcal{D}$, [12] and [144] derive expectation values in terms of $\mathcal{D}$'s mean $\langle\mathcal{D}\rangle$ and its second moment $\langle\mathcal{D}^2\rangle$. For $n\to\infty$, the authors show:

$$\mathbb{E}[\text{\#self-loops}(\mathcal{D})] = \frac{\langle\mathcal{D}^2\rangle-\langle\mathcal{D}\rangle}{2(\langle\mathcal{D}\rangle-\frac{1}{n})} \longrightarrow \frac{\langle\mathcal{D}^2\rangle-\langle\mathcal{D}\rangle}{2\langle\mathcal{D}\rangle} \tag{2.1}$$

$$\mathbb{E}[\text{\#multi-edges}(\mathcal{D})] \leq \frac{1}{2}\left(\frac{(\langle\mathcal{D}^2\rangle-\langle\mathcal{D}\rangle)^2}{(\mathcal{D}-\frac{1}{n})(\mathcal{D}-\frac{3}{n})}\right) \longrightarrow \frac{1}{2}\left(\frac{\langle\mathcal{D}^2\rangle-\langle\mathcal{D}\rangle}{\langle\mathcal{D}\rangle}\right)^2 \tag{2.2}$$

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 $\textsc{Pld}\left([a,b),\gamma\right)$. 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 $\int_a^{b+1} x^q\,\mathrm{d}x \leq \sum_{i=a}^{b} i^q \leq \int_{a-1}^{b} x^q\,\mathrm{d}x$ 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 $(\int_a^{b+1} x^{-1}\,\mathrm{d}x)/C_\mathcal{D} \leq \langle\mathcal{D}\rangle$ to Section 2.6.1, directly yields the claim. $\square$

2.6.2 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

[8]If $M > \sqrt{mB}(1+o(1))+\mathcal{O}(B)$ this can be improved to $\mathcal{O}(\text{scan}(m))$ I/Os [159] 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. [186]). 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 $\texttt{exist\_msg}(s, e, \#(e))$ 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 $e_a, e_b, \operatorname{fst}(\sigma_1)$ and $\operatorname{snd}(\sigma_1)$ 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.

2.7 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 2.1) 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 $\chi\colon V \to [C]$ with:

(R1) Every community ξ is assigned s_ξ nodes as requested, with $s_\xi := \big|\{v \mid v \in V \wedge \chi(v){=}\xi\}\big|$.

(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.

2.7.1 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 2.7.2. Then the assignment graph can be sampled in the spirit of the Configuration Model (cf. Section 2.6). 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 2.7 (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 [132] based on lazy evaluation for such dynamic probability distributions enables a fully external algorithm with $\mathcal{O}\Big(n/B \cdot \log_{M/B}(C/B)\Big) = \mathcal{O}(\text{sort}(n))$ 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.

2.7.2 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 $\{1, \ldots, p_v\}$, 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 2.7. 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 $s_{,1}c = s_c$ 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 $u > 1$:

$$\begin{aligned}
&\mathbb{P}[\chi(u){=}c] \\
&\quad = s_{c,u}/\mathcal{W}_u \quad = \quad \mathbb{P}[\chi(u{-}1){=}c]\frac{s_{c,u-1} - 1}{\mathcal{W}_u} + \mathbb{P}[\chi(u{-}1){\neq}c]\frac{s_{c,u-1}}{\mathcal{W}_u} \\
&\quad = \frac{s_{c,u-1}}{\mathcal{W}_{u-1}}\frac{s_{c,u-1} - 1}{\mathcal{W}_u} + \left(1 - \frac{s_{c,u-1}}{\mathcal{W}_{u-1}}\right)\frac{s_{c,u-1}}{\mathcal{W}_u} \\
&\quad = \frac{s_{c,u-1} \cdot \mathcal{W}_{u-1} - s_{c,u-1}}{\mathcal{W}_{u-1} \cdot \mathcal{W}_u} \quad = \quad \frac{s_{c,u-1}(\mathcal{W}_{u-1} - 1)}{\mathcal{W}_{u-1} \cdot (\mathcal{W}_{u-1} - 1)} \\
&\quad = \frac{s_{c,u-1}}{\mathcal{W}_{u-1}} \overset{\text{Ind. Hyp.}}{=} \frac{s_{c,1}}{\mathcal{W}_1}
\end{aligned}$$

□

2.7.3 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^{in}/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 2.6.2.

2.8 EM-GER/EM-CER: Merging Intra- and Inter-Community Graphs

As illustrated in Figure 2.1, *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.

2.8.1 EM-GER: Global Edge Rewiring

The global graph is materialized without taking the community structure into account. As illustrated in Figure 2.2 (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 $\mathcal{O}(\mathrm{sort}(m))$ 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.

2.8.2 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 2.6.2 and 2.8. 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.

2.9 Implementation

We implemented the proposed algorithms in C++ based on the *STXXL* library [60], 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

[9]Consider 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 2.10).

2.10 Experimental Results

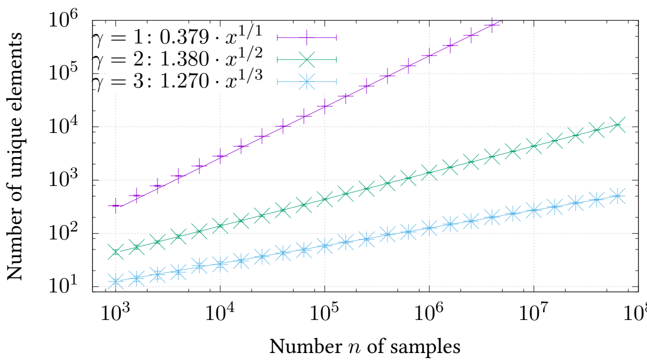
2.10.1 Notation and Setup

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: $\mu \in \{0.2, 0.4, 0.6\}$, $d_{\text{min}}{=}10\nu$, $d_{\text{max}}{=}n\nu/20$, $\gamma{=}2$, $s_{\text{min}}{=}20$, $s_{\text{max}}{=}n/10$, $\beta{=}1$, $O{=}n$.
- **const** — We keep the community sizes and the degrees constant and consider only non-overlapping communities. The parameters are chosen as: $d_{\text{min}}{=}50$, $d_{\text{max}}{=}10{,}000$, $\gamma{=}2$, $s_{\text{min}}{=}50$, $s_{\text{max}}{=}12{,}000$, $\beta{=}1$, $O{=}n$.

Real-world networks have been shown to have increasing average degrees as they become larger [119]. 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 [178]. The same study however found that strict powerlaw models are unable to accurately mimic Facebook's degree distribution. Further, the



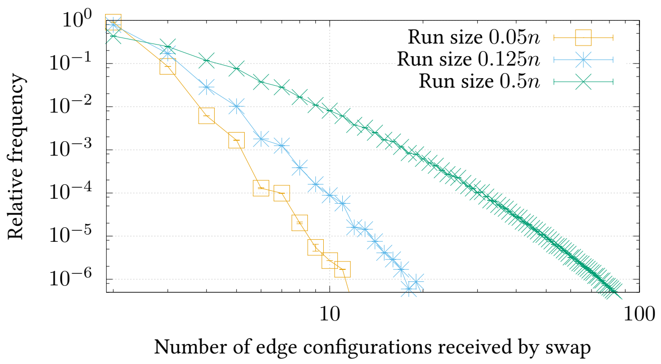


Figure 2.8: **Left:** Number of distinct elements in n samples (i.e. node degrees in a degree sequence) taken from $\textsc{Pld}([1, n), \gamma)$; cf. Section 2.10.2. **Right:** Overhead induced by tracing inter-swap dependencies. Fraction of swaps as function of the number of edge configurations they receive during the simulation phase (cf. Section 2.10.3).

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.

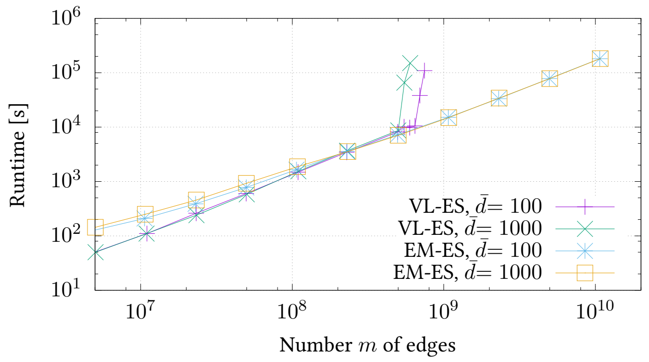
2.10.2 EM-HH's State Size

In Lemma 2.2, we bound EM-HH's internal memory consumption by showing that a sequence of n numbers randomly sampled from $\textsc{Pld}([1, n), \gamma)$ contains only $\mathcal{O}\left(n^{1/\gamma}\right)$ distinct values with high probability.

In order to support Lemma 2.2 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 2.8 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.

2.10.3 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 2.5.9, we argue that the number of



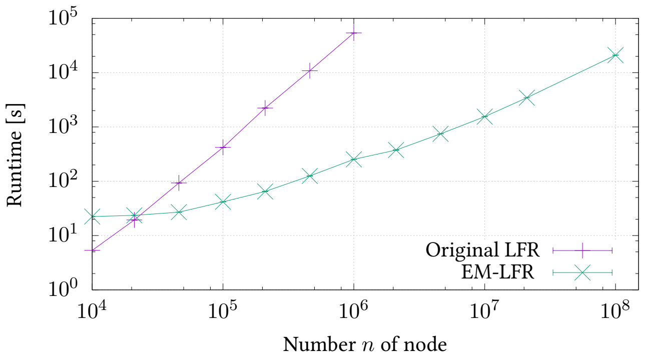


Figure 2.9: **Left:** Runtime on SysB of VL-ES and EM-ES on graph with m edges and avg. deg. $\bar{d}$ executing k=$10m$ swaps (cf. Section 2.10.6). **Right:** Runtime on SysA of the original *LFR* implementation and EM-LFR for μ=0.2 (cf. Section 2.10.9).

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 2.8, 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.

2.10.4 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.

2.10.5 Performance of EM-HH

EM-HH: ☞ *Section 2.4*

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.

2.10.6 Performance of EM-ES

Figure 2.9 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

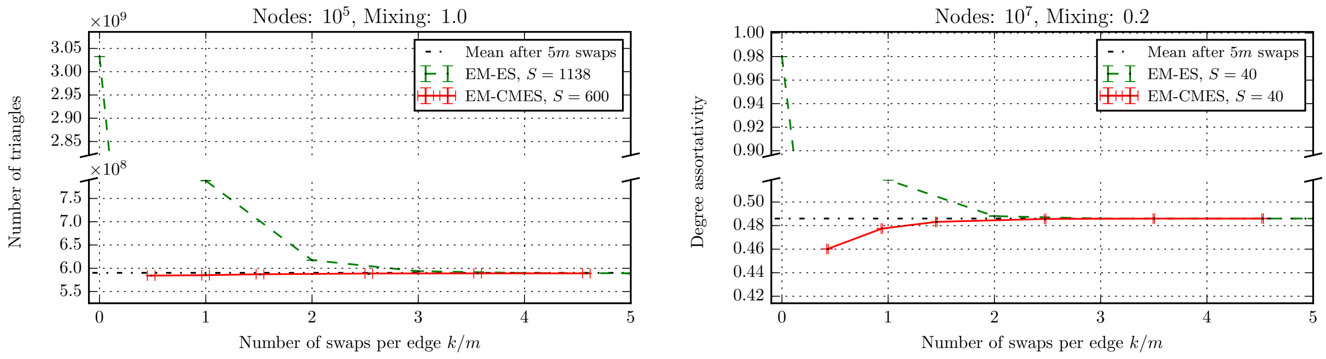


Figure 2.10: **Left:** Number of triangles on **const** with $n = 1 \cdot 10^5$ and $\mu = 1.0$. **Right:** Degree assortativity on **const** with $n = 1 \cdot 10^7$ and $\mu = 0.2$. In order to factor in the increased runtime of EM-CM/ES compared to EM-HH, plots of EM-CM/ES are shifted by the runtime 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 [181].[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.

EM-ES: ☞ *Section 2.5*

FDSM has applications beyond synthetic graphs, and is for instance used on real data to assess the statistical significance of observations [164]. In that spirit, we execute EM-ES on an undirected version of the crawled ClueWeb12 graph's core [176] 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 [30]. 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 [132] 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.

2.10.7 EM-CM/ES's Performance and Mixing Comparison with EM-ES

In Section 2.6, 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

[10]Here 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.

[11]We consider such vertices atypically simple as they have degree 1 and account for ≈84 % of nodes.

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 2.6*

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 2.1.1), we estimate the mixing times of both approaches as follows.

Starting from a common seed graph $G^{(0)}$, we generate an ensemble $\{G_1^{(k)}, \ldots, G_S^{(k)}\}$ 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 2.10.8).

As illustrated in Figure 2.10 and Section 2.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.

[12]Chosen to yield execution times similar to the $m/8$-setting of EM-ES on simple graphs.

[13]In 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.

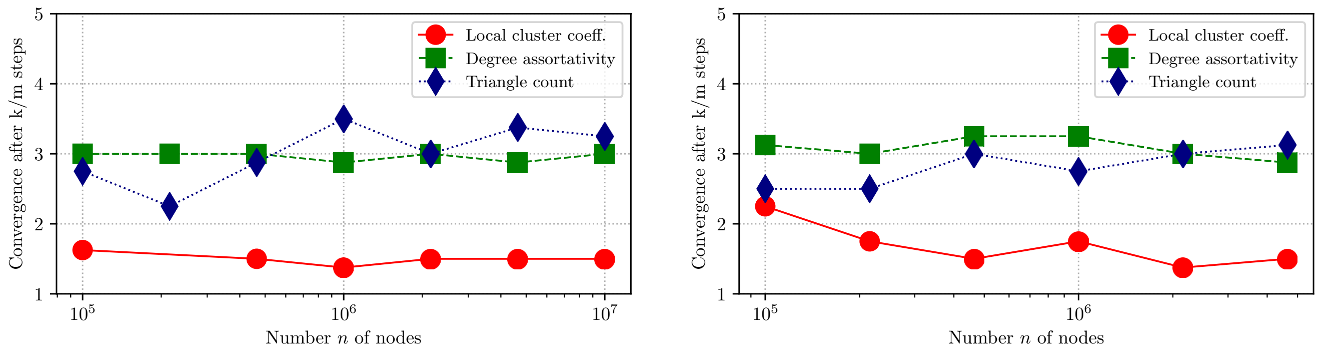


Figure 2.11: Number of swaps per edge after which ensembles of graphs with the following parameters converge: **const**, $1 \cdot 10^5 \leq n \leq 1 \cdot 10^7$ and $\mu = 0.4$ (left) and $\mu = 0.6$ (right). Due to computational costs, the ensemble size is reduced from $S > 100$ to $S > 10$ for large graphs.

2.10.8 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 2.11, 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.

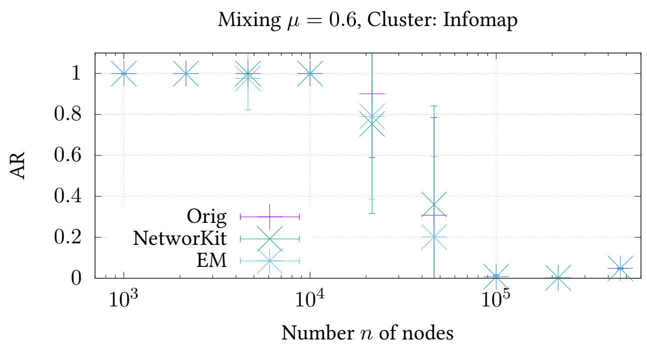
2.10.9 Performance of EM-LFR

Figure 2.9 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.

2.10.10 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* [171].

[14]Roughly 1.5 h are spend 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.



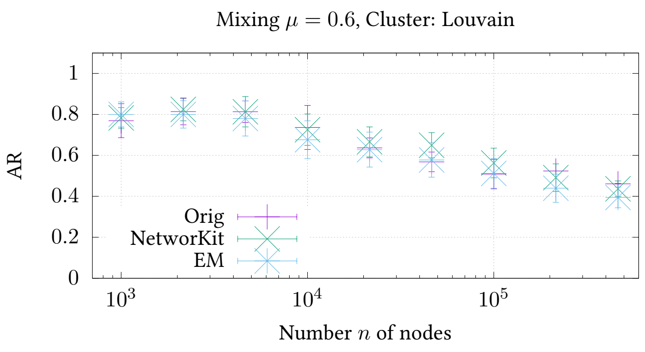


Figure 2.12: Adjusted rand of Infomap or Louvain and ground truth at μ=0.6 with disjoint clusters, $s_{\min}$=10, $s_{\max}$=$n/20$.

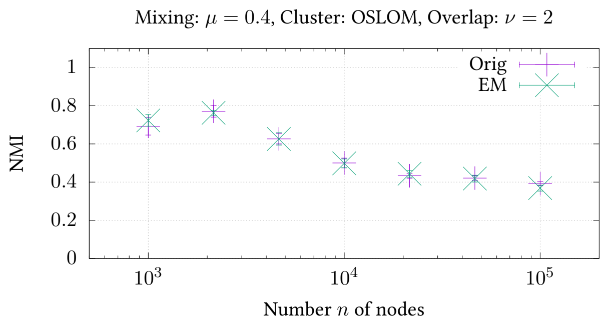
Infomap and Louvain

For disjoint clusters, we evaluate the results of the Infomap [158] and the Louvain [36] algorithm. The Louvain algorithm optimizes the famous modularity measure [145] while Infomap optimizes the map equation [158]. 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.

coverage

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 [70]. The map equation has a resolution limit, too, but in practice it is orders of magnitudes smaller [109]. The Louvain algorithm as well as Infomap were found to achieve high-quality results on *LFR* benchmark graphs while being fast [115]. In particular the Louvain method is also among the most frequently used community detection algorithms [71, 63].



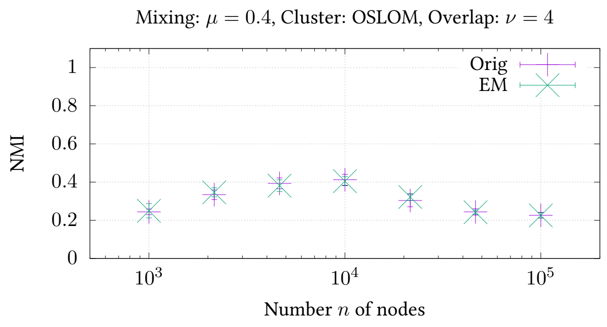


Figure 2.13: NMI of OSLOM and ground truth at $\mu = 0.4$ with 2/4 overlapping clusters per node.

OSLOM

For overlapping clusters, we evaluate the results of OSLOM [117]. 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.

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 [43, 71]. We compare the clusterings of the algorithms to *LFR*'s ground truth using the adjusted rand measure [98] for disjoint clusters and NMI [67] 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 [106]. We report these measures for graphs ranging from 10^3 to 10^6 nodes and present a selection of results in figures 2.12 to 2.14; all of them can be found in Section 2.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 2.12.

Similar to the results in [63], 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 degrees. Results on *LFR* graphs with one million nodes in [92] show that both Louvain

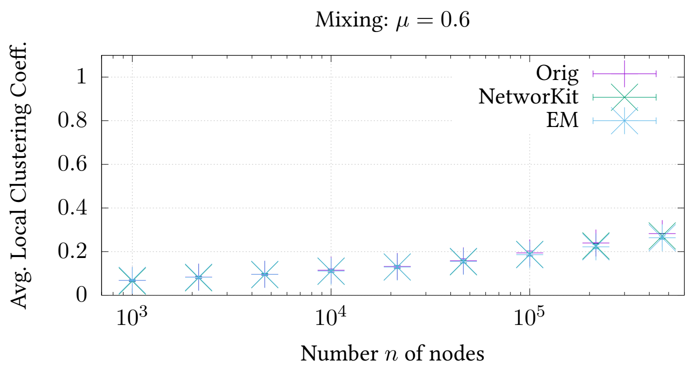


Figure 2.14: Average local clustering coefficient with mixing of $\mu = 0.6$ and disjoint clusters.

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 [145] 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 $\mathcal{Q} \to 1-\mu$ as the coverage corresponds to $1-\mu$ while the expected coverage approaches 0. For each configuration $n \in \{10^3, \ldots, 10^6\}$ 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.

2.11 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 [181] even for large instances still fitting in main memory and scales well beyond the limited main memory. Compared to the distributed approach by [30] 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 [92] 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.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and recommendations and Hannes Seiwert and Mark Ortmann for valuable discussions on EM-HH.

Appendix 2.A Summary of Definitions

Table 2.1: *Definitions used in this paper.*

Symbol	*Description*
$[k]$	$[k] := \{1, \dots, k\}$ for $k \in \mathbb{N}_+$ (Sec. 2.2)
$[u, v]$	Undirected edge with implication $u \leq v$ (Sec. 2.2)
$\langle X \rangle$	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. 2.2.2)
$d_{\text{min}}, d_{\text{max}}$	Min/max degree of nodes in *LFR* benchmark (Sec. 2.3)
d_v^{in}	$d_v^{\text{in}} = (1-\mu) \cdot d_v$, intra-community degree of node v (Sec. 2.3)
$\mathcal{D}$	$\mathcal{D} = (d_1, \dots, d_n)$ with $d_i \leq d_{i+1} \forall i$. Degree sequence of a graph (Sec. 2.4)
$D(\mathcal{D})$	$D(\mathcal{D}) = \left\lvert \{d_i : 1 \leq i \leq n\} \right\rvert$ where $\mathcal{D} = (d_1, \dots, d_n)$ (Sec. 2.4)
n	Number of vertices in a graph (Sec. 2.2)
m	Number of edges in a graph (Sec. 2.2)
μ	Mixing parameter in *LFR* benchmark, i.e. ratio of neighbors that shall be in other communities (Sec. 2.3)
M	Number of items fitting into internal memory (Sec. 2.2.2)
$\textsc{Pld}\left([a, b), \gamma\right)$	Powerlaw distribution with exponent $-\gamma$ on the interval $[a, b)$ (Sec. 2.2)
$s_{\text{min}}, s_{\text{max}}$	Min/max size of communities in *LFR* benchmark (Sec. 2.3)
$\text{scan}(n)$	$\text{scan}(n) = \Theta(n/B)$ I/Os, scan complexity (Sec. 2.2.2)
$\text{sort}(n)$	$\text{sort}(n) = \Theta((n/B) \cdot \log_{M/B}(n/B))$ I/Os, sort complexity (Sec. 2.2.2)

Table 2.2: *Parameters of overlapping LFR. The typical values are based on the suggestions by [114].*

Parameter	*Meaning*
n	Number of nodes to be produced
$\textsc{Pld}\left([d_{\text{min}}, d_{\text{max}}), \gamma\right)$	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
$\textsc{Pld}\left([s_{\text{min}}, s_{\text{max}}), \beta\right)$	Size distribution of communities, typically β=1
$0 < \mu < 1$	Mixing parameter: fraction of neighbors of every node u that shall not share a community with u

Appendix 2.B Comparing LFR Implementations

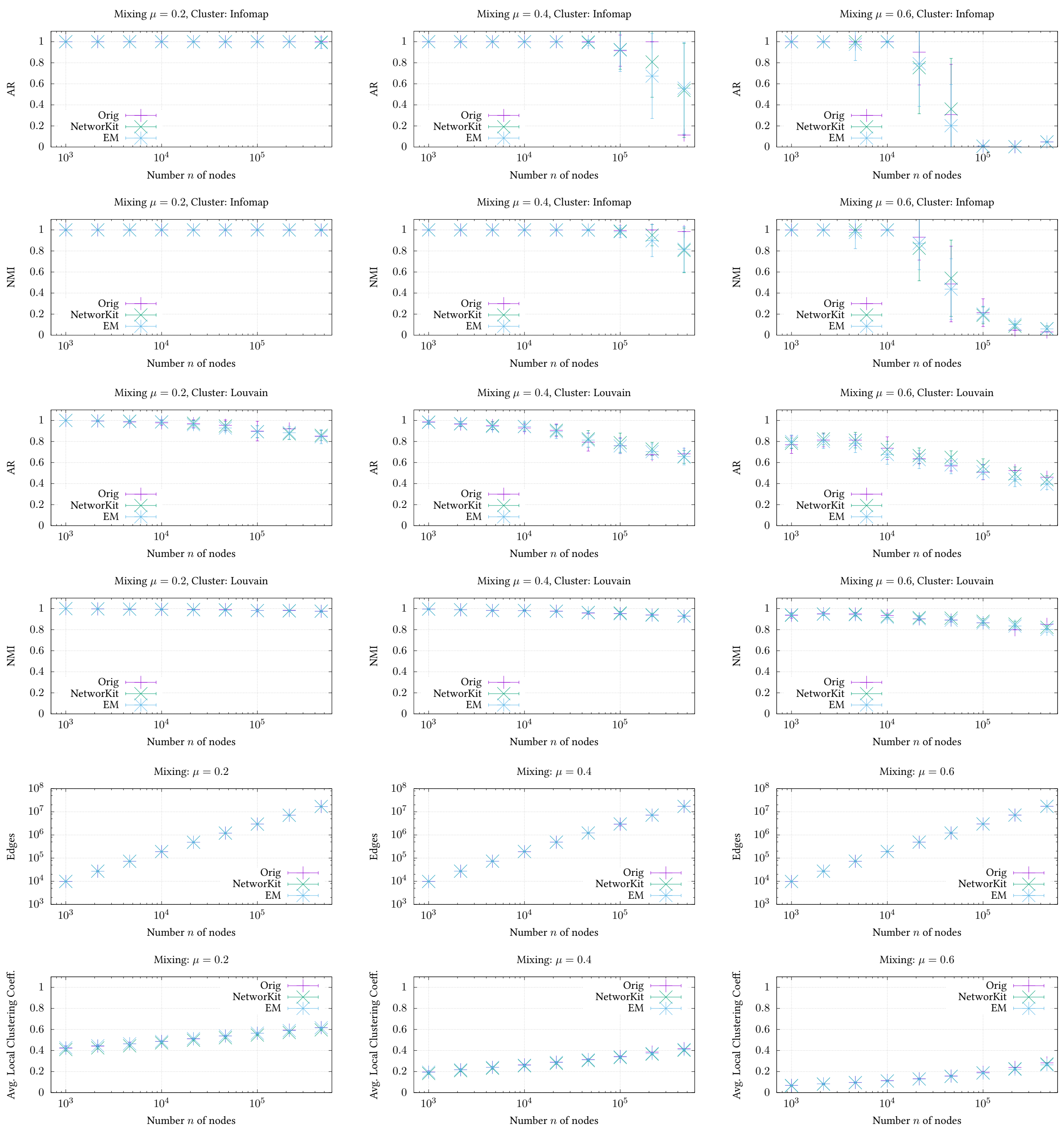
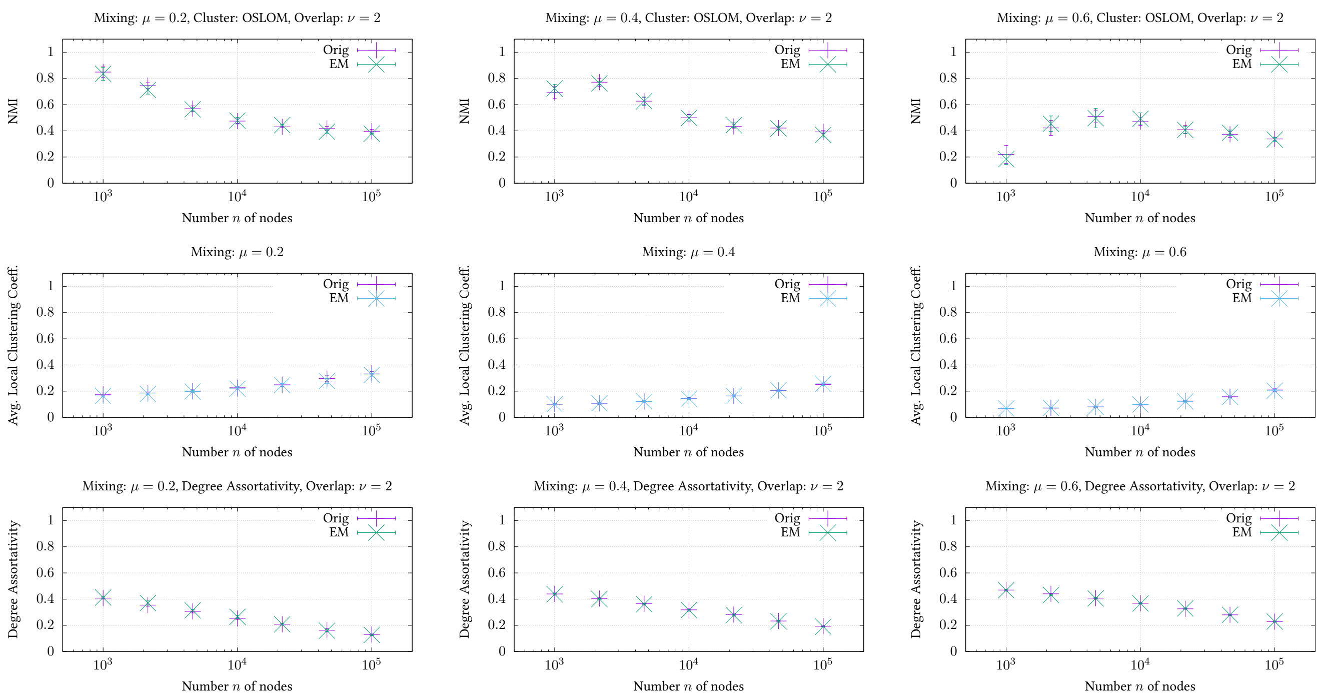
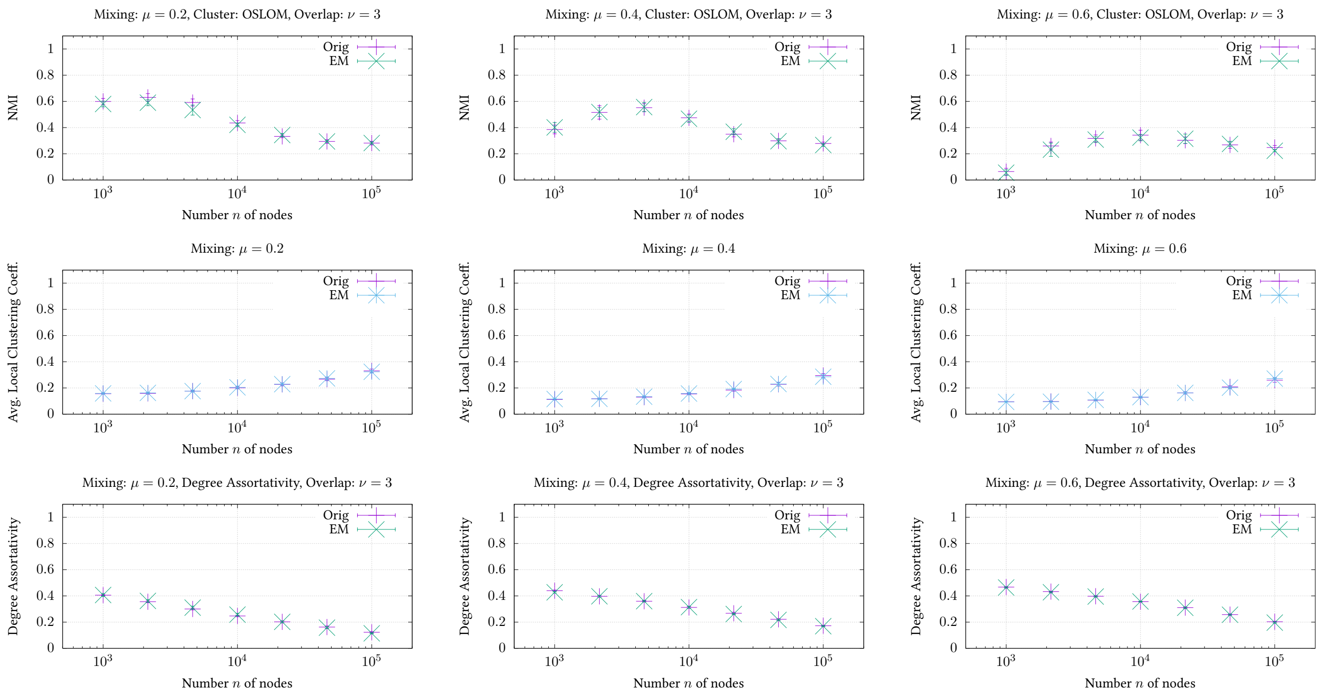


Figure 2.15: Comparison of the original *LFR* implementation, the *NetworKit* implementation and our EM solution for values of $10^3 \leq n \leq 10^6$, $\mu{\in}\{0.2, 0.4, 0.6\}$, $\gamma{=}2$, $\beta{=}1$ $d_{\text{min}}{=}10$, $d_{\text{max}}{=}n/20$, $s_{\text{min}}{=}10$, $s_{\text{max}}{=}n/20$. 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.


Mixing: $\mu = 0.2$, Cluster: OSLOM, Overlap: $\nu = 2$
Mixing: $\mu = 0.4$, Cluster: OSLOM, Overlap: $\nu = 2$
Mixing: $\mu = 0.6$, Cluster: OSLOM, Overlap: $\nu = 2$
Mixing: $\mu = 0.2$
Mixing: $\mu = 0.4$
Mixing: $\mu = 0.6$
Mixing: $\mu = 0.2$, Degree Assortativity, Overlap: $\nu = 2$
Mixing: $\mu = 0.4$, Degree Assortativity, Overlap: $\nu = 2$
Mixing: $\mu = 0.6$, Degree Assortativity, Overlap: $\nu = 2$
NMI
Avg. Local Clustering Coeff.
Degree Assortativity
Orig
EM
Number n of nodes
10^3
10^4
10^5


Mixing: $\mu = 0.2$, Cluster: OSLOM, Overlap: $\nu = 3$
Mixing: $\mu = 0.4$, Cluster: OSLOM, Overlap: $\nu = 3$
Mixing: $\mu = 0.6$, Cluster: OSLOM, Overlap: $\nu = 3$
Mixing: $\mu = 0.2$
Mixing: $\mu = 0.4$
Mixing: $\mu = 0.6$
Mixing: $\mu = 0.2$, Degree Assortativity, Overlap: $\nu = 3$
Mixing: $\mu = 0.4$, Degree Assortativity, Overlap: $\nu = 3$
Mixing: $\mu = 0.6$, Degree Assortativity, Overlap: $\nu = 3$
NMI
Avg. Local Clustering Coeff.
Degree Assortativity
Orig
EM
Number n of nodes
10^3
10^4
10^5

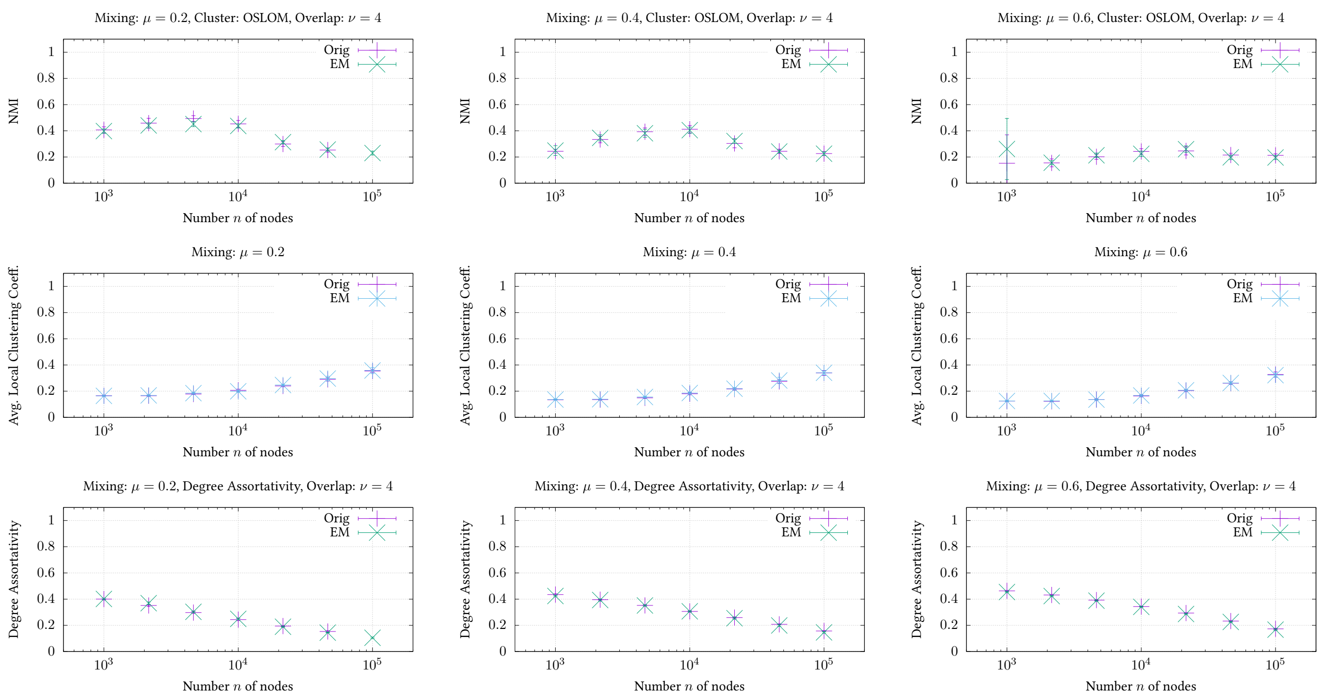


Figure 2.16: Comparison of the original *LFR* implementation and our EM solution for values values of $10^3 \leq n \leq 10^6$, $\mu \in \{0.2, 0.4, 0.6\}$, $\nu \in \{2, 3, 4\}$, $O = n$, $\gamma = 2$, $\beta = 1$ $d_{\text{min}} = 10$, $d_{\text{max}} = n/20$, $s_{\text{min}} = 10\nu$, $s_{\text{max}} = \nu \cdot n/20$. 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 2.C Comparing EM-ES and EM-CM/ES

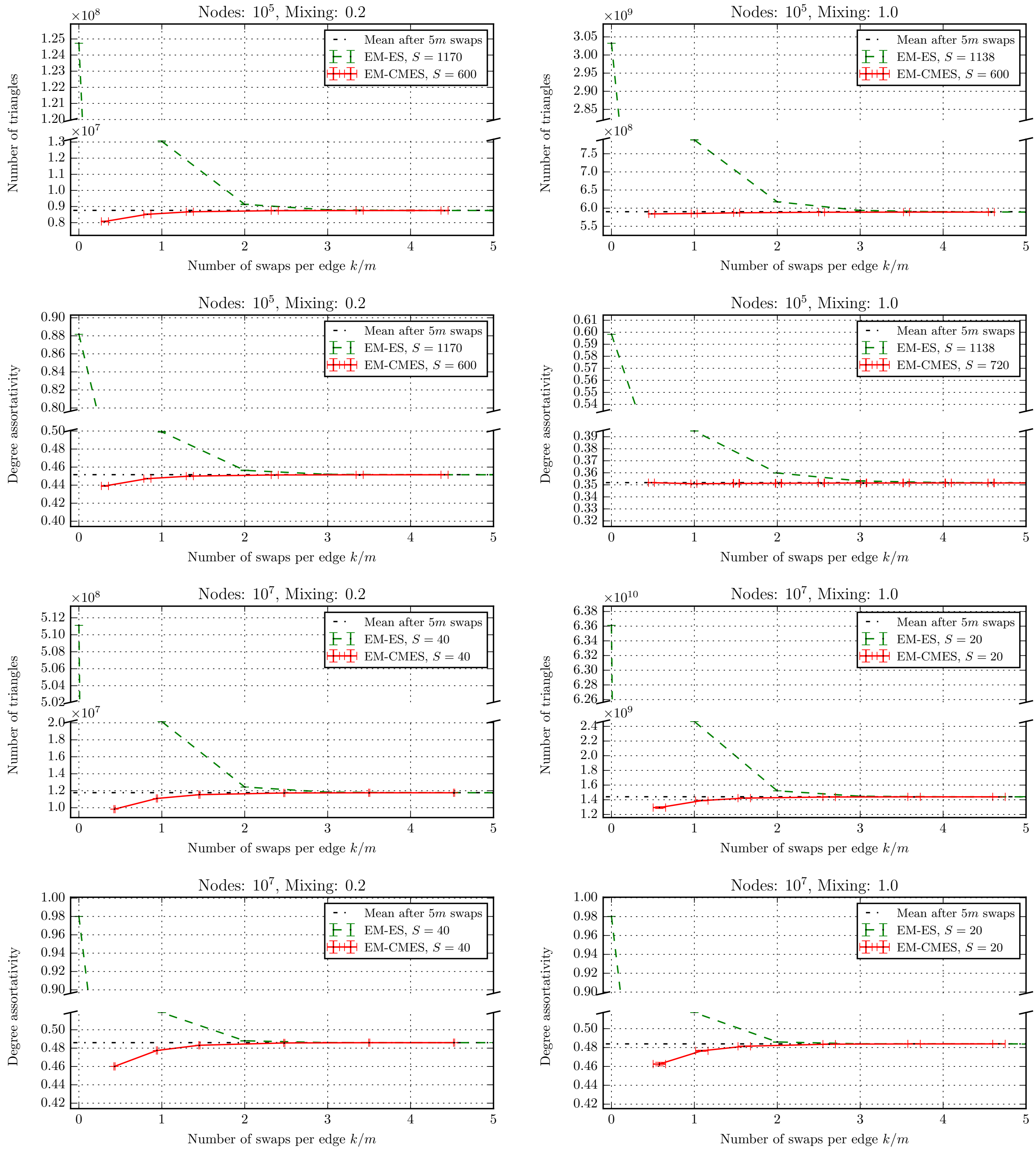


Figure 2.17: Triangle count and degree assortativity of a graph ensemble obtained by applying random swaps/the Configuration Model to a common seed graph. Refer to section 2.10.7 for experimental details.

3 Parallel Global Edge Switching for the Uniform Sampling of Simple Graphs with Prescribed Degrees

joint work with D. Allendorf, U. Meyer, and M. Penschuck

The uniform sampling of simple graphs matching a prescribed degree sequence is an important tool in network science, e.g., to construct graph generators or null-models. Here, the *Edge Switching Markov Chain* (*ES-MC*) is a common choice. Given an arbitrary simple graph with the required degree sequence, *ES-MC* carries out a large number of small changes involving at most four edges to eventually obtain a uniform sample. In practice, reasonably short runs efficiently yield approximate uniform samples.
We first engineer a simple sequential *ES-MC* implementation representing the graph in a hash-set. Despite its simplicity and to the best of our knowledge, our implementation significantly outperforms all openly available solutions.
Secondly, we propose the *Global Edge Switching Markov Chain* (*G-ES-MC*) and show that it, too, converges to a uniform distribution. We provide empirical evidence that *G-ES-MC* requires not more switches than *ES-MC* (and often fewer).
Thirdly, we engineer shared-memory parallel algorithms for *ES-MC* and *G-ES-MC*; we find that they benefit from the easier dependency structure of the *G-ES-MC*. In an empirical evaluation, we demonstrate the scalability of our implementations.

This chapter is based on the peer-reviewed conference article [6]:

[6] D. Allendorf, U. Meyer, M. Penschuck, and H. Tran. Parallel global edge switching for the uniform sampling of simple graphs with prescribed degrees. In *IEEE Int. Parallel and Distributed Processing Symp. IPDPS*, pages 269–279. IEEE, 2022. doi:10.1109/IPDPS53621.2022.00034 .

My contribution

Daniel Allendorf, Manuel Penschuck and I are main authors of this paper. Together, we contributed most of the algorithms and their implementations.

3.1 Introduction

In network science there are various measures, so-called centralities, to quantify the importance of nodes [37]. The degree centrality, for instance, suggests that a node's importance is proportional to its degree, i.e., the number of neighbors it has (see also [22]). This leads to the natural question whether graphs with matching degrees share structural properties. While this is not the case in general, a reoccurring task in practice is to quantify the statistical significance of some property observed in a network. Given an observed graph with degrees $\mathcal{D}$, a popular null-model is the uniform distribution over all simple graphs $\mathcal{G}(\mathcal{D})$ with matching degrees [55, 102, 163].

In this context, the *Edge Switching Markov Chain* (*ES-MC*) is a common choice to obtain an approximate uniform sample from $\mathcal{G}(\mathcal{D})$. In each so-called *edge switch*, two edges are selected uniformly at random and modified by exchanging their edges' endpoints. This process preserves the degrees of all nodes involved. We further keep the graph simple by rejecting all edge switches that introduce non-simple edges.

There exist different variants of *ES-MC* catering to various graph classes (e.g., Carstens [46] considers directed/undirected graphs, with/without loops, with/without multi-edges). Here, we focus on simple and undirected graphs. It is, however, straight-forward to adopt our findings to the other cases (some of which even lead to easier algorithms).

3.1.1 Related Work

Various methods to obtain a graph from a prescribed degree sequence have been studied [150].

Havel [95] and Hakimi [89] independently lay the foundation for a deterministic linear time generator. The algorithm, however, does not yield random graphs; while randomizations (e.g., [35, 32]) are available, they produce non-uniform samples.

The *Chung-Lu* Model [52] constructs graphs that match the prescribed degrees only in expectation. Under reasonable assumptions [150] it can be sampled in linear time [140].

Configuration Model: ☞ *Section 2.6.1*

The *Configuration Model* [26] outputs a random, but possibly non-simple, graph in linear time; adding rejection-sampling yields simple graphs in polynomial time if the maximum degree is $\mathcal{O}\left(\sqrt{\log n}\right)$ (cf. [25, 144, 38, 39]).

Efficient and exact uniform generators can be obtained by adding a repair step between the *Configuration Model* and the rejection-sampling to boost the acceptance probability. Such algorithms are available for several degree sequences classes including bounded regular graphs or power-law sequences with sufficiently large exponents [128, 75, 17].

(Global) Curveball: ☞ *Chapter 4*

Inc-Powerlaw: ☞ *Chapter 5*

Further a plethora of Markov Chain Monte-Carlo (MCMC) algorithms have been proposed and analyzed (e.g. [105, 56, 81, 85, 107, 122, 170, 174, 180, 181]). In comparison to the aforementioned exactly uniform generators, these algorithms allow for larger families of degree sequences, topological restrictions (e.g., connected graphs [81, 181]), or more general characterizations (e.g., joint degrees [170, 122]). Unfortunately, to the best of our knowledge, rigorous bounds on their mixing times either remain elusive or impractical (c.f. [56, 66, 86, 65, 11, 76]).

3.1.2 Our Contributions

We engineer a fast sequential *ES-MC* implementation as a baseline to quantify further speed-ups. This algorithm has expected linear running-time, and to the best of our knowledge, outperforms all freely available *ES-MC* implementations.

Next, we consider a simple parallelization of *ES-MC*, but show that it may deviate from the intended Markov Chain depending on the scheduler. We then consider a more involved parallelization that avoids deviations by taking into account the dependencies between edge switches. As we discuss, however, this parallelization is unlikely to scale well due to the complex nature of the dependencies.

For this reason, we propose the *Global Edge Switching Markov Chain* (*G-ES-MC*) and show that it too converges to a uniform distribution on $\mathcal{G}(\mathcal{D})$. This *ES-MC* variant is designed with parallel algorithms in mind and exhibits easier dependencies between edge switches. We then present an efficient parallel algorithm for *G-ES-MC*.

In an experimental section, we provide empirical evidence that *G-ES-MC* uses the same number of edge switches (or even fewer) as the standard *ES-MC*, and measure the efficiency and scalability of our algorithms.

3.2 Preliminaries

3.2.1 Notation and Definitions

Define the short-hands $[k..n] := \{k, \ldots, n\}$ and $[n] := [1..n]$. A graph $G = (V, E)$ has n nodes $V = \{v_1, \ldots, v_n\}$ and m undirected edges E. We assume that edges are indexed (e.g., by their position in an edge list) and denote the i-th edge of a graph as $E[i]$. Given an undirected edge $e = \{v_i, v_j\}$ we denote a directed representation as $\vec{e}$. We treat both as the same object and defining one implies the other; we default to the canonical orientation $\vec{e} = (v_{\min(i,j)}, v_{\max(i,j)})$ whenever the direction is ambiguous. An edge (v, v) is called a *loop* at v; an edge that appears more than once is called a *multi-edge*. A graph is *simple* if it contains neither multi-edges nor loops.

Given a graph $G = (V, E)$ and a node $v \in V$, define the ***degree*** $\deg(v) = |\{u : \{u, v\} \in E\}|$ as the number of edges incident to node v. Let $\mathcal{D} = (d_1, \ldots, d_n) \in \mathbb{N}^n$ be a *degree sequence* and denote $\mathcal{G}(\mathcal{D})$ as the set of simple graphs on n nodes with $\deg(v_i) = d_i$ for all $v_i \in V$. The degree sequence $\mathcal{D}$ is *graphical* if $\mathcal{G}(\mathcal{D})$ is non-empty.

$\deg(\cdot)$: *degree*

graphical degree sequence

A commonly considered class of graphs are power-law graphs where the degrees follow a power-law distribution. To this end, let $\textsc{Pld}([a..b], \gamma)$ refer to an integer Power-law 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 $\textsc{Pld}([a..b], \gamma)$ then $\mathbb{P}[X = k] \propto k^{-\gamma}$ (proportional to) if $a \leq k \leq b$ and $\mathbb{P}[X = k] = 0$ otherwise.

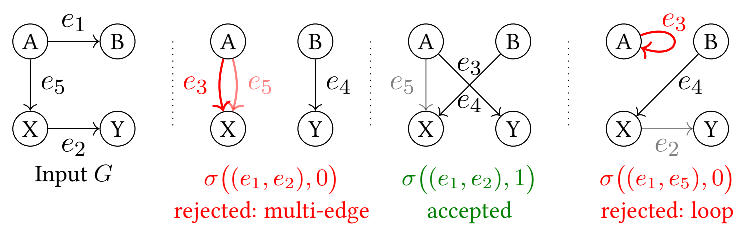


Figure 3.1: Edge Switch on an undirected graph $G = (V, E)$. To avoid ambiguity, we indicate for each $e \in E$ the orientation $\vec{e}$ used in Definition 3.1.

3.2.2 The Edge Switching Markov Chain (ES-MC)

Definition 3.1 (Edge Switch, ES-MC). Let $G = (V, E) \in \mathcal{G}(\mathcal{D})$ be a simple undirected graph. We represent an edge switch $\sigma = (i, j, g)$ by two indices $i, j \in [m]$ and a direction bit g. Then, we compute $G' \in \mathcal{G}(\mathcal{D})$ based on σ as follows:

1. Let $e_1 = E[i]$ and $e_2 = E[j]$.
2. Compute new edges $(\vec{e}_3, \vec{e}_4) = \tau(\vec{e}_1, \vec{e}_2, g)$ where
$$\tau\big((a, b), (x, y), g\big) = \begin{cases} \big((a, x), (b, y)\big) & \text{if g = 0} \\ \big((a, y), (b, x)\big) & \text{if g = 1} \end{cases}.$$
3. *Reject* if either of e_3 or e_4 is a loop or already exists in E; otherwise *accept* and set $E[i] \leftarrow e_3$ and $E[j] \leftarrow e_4$.

The *Edge Switching Markov Chain* (*ES-MC*) transitions from state G to state $G' \in \mathcal{G}(\mathcal{D})$ by sampling i, j, and g uniformly at random. ◀

Observation 3.2. Any edge switch $\sigma = (i, j, g)$ that translates G into G' can be reversed in a single step, i.e., there exists an inverse edge switch $\tilde{\sigma}$ that translates G' back into G. If σ is rejected, it does not alter the graph ($G = G'$). Thus the claim trivially holds with $\tilde{\sigma} = \sigma$. For an accepted edge switch σ, it is easy to verify that $\tilde{\sigma} = (i, j, 0)$ reverses the effects of σ. Further observe that the probability of choosing σ in state G equals the probability of choosing $\tilde{\sigma}$ in state G' [76]. ◀

3.3 Global Edge Switching Markov Chain (G-ES-MC)

Hamann et al. [90] consider the out-of-order execution of a batch consisting of ℓ edge switches. To this end, they classify the dependencies within the batch that arise if the switches were executed in-order. We adopt this characterization distinguishing between source- and target dependencies:

source dependency

target dependency

Definition 3.3 (Source/target dependencies). Two switches $\sigma_1 = (i_1, j_1, \cdot)$ and $\sigma_2 = (i_2, j_2, \cdot)$ are *source dependent* if they share at least one source index, i.e., $\{i_1, j_1\} \cap \{i_2, j_2\} \neq \emptyset$. Two switches $(e_1, f_1) \leftarrow \sigma(\cdot, \cdot, \cdot)$ and $(e_2, f_2) \leftarrow \sigma(\cdot, \cdot, \cdot)$ have a *target dependency* if they try to produce at least one common edge, i.e., $\{e_1, f_1\} \cap \{e_2, f_2\} \neq \emptyset$; this dependency is counted even if one or both edge switches are rejected. ◀

Source dependencies can be modelled by a balls-into-bins process where edges correspond to bins and each edge switch throws a linked pair of balls into two bins chosen uniformly at random. A source dependency arises whenever a ball falls into a non-empty bin. Czumaj and Lingas [59] analyse this process in a different context. Interpreted for *ES-MC*, they show that for $\ell = m$ the longest source dependency chain has an expected length of $\Theta(\log m / \log \log m)$. The distribution of target dependencies, on the other hand, depends on the graph's degree sequence as the probability that a random edge switch produces the edge $\{u, v\}$ is proportional to $\deg(u) \cdot \deg(v)$.

From an algorithmic point of view, source dependencies are more difficult to deal with if we want to process edge switches out-of-order (e.g., for parallel execution). Since each previous edge switch may or may not change the edge associated with the colliding edge index, the number of possible assignments may grow exponentially in the length of the dependency chain (if multiple chains cross). Consequently, we need to either serialize such edge switches or accommodate all possible assignments. In contrast, target dependencies only imply a binary predicate, namely whether a previous edge switch already introduced the target edge.

In a quest for more efficient parallel algorithms, we define a *global switch* — a batch of up to $\lfloor m/2 \rfloor$ edge switches without any source dependencies; conceptually, we place all edges into an urn and iteratively draw without replacement a pair of edges until the urn is empty; each pair implies an edge switch. It is folklore to encode such a process in a permutation that captures the order of edges drawn [150].

Similarly to techniques of [47, 48], our proof of Theorem 3.5 requires a small positive probability that any global switch collapses into a single switch. We implement this by independently rejecting each switch with probability P_L.

Definition 3.4 (Global Switch, G-ES-MC). Let $G = (V, E)$ be a simple undirected graph. A *global switch* is represented by $\Gamma = (\pi, \ell)$ where π is a permutation of $[m]$ and ℓ an integer with $0 \leq \ell \leq \lfloor m/2 \rfloor$. The global switch Γ consists of ℓ edge switches $\sigma_1, \ldots, \sigma_\ell$ that are executed in sequence, where $\sigma_k = (\pi(2k-1), \pi(2k), g_k)$ and $g_k = \mathbb{1}_{\pi(2k-1)<\pi(2k)}$, and where $\mathbb{1}$ denotes the indicator function.

The *Global Edge Switching Markov Chain* (*G-ES-MC*) transitions from graph G using a random global switch $\Gamma = (\pi, \ell)$. To this end, π is drawn uniformly from all permutations on $[m]$ and ℓ is drawn from a binomial distribution of $\lfloor m/2 \rfloor$ trails with success probability $0 < P_L < 1$. ◀

By selecting ℓ from a binomial distribution and executing the first ℓ edge switches of a random permutation, we simulate $\lfloor m/2 \rfloor$ edge switches that are each executed only with probability $1 - P_L$. Also note, that the direction bits $g_k = \mathbb{1}_{\pi(2k-1)<\pi(2k)}$ are independent and unbiased random bits because the permutation π is drawn uniformly at random.

Theorem 3.5. Let $G \in \mathcal{G}(\mathcal{D})$ be a simple undirected graph with degree sequence $\mathcal{D}$. The Global Edge Switching Markov Chain started at G converges to the uniform distribution on $\mathcal{G}(\mathcal{D})$. ◀

Proof. Any Markov Chain that is irreducible, aperiodic and symmetric converges to a uniform distribution [138, Th. 7.10]. We show that *G-ES-MC* has these three properties.

For **irreducibility** we observe that whenever there exists an edge switch from state A to B, there also exists a global switch from A to B (e.g., if $\ell = 1$). The state graph of *ES-MC* is therefore a subgraph of the state graph of *G-ES-MC*. In addition, both Markov Chains share the same set of states, i.e., the set of all simple graphs with the given degree sequence. Then, since the state graph of *ES-MC* is already strongly connected [153], so is the state graph of *G-ES-MC*, and thus both Markov Chains are irreducible.

For **aperiodicity** we note that a global switch $\Gamma = (\pi, \ell)$ may not alter the graph (e.g.,, if $\ell = 0$). Thus each state in the Markov Chain has a self-loop with strictly positive probability mass. This guarantees aperiodicity.

It remains to show the **symmetry** of transition probabilities. Let $\mathcal{S}_{AB}$ be the set of global switches Γ that transform graph A into graph B. Then the transition probability P_{AB} from A to B equals the probability of drawing a global switch from $\mathcal{S}_{AB}$,

$$P_{AB} = \sum_{\Gamma \in \mathcal{S}_{AB}} P(\Gamma),$$

where $P(\Gamma)$ is the probability of selecting Γ. A global switch $\Gamma = (\pi, \ell)$ is selected by drawing its permutation π and executing ℓ edge switches. In particular, we have

$$P(\Gamma) = \underbrace{\frac{1}{m!}}_{\text{uniform } \pi} \cdot \underbrace{\binom{\lfloor m/2 \rfloor}{\ell} (1 - P_L)^{\ell} P_L^{\lfloor m/2 \rfloor - \ell}}_{\text{binomially distributed } \ell}.$$

Observe that $P(\Gamma = (\pi, \ell))$ depends on ℓ, but neither on a specific choice of π nor the states A and B. Thus the symmetry $P_{AB} = P_{BA}$ follows by establishing a bijection μ_{AB} between any forward global switch $\Gamma{=}(\pi, \ell) \in \mathcal{S}_{AB}$ and an inverse global switch $\tilde{\Gamma}{=}(\tilde{\pi}, \ell) \in \mathcal{S}_{BA}$ with matching ℓ.

We construct the bijection μ_{AB} as follows. For a global switch $\Gamma = (\pi, \ell)$ that executes the edge switches $\sigma_1, \ldots, \sigma_\ell$ with $\sigma_k = (\pi(2k-1),\ \pi(2k),\ g_k)$ in sequence, define the inverse global switch $\tilde{\Gamma} = (\tilde{\pi}, \ell)$. The global switch $\tilde{\Gamma}$ executes the inverse edge switches in reverse order, i.e., $\tilde{\sigma}_{\ell-k+1}$ recovers the effect of the forward edge switch σ_k.

Recall that (i) the inverse of an accepted edge switch is given by a direction flag $g = 0$, and that (ii) the forward direction bit is defined as $g_k = \mathbb{1}_{\pi(2k-1)<\pi(2k)}$. Thus, if σ_k is legal and $g_k = 1$, we need to switch the order of the edge indices in the inverse switch $\tilde{\sigma}_{\ell-k+1}$. This implies $\tilde{\pi}$ on positions $[2\ell]$. In particular, we have for $k \in [\ell]$:

$$\Big(\tilde{\pi}(2[\ell{-}k{+}1] - 1), \quad \tilde{\pi}(2[\ell - k + 1])\Big) = \begin{cases} \big(\pi(2k-1), \quad \pi(2k)\big) & \sigma_k \text{ is illegal or } g_k = 0 \\ \big(\pi(2k), \quad \pi(2k-1)\big) & \sigma_k \text{ is legal and } g_k = 1 \end{cases}$$

For the unused entries $i \in [2\ell{+}1..m]$ choose $\tilde{\pi}(i) = \pi(i)$. □

3.4 Parallel Algorithms for ES-MC and G-ES-MC

3.4.1 Eager ES

EagerES is a simplistic parallelization of *ES-MC* to establish a performance baseline for parallel algorithms. Each PU (processing unit) performs switches independently while synchronizing implicitly only by preventing concurrent updates of individual edges. To ensure that no edge is erased or inserted twice, we store the edges in a concurrent hash-set using the following semantics: to remove an edge from the set, a ticket has to be acquired first; this can be done by locking an existing edge or by inserting-and-locking a new edge. These operations are implemented using a compare-and-exchange primitive. Concurrent updates to the same edge are sequenced by the hardware.

PU: processing unit

To perform an edge switch, a PU draws the indices of two edges $i, j \in [m]$ (independently of other PUs) and then attempts to rewire the edges $E[i]$ and $E[j]$ as in Definition 3.1. To this end, it first attempts to lock the two source edges and then to insert-and-lock the target edges. If any of them is already locked by a concurrent edge switch, the PU releases all locks it might hold to prevent dead locks, and then restarts using the same edge indices (observe that $E[i]$ and $E[j]$ might be different after the restart). If any of the target edges exists and is unlocked, the switch is rejected as ordinary. An implementation of EagerES is shown in Algorithm 2.

We call this algorithm *eager*, as it performs every edge switch that is legal after synchronization, but lacks a well-defined order. Instead, the order in which edge switches are performed is left up to the scheduler[1]. This however can lead to deviations from the intended Markov Chain. For example, consider two edge switches that are performed in parallel, and are legal, but attempt to insert the same edge, causing a target dependency. In that case, only the edge switch whose PU is able to lock both its target edges first is successful, and the other switch fails. Therefore, the transition probability between two adjacent states in the state graph, is not only determined by the probability of selecting the two source edges, but also by the probability of the switch having a target dependency and the probability of being scheduled first.

Even in the case of a fair scheduler that grants each edge switch the same probability of being scheduled first, the probability of the switch having a target dependency is not symmetric between adjacent states. For example, consider two adjacent states $A \neq B$, where A contains the edges $\{u, v\}, \{a, b\}$ and B contains the edges $\{u, a\}, \{v, b\}$. The probability that an edge switch between A and B encounters a target dependency is equal to the probability that another switch attempts to insert either one of the same edges. However, for the edge switch $A \to B$, the probability that another switch attempts to insert either $\{u, a\}$ or $\{v, b\}$ is proportional to the node degrees $\deg(u)\deg(a) + \deg(v)\deg(b)$, whereas for the edge switch $B \to A$, the probability is proportional to $\deg(u)\deg(v) + \deg(a)\deg(b)$.

In conclusion, while this algorithm is simple to describe and implement, it does not faithfully implement *ES-MC*. To do so, we need to take into account the dependencies

[1]On real hardware, a combination of the CPU, memory subsystem, and OS.

Algorithm 2: EagerES

```
Data: edge list $E$ of a simple graph, requested switches $r$
Let H $\leftarrow \emptyset$                                        // Initialize edge hash set H
for each edge $e \in E$ in parallel
    H.insert($e$)
for $i$ from $1$ to $r$ in parallel                      // Independently sample an edge switch $\sigma$
    Sample edge indices $i, j \sim [m]$ with $i \neq j$
    Sample direction bit $g \sim \{0, 1\}$
    Let $t_1, t_2$ be tickets for the source edges                    // Lock the source edges
    while not both $t_1, t_2$ acquired do
        Let $e_1 = E[i]$
        $t_1 \leftarrow$ H.lock($e_1$)
        if $t_1$ acquired then
            Let $e_2 = E[j]$
            $t_2 \leftarrow$ H.lock($e_2$)
            if not $t_2$ acquired then
                H.release($e_1, t_1$)
    Compute target edges $(\vec{e}_3, \vec{e}_4) = \tau(\vec{e}_1, \vec{e}_2, g)$
    Let $t_3, t_4$ be tickets for the target edges                    // Lock the target edges
    while not both $t_3, t_4$ acquired do
        $t_3 \leftarrow$ H.lock($e_3$)
        if $t_3$ acquired then
            $t_4 \leftarrow$ H.lock($e_4$)
            if not $t_4$ acquired then
                H.release($e_3, t_3$)
    if $e_3$ is not self-loop $\wedge$ $e_4$ is not self-loop then
        if H.insert($e_3, t_3$) succeeds then
            if H.insert($e_4, t_4$) succeeds then
                $E[i] \leftarrow e_3, E[j] \leftarrow e_4$                   // Success, rewire the edges
                $\forall i \in \{1, 2\}:$ H.erase($e_i, t_i$)
            else
                H.erase($e_3, t_3$)
        $\forall i \in \{3, 4\}:$ H.release($e_i, t_i$)
    $\forall i \in \{1, 2\}:$ H.release($e_i, t_i$)
```

between the switches and ensure that switches with dependencies are processed in a well-defined order. As discussed in Section 3.3, this is more difficult for *ES-MC* as it exhibits both source and target dependencies. In particular, if an edge switch has a source dependency on another, then the target edges of the second switch can only be determined once we know whether the first switch is legal. Thus, source dependencies make it difficult to even detect other dependencies deeper into the dependency chain.

We might consider an algorithm that performs edge switches in batches, and choose a small enough batch size so that each batch contains only target dependencies with high probability. This requires the batch size to be very small, e.g. at most $O(\sqrt{m})$ switches. However, this algorithm is unlikely to scale well, as we may only parallelize small batches of $O(\sqrt{m})$ switches each, but must perform $\Omega(\sqrt{m})$ batches sequentially to even perform m switches in total. *G-ES-MC* on the other hand, does not exhibit source dependencies by design, and allows us to parallelize global switches of up to $\lfloor m/2 \rfloor$ single edge switches each.

3.4.2 Steady Global-ES

We now present SteadyGlobalES, a parallel algorithm that faithfully implements *G-ES-MC*. It selects a random uniform global switch $\Gamma = (\pi, \ell)$, and processes[2] the ℓ single edge switches in Γ in order $\sigma_1, \dots, \sigma_\ell$, or any equivalent order (i.e., any reordering that preserves this transition).

To this end, we detect all edge switches in Γ that have dependencies on other switches, and ensure that if a switch σ_i depends on another switch σ_j, then σ_i is processed only after σ_j is. For our purposes, it is convenient to think in terms of the following two types of target dependencies (recall that a global switch does not have source dependencies):

- An *erase dependency* occurs if an edge switch σ_i creates an edge e that exists in the initial graph A but is erased by a switch σ_j with $j < i$. Hence, σ_i may be legal according to the order $\sigma_1, \dots, \sigma_\ell$ if σ_j is legal, and hence, we must ensure that σ_j is processed before σ_i.

erase dependency

- An *insert dependency* occurs if two edge switches σ_i and σ_k attempt to create the same edge e. In that case, we need to ensure that the switch with the smaller index $\min\{i, k\}$ is processed first to decide which of the switches is legal in the order $\sigma_1, \dots, \sigma_\ell$.

insert dependency

Observation 3.6. Given a graph $G = (V, E) \in \mathcal{G}(\mathcal{D})$ the global switch Γ attempts to remove only edges $e \in E$ and does so only once. As a direct consequence all erase dependencies for some edge $e \in E$ originate in the same switch σ_i. Also, for any number of insert dependencies for some edge e at most one switch σ_i can be successful. ◀

Algorithm 3 shows an implementation of SteadyGlobalES in pseudocode. Before performing a global switch Γ, we store the dependencies of the edge switches $\sigma_1, \dots, \sigma_\ell$

[2] We say that a single edge switch is *processed* when we decide with finality wether the switch is illegal, in which case the switch is rejected, or legal, in which case the edges are rewired.

Algorithm 3: SteadyGlobalES

```
Data: simple graph $G = (V, E)$ by edge list $E$
1  Shuffle edge list $E$ in parallel                      // Select random global ES $\Gamma = (\pi, \ell)$
2  Let $\ell \sim Binom(\lfloor m/2 \rfloor, 1 - P_L)$
3  Let T $\leftarrow \emptyset$                                   // Initialize dependency table T
4  for $i$ from $1$ to $\ell$ in parallel                   // Announce erased and inserted edges
5      Let $e_1 \leftarrow E[2i-1], e_2 \leftarrow E[2i]$
6      Compute new edges $(\vec{e}_3, \vec{e}_4) \leftarrow \tau(\vec{e}_1, \vec{e}_2, g_i)$
7      $\forall e_a \in \{e_1, e_2\}$: T.store($e_a$, $i$, erase, undecided)
8      $\forall e_b \in \{e_3, e_4\}$: T.store($e_b$, $i$, insert, undecided)
9  for for $j$ from $2\ell + 1$ to $m$ in parallel        // Announce edges not be erased by $\Gamma$
10     T.store($E[j]$, $\infty$, erase, illegal)
11 U $\leftarrow [1..\ell]$                              // Initialize array U with undecided switches
12 while U not empty do
13     D $\leftarrow \emptyset$                              // Initialize array D for delayed switches
14     for $i \in$ U in parallel
15         Let $e_1 \leftarrow E[2i-1], e_2 \leftarrow E[2i]$
16         Compute new edges $(\vec{e}_3, \vec{e}_4) \leftarrow \tau(\vec{e}_1, \vec{e}_2, g_i)$
17         Let $s_i \leftarrow$ legal                                 // Initialize status $s_i$
18         if $e_3$ or $e_4$ is self-loop then
19             $s_i \leftarrow$ illegal
20         for $e_b \in \{e_3, e_4\}$ do               // Lookup dependencies to decide $s_i$
21             $j, s_j \leftarrow$ T.lookup($e_b$, erase)
22             $k, s_k \leftarrow$ T.lookupFirst($e_b$, insert)
23             if $j > i \vee s_j =$ illegal then               Determine whether illegal
24                 $s_i \leftarrow$ illegal
25             if $k < i \wedge s_k =$ legal then
26                 $s_i \leftarrow$ illegal
27             if $s_i \neq$ illegal then              // Check whether has to be delayed
28                 if $j < i \wedge s_j =$ undecided then
29                     $s_i \leftarrow$ undecided
30                 if $k < i \wedge s_k =$ undecided then
31                     $s_i \leftarrow$ undecided
32         if $s_i =$ undecided then
33             D.append($i$)
34         else
35             if $s_i =$ legal then
36                 Set $E[2i-1] \leftarrow e_3, E[2i] \leftarrow e_4$      // Success, rewire the edges
37             $\forall e_a \in \{e_1, e_2\}$ : T.update($e_a$, $i$, erase, $s_i$)     // Update status flags
38             $\forall e_b \in \{e_3, e_4\}$ : T.update($e_b$, $i$, insert, $s_i$)
39     barrier: wait until all switches completed
40     U $\leftarrow$ D
```

in an additional data structure. Then, while attempting to process $\sigma_1, \dots, \sigma_\ell$ in parallel, this allows us to determine if a switch is ready to be processed or has unresolved dependencies. A global switch is then performed incrementally during multiple rounds. In each round, we only process switches that have no dependencies on unprocessed switches. This in turn resolves the dependencies of all switches that only depend on the processed switches, and as the dependencies cannot be circular, we eventually process all switches in this way. Then, finally, once all switches have been processed, the global switch has been performed.

In practice, we store the dependencies as tuples in a concurrent hash table with open addressing, and index them by the source or target edge. For each switch σ_i, we store four tuples, one for each source and each target edge, containing the edge e, the index i of the switch, the type of operation the switch attempts to perform on the edge $t_{e,i} \in \{\textsc{erase}, \textsc{insert}\}$ and a status flag $s_i \in \{\textsc{undecided}, \textsc{legal}, \textsc{illegal}\}$, that is initially set to $s_i = \textsc{undecided}$. Usually, in a global switch, most edges are erased, and so we use the same data structure to lookup the existence of edges. To this end, we also store a tuple for each edge that is not erased by the global switch, with the information that the edge is not erased, causing a switch that attempts to insert this edge to be decided as illegal.

Now, when attempting to process a switch σ_i, we lookup all tuples where the edge is one of its target edges. By observation 3.6, for each target edge e, there is only one tuple stored by a switch σ_j where $t_{e,j} = \textsc{erase}$, i.e. that erases the edge. Then, if $j < i$, we know that i has an erase dependency on j. Similarly, for each target edge e, there is only one tuple stored by a switch σ_k, that is legal, and inserts the target edge. Specifically, at any point, the only tuple that needs to be considered is the tuple with the smallest index k where $t_{e,k} = \textsc{insert}$ and $s_k \neq \textsc{illegal}$. Then, if $k < i$, we know that i has an insert dependency on k.

We then use this information to decide if the switch is illegal, must be delayed, or is legal. If $j > i$ or $s_j = \textsc{illegal}$, then a target edge of the switch σ_i is only erased by a later switch σ_j, or not erased at all, and thus the switch is illegal. Similarly, if $k < i$ and $s_k = \textsc{legal}$, then a target edge of σ_i is already inserted by the earlier switch σ_k and thus the switch is illegal. Otherwise, if an erase or insert dependency exists, but the status of the other switch is $\textsc{undecided}$, the switch must be delayed. Finally, if none of the above holds true, then the switch is legal.

If the switch is legal, the edges are rewired and the status of the tuples is set to $\textsc{legal}$. Otherwise, if the switch is illegal, the status of the tuples is set to $\textsc{illegal}$. Finally, if the switch has unresolved dependencies, we delay it until the next round.

3.5 Implementation

In this section, we describe the implementation of our sequential and parallel algorithms. All algorithms (including previously existing ones) are implemented in C++.

3.5.1 Graph and dependency representation

Most *ES-MC* implementations use an adjacency list to store the graph and manipulate it with each switching [31, 171, 88, 114, 182][3]. This design choice often leads to an easy integration with other algorithms. However, *ES-MC* requires a graph representation that efficiently supports edge insertion, deletion, and existence queries. Unfortunately, an adjacency list cannot support updates and search both in constant time (cf. the hybrid data structure of [182]).

In contrast, hash-sets support all required operations in expected constant time. From a practical point of view, we require a hash-set implementation that can handle a roughly balanced mix of insertions, deletions, and search queries. After preliminary experiments on various graphs, machines, and hash-set implementations[4], we find that in most cases RobinMap with a maximum load-factor of $1/2$ is the fastest sequential solution. Observe that for performance reasons, all our implementations use hash-tables where the number of buckets is a power-of-two; hence the actual load-factor can be lower. Our hash function uses the 64bit variant of the `crc32` instruction available on `x64` processes with SSE 4.2 [100].

Our parallel algorithms require concurrent hash-tables with stable iterators (i.e., once an element is placed into a bucket, it is not moved until it gets erased). It is folklore that such a data structure can be efficiently implemented with open addressing and lock-free compare-and-swap instructions (cf. [124]).

We implement the locking of edges as follows: each edge is kept in an 64bit-wide bucket, where 56 bits are used to store the edge and 8 bit are reserved for locking. To acquire a lock, a PU tries to compare-and-swap its thread id into the lock bits and succeeds only if the bucket previously kept the edge in an unlocked state. This implementations allows us to process graphs with up-to $n \leq 2^{28}$ nodes on $P < 256$ threads. Observe that these restrictions can be lifted quite easily, as virtually all relevant processors support 128bit compare-and-swap instructions with only moderate performance penalities.

3.5.2 Sampling edges

Pseudo-random bits are generated using the MT19937-64 variant of the Mersenne Twister [126] implemented by `libstdc++` and translated into unbiased random integers using [118]. Random permutations are sampled in parallel with an optimized implementation inspired by [159, 118].

To sample edges uniformly at random we consider two options: Firstly, we maintain an auxiliary array of edges. In order to sample an edge uniformly at random, we read from a random index — this closely resembles the way we introduced edge switching in

[3][31] use a *reduced* adjacency matrix that only stores one directed edge. [114] use a different MC with the same switchings. [182] interleave *ES-MC* with connectivity checks after each edge switching. They use an adjacency list where high-degree nodes store their neighborhoods in individual hash tables.

[4]We considered the following hash-sets: https://gcc.gnu.org, https://github.com/{Tessil/robin-map, Tessil/hopscotch-map,sparsehash/sparsehash}.

the previous chapters. Secondly, the use of open-addressing hash-tables allows us to directly sample from the hash-set by repeatedly drawing random buckets until we hit the first non-empty one.

While the second option avoids additional memory, it leads to a time trade-off: while all queries discussed in Section 3.5.1 benefit from a low load-factor L, the sampling time is geometrically distributed with a success probability of L. It, thus, favors a high load-factor. In preliminary experiments, we found that decreasing the load factor to allow faster queries and sampling using an additional array yields up-to 30 % faster overall performance compared to balancing the load factor for both queries and sampling.

3.5.3 Prefetching

The rewiring of random edges inherently leads to unstructured accesses to main memory, especially if the graph is represented in a hash set. While [90] propose an I/O-efficient edge switching implementation for graphs exceeding main memory, their solution requires to repeatedly sort the edge list (and other data structures). In the context of a parallel algorithm, this sorting step alone is more expensive than a global switch using unstructured accesses.

We therefore accept the random I/Os and accelerate them using prefetching instructions. To this end, we split all insertion, deletion, and search queries to our hash-sets into two: in a first step, we hash the key and identify the bucket in which the item is placed if there is no collision. We then prefetch this bucket as well as its direct successor, and return precomputed values that are required when we carry out the actual operation in a second step. Since we use linear probing hash-sets with a low load factor and a prefetch in advance, we effectively eliminate almost all cache misses if there is sufficient time between both steps. To increase this time window, we use a pipeline of four edge switches in different progress stages.

3.6 Experiments

In the following, we empirically investigate the mixing times of the Markov Chains and the runtime performances of their derived algorithms. The performance benchmarks are built with GNU g++-9.3 and executed on a machine equipped with an AMD EPYC 7702P 64-core processor running Ubuntu 20.04.

Our experiments are performed on the following datasets:

- (SynGnp) — We generate $G(n, p)$ graphs [79] (each edge exists independently with probability p) for varying node counts n and p.

- (SynPld) — For varying node counts n and degree exponents γ, we generate power-law degree sequences according to the degree distribution $\text{Pld}([1..\Delta], \gamma)$ where the maximum degree is set to $\Delta = n^{1/(\gamma-1)}$ matching the analytic bound [78]. Thereafter, the generated sequences are materialized by the Havel-Hakimi algorithm [95, 89]. Both steps are performed using *NetworkKit* [171].

γ: *degree exponent*

Δ: *maximum degree*

- (NetRep) — We consider a subset of graphs[5] of the network repository [157] with the following modifications: all directed edges (u, v) are replaced by the undirected edges $\{u, v\}$, and self-loops and multi-edges are removed.

3.6.1 Empirical Mixing Time of ES-MC and G-ES-MC

Here, we compare the mixing times of *ES-MC* and the novel *G-ES-MC*. While we argue in Section 3.3 that the lack of source dependencies of *G-ES-MC* improves parallelizability over *ES-MC*, it is a priori unclear whether this restriction affects the randomization quality of the Markov Chain.

mixing time

For simple graphs, the mixing time of *ES-MC* has been studied for many families of degree sequences (c.f. [56, 66, 86, 65, 11, 76]). In practice, the mixing time is approximated by empirical proxies or estimated by data driven methods [169, 155]. The former measures the convergence to the stationary distribution by convergence of its proxy or by some aggregated value. In doing so, the Markov Chain is reflected by a projection which may converge faster [29]. The result depends on the proxy and might be insufficient for other more sensitive proxies. Additionally, it has been observed that common measures, e.g., assortativity coefficients, clustering coefficients, diameter, maximum eigenvalue and triangle count, are less sensitive than data-driven methods [90, 155]. Thus, we consider the *autocorrelation analysis*, an approximate non-parametric method [155].

autocorrelation analysis

The autocorrelation analysis proceeds in two steps. First, execute the Markov Chain for a large number of steps K, and for each possible edge e track in a binary time-series $\{Z_t\}$ whether edge e exists at time t. By its own, $\{Z_t\}$ and its transitions will be correlated [169] which naturally indicates that a single Markov Chain step is insufficient.

Consider now the k-thinned chain $\{Z_t^k\}$ which retains every k-th entry of $\{Z_t\}$. The k-thinned chain will have smaller autocorrelation and begin to resemble independent draws from a distribution for sufficiently large k. At some point, the thinned time-series should resemble an independent process more than a first-order Markov process. To determine which of the models is a better fit, the Bayesian Information Criterion (BIC) is computed using the G^2-statistic [34] (see [155] for details). Thus, in a second step, the time-series $\{Z_t\}$ is progressively thinned to determine independent edges for the thinned time-series $\{Z_t^k\}$ for increasing values of k.

For our purposes, instead of first computing the whole time-series $\{Z_t\}$ and then considering increasing thinning values in a post-processing step, we define a fixed set of thinning values T and aggregate relevant entries of $\{Z_t\}$ on-the-fly for each $k \in T$. While this approach is far less memory-consuming, we cannot recover for each edge the earliest point of time it would have been deemed independent. Instead, for a thinning value k, we report the fraction of edges that would be deemed independent irrespective of a smaller thinning value $k' < k$.

In this context, we compare *ES-MC* and *G-ES-MC*. In order to visually align the results we define a *superstep* for both Markov Chains. To this end, let $m/2$ uniform

[5]We exclude: *dimacs*, *dimacs10*, *graph500* (benchmark graphs), *dynamic*, *misc* (unclassified graphs), *rand* (synthetic graphs) and *tscc* (temporal graphs).

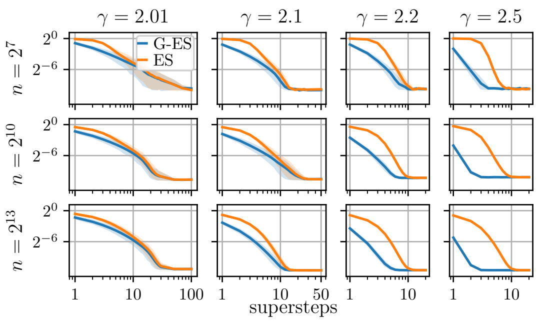


Figure 3.2: Fraction of non-independent edges as a function of the thinning value k as a multiple of the supersteps for the SynPld dataset where $(n, \gamma) \in \{2^7, 2^{10}, 2^{13}\} \times \{2.01, 2.1, 2.2, 2.5\}$. Each data point is represented by its mean value μ (line) and its 2σ error (shade).

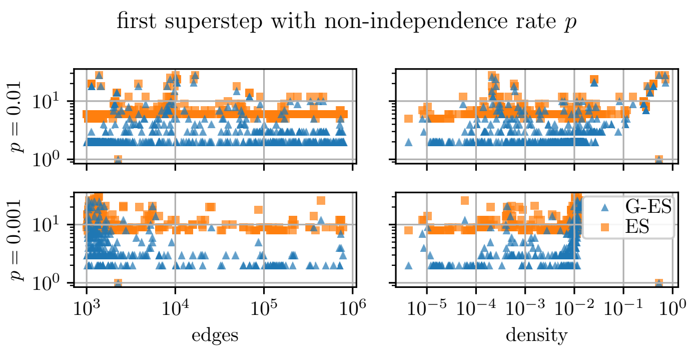


Figure 3.3: Scatterplots for the NetRep dataset where the x-coordinate either denotes the number of edges m (left) or the density $m/\binom{n}{2}$ (right) of a graph with n nodes and m edges. The y-coordinate represents the first superstep k at which the mean fraction of non-independent edges drops below either $1 \cdot 10^{-2}$ (top) or $1 \cdot 10^{-3}$ (bottom). The outlier that merely requires $k = 1$ supersteps for both Markov Chains possesses only two unique node degrees.

random edge switches and one uniform random global switch be a designated superstep for *ES-MC* and *G-ES-MC*, respectively. This accounts for the fact that one global switch potentially executes $m/2$ (non-uniform) edge switches.

We first consider the SynPld dataset and generate for each $(n, \gamma) \in \{2^7, 2^{10}, 2^{13}\} \times \{2.01, 2.1, 2.2, 2.5\}$ forty power-law graphs.[6] In Figure 3.2 we report the mean fraction of non-independent edges depending on the number of supersteps for a subset of the node counts and degree exponents. For highly skewed degree sequences, e.g., $\gamma = 2.01$, the *G-ES-MC* performs slightly better than the *ES-MC* for small supersteps. Increasing the number of supersteps results in matching performances for both. For larger degree exponents $\gamma \geq 2.2$ *G-ES-MC* consistently outperforms *ES-MC* where the advantage increases with γ. We observe both features for up to two orders of magnitude, and we expect this to hold for even larger values of n.

Next we investigate real-world graphs of the NetRep dataset. Due to the high computational cost, we restrict ourselves to graphs with $1000 \leq m \leq 800{,}000$ edges. To further reduce the cost, we perform the autocorrelation analysis only for the edges of the initial graph, reducing the memory footprint of each thinning to $\Theta(m)$ where m is the number of edges. In Figure 3.3 we present for each graph the first reported superstep[7] at which the mean fraction of non-independent edges of at least 15 runs is below a threshold τ. For $\tau = 1 \times 10^{-2}$, *G-ES-MC* seems to consistently outperform *ES-MC* except for very dense graphs where the performance is similar. The $\tau = 1 \times 10^{-3}$ is reached by only 46% of the 594 instances within 30 supersteps. Here, *G-ES-MC* still

[6] We limit the largest node count to $n = 2^{13}$ since the longest individual run already took 18 hours using an Intel Skylake Gold 6148 processor.

[7] We do not use large primes and numbers with many divisors as thinning values. This yields an uneven (but inconsequential) quantization of the y-axis.

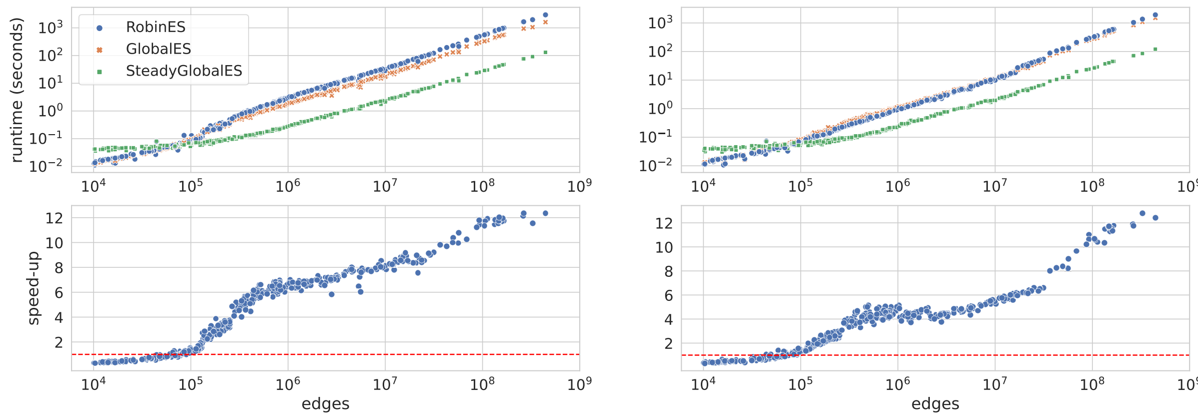


Figure 3.4: Scatterplot showing runtimes and speed-ups on NetRep. Top row: runtimes of RobinES and GlobalES on $P = 1$ PU and SteadyGlobalES on $P = 32$ PUs. The color and symbol indicates the algorithm. Bottom row: speed-ups of SteadyGlobalES over GlobalES. The height of the red dashed line corresponds to a speed-up of 1. Left column: without prefetching. Right column: with prefetching.

outperforms *ES-MC* on most instances except for moderately dense graphs on which both chains converge significantly slower.

3.6.2 Performance Benchmarks

In this section, we benchmark our *ES-MC* and *G-ES-MC* implementations. In each experiment, we run a subset of the implementations on the same initial graph and measure the average time required to initialize the data structures and perform 20 supersteps (e.g. 10 switches per edge). In practice, common choices [137, 81, 154] are 10 to 30 switches per edge. As *G-ES-MC* typically requires fewer supersteps (compare Section 3.6.1), this gives a slight advantage to *ES-MC* over the *G-ES-MC* implementations.

Runtime

We compare existing sequential implementations to our solutions and report absolute runtimes. To this end, we benchmark all implementations on a sample of graphs from NetRep and report their runtimes in Table 3.1. We select the graphs in this sample to cover a variety of sizes, average degrees and maximum degrees. As some of the networks are quite large, we set a timeout of 1000 seconds.

ES-MC and G-ES-MC: ☞ *Section 3.2*

We first compare RobinES and GlobalES, our sequential *ES-MC* and *G-ES-MC* solutions, with existing implementations from *NetworKit* [171] and *Gengraph* [181]. Our solutions run 15-50 times faster than *NetworKit* and 5-10 times faster than *Gengraph*. We also observe that GlobalES is faster than RobinES on large graphs, where shuffling is more efficient than sampling the edges, whereas RobinES runs faster on small graphs. In conclusion, our sequential implementations provide a meaningful baseline to measure further speed-ups.

Next, we report the runtimes of the parallel algorithms. For $P = 32$ PUs, all parallel implementations run much faster than the sequential implementations. On the largest

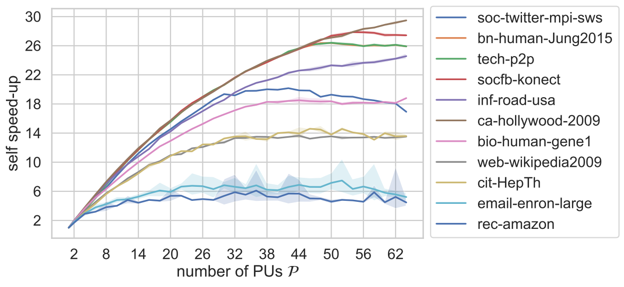


Figure 3.5: Scaling of SteadyGlobalES on a sample of graphs from NetRep for $1 \leq P \leq 64$. The line colors indicate the graph and are sorted by graph size.

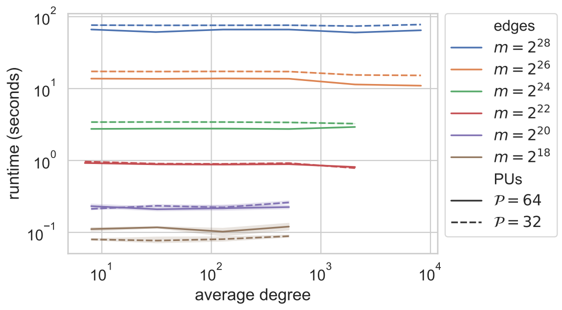


Figure 3.6: Runtime of SteadyGlobalES on graphs from SynGnp where $m \in \{2^{18}, 2^{20}, 2^{22}, 2^{24}, 2^{26}\}$ in dependence of the average degree $\overline{d} = 2m/n$. The color indicates the number of edges and the line style the number of PUs.

graph, only the parallel implementations were able to perform 20 supersteps before the timeout. Here, SteadyGlobalES is up to 12 times faster than GlobalES. On all graphs, SteadyGlobalES only shows a slowdown of at most 2 compared to EagerES.

SteadyGlobalES: ☞ *Section 3.4.2*

EagerES: ☞ *Section 3.4.1*

In Figure 3.4, we evaluate RobinES, GlobalES and SteadyGlobalES on all graphs from NetRep with at least $m = 10^4$ edges. For each graph, we run RobinES and GlobalES on one PU and SteadyGlobalES on $P = 32$ PUs and report the absolute runtimes and the speed-up of SteadyGlobalES over GlobalES. On all graphs with $m > 10^5$, the parallel algorithm is faster than the sequential algorithms and the observed speed-up increases with the size of the graph.

Scaling

We first report the self speed-up of SteadyGlobalES on the sample of graphs from NetRep in dependence of P (Figure 3.5). For larger graphs, the maximum speed-up ranges between 20 and 30 using 32 to 64 PUs. On the two smallest graphs, the work is likely too small to be efficiently parallelized (e.g. compare Table 3.1). An outlier is the largest graph (soc-twitter-mpi-sws); here, we expect the highest speed-up, but it is possible that a property of this graph, such as the high maximum degree, increases the runtime.

To measure the influence of the graph properties on the runtime, we consider synthetic graphs. We first examine the influence of the *density* of the graph, i.e. the average degree $\overline{d} = 2m/n$. To this end, we benchmark SteadyGlobalES on graphs from SynGnp with various n and edge probability $p = \overline{d}/(n-1)$, and plot the runtime as a function of the average degree in Figure 3.6. The density seems to have no significant effect on the runtime, even in the case where the average degree approaches the possible maximum $n - 1$ (bottom-right in the plot). Hence, the density of the graph does not seem to explain the variance in the speed-up we observed in Figure 3.5. On the other hand, we can conclude that the algorithm offers a stable performance regardless of the density of the graph.

density
$\overline{d}$: *average degree*

Table 3.1: Runtimes in seconds on a sample of graphs from NetRep sorted by network size. The left columns lists the graph, number of nodes n, number of edges m and maximum degree $d_{\max}$. The center columns list the sequential and parallel implementations for $P = 1$ PU. The right columns list the parallel implementations for $P = 32$ PUs. The best time in each group is indicated by the bold font. A dash (—) indicates a runtime of more than 1000 sec. for 20 supersteps.

				NetworKit	Gengraph	RobinES	GlobalES	EagerES	StdGlbES	EagerES	StdGlbES
Graph	n	m	$d_{\max}$	$P = 1$						$P = 32$	
soc-twitter-mpi-sws	41 M	1.2 B	2.9 M	—	—	—	—	—	—	**251**	397
bn-human-Jung2015	1.8 M	146 M	8.7 K	—	—	517	**460**	**448**	784	**20.0**	36.7
tech-p2p	5.7 M	140 M	675 K	—	—	530	**464**	**477**	788	**21.3**	37.2
socfb-konect	59 M	92 M	4.9 K	—	—	287	**253**	**228**	459	**11.9**	21.7
ca-holywood2009	1 M	56 M	11 K	—	686	140	**112**	**116**	244	**8.1**	11.4
inf-road-usa	23 M	28 M	9	619	186	49.2	**41.4**	**53.6**	97.0	5.2	**5.1**
bio-human-gene1	220 K	12 M	7.9 K	512	109	**12.3**	12.5	**18.1**	32.0	**1.3**	2.0
web-wikipedia2009	1.8 M	4.5 M	2.6 K	65.4	36.5	**4.7**	4.9	**6.6**	9.7	**0.58**	0.95
cit-HepTh	22 K	2.4 M	8.7 K	45.0	20.4	**2.2**	2.3	**3.4**	5.3	**0.25**	0.47
email-enron-large	33 K	180 K	1.3 K	0.92	0.44	**0.12**	0.14	**0.21**	0.37	**0.060**	0.073
rec-amazon	91 K	120 K	5	0.57	0.16	**0.098**	0.11	0.18	**0.17**	0.060	**0.045**

Next, we consider power-law graphs from SynPld with degree exponent $3 \geq \gamma \geq 2.01$ to evaluate the influence of the degree distribution's skewness. Note that increasing γ increases the number of edges even when fixing n, therefore we normalize the runtime by dividing by the number of edges. We report the runtime per edge as a function of γ in Figure 3.7. We observe an effect both in n and γ. For $n \geq 2^{26}$, the runtime on 64 PUs increases slightly as γ approaches 2. A likely explanation is that in a graph with highly skewed degree sequence, most edge switches will attempt to create the same few edges[8], causing many target dependencies and more synchronization overhead. This would also explain why the speed-up on the largest graph was lower than expected; checking the properties of this graph, we see that the maximum degree is close to n, but the average degree is rather small, which suggests a highly skewed degree sequence.

Recall that SteadyGlobalES may delay edge switches to resolve target dependencies. It does so by executing several rounds. To study the performance impact, we execute 20 global switches per graph of NetRep using $P = 32$ PUs and record the number of rounds per global switch, and the time accumulated on all rounds excluding the first one. As reported in Figure 3.8, the average number of rounds is low with a mean of 2.2. Only one global switch required 8 rounds and none more. This is a consequence of observation 3.6; as soon as the first edge switch of an insertion dependency chain is successful, the remaining ones can be rejected in the next round. As most edge switches are decided in the first round, the runtime impact of the following iterations is negligible for sufficiently large graphs: for all networks with more than 4M edges the first round accounts for more than 99% of the runtime.

[8]Recall that the probability that a random edge switch creates edge $e = \{u, v\}$ is proportional to $\deg(u) \cdot \deg(v)$.

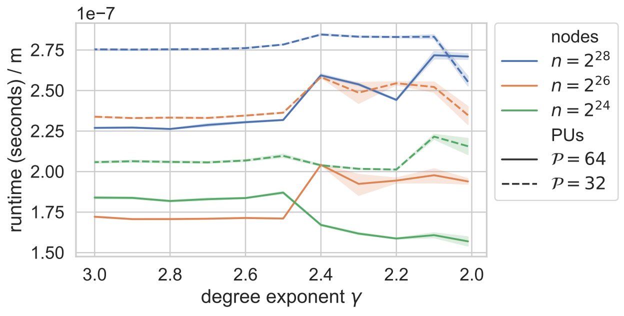


Figure 3.7: Runtime per edge of SteadyGlobalES on graphs from SynPld where $n \in \{2^{24}, 2^{26}, 2^{28}\}$ in dependence of the degree exponent γ. The color indicates the number of nodes and the line style the number of PUs.

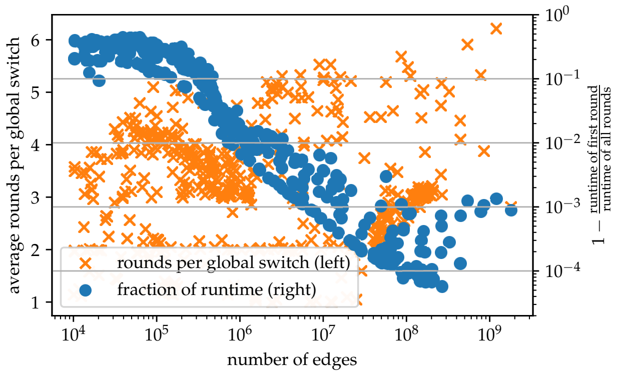


Figure 3.8: Rounds per global trade performed by SteadyGlobalES on NetRep. Each orange cross corresponds the average number of rounds recorded for a graph; the blue dots indicate the fractional runtime of all rounds excluding the first one.

3.7 Conclusions

We propose *G-ES-MC*, a novel *ES-MC* variant that converges to the uniform distribution on $\mathcal{G}(\mathcal{D})$. Our parallel algorithm SteadyGlobalES faithfully implements this Markov Chain. To the best of our knowledge, it is the first parallel algorithm to implement an *ES-MC* variant that eliminates possible errors due to concurrent/racing switches.

Our experiments suggest that *G-ES-MC* typically requires fewer steps than standard *ES-MC* to randomize a graph. On $P = 32$ PUs, our parallel algorithm executes $10 - 12$ times faster than our sequential *G-ES-MC* implementation, and $50 - 100$ faster than existing *ES-MC* implementations.

We investigate the influence of graph properties and provide evidence that the runtime of the algorithm is not affected by the density of the graph. For large graphs $n \geq 2^{26}$, and very small degree exponents $\gamma < 2.2$, there is a slight slowdown. On the other hand, in our experiment on over 600 real graphs, this occurs for only very few outliers. The last experiment suggests that SteadyGlobalES requires only few rounds to perform a global switch. This indicates that the underlying principle works well to parallelize the execution. We expect that dedicated base cases for small graphs can further reduce the overhead due to the synchronization and concurrent data structures and thereby improve the scaling on such graphs.

Acknowledgments

This work was partially supported by the Deutsche Forschungsgemeinschaft (DFG) under grants ME 2088/4-2 (SPP 1736 — Algorithms for BIG DATA) and ME 2088/5-1 (FOR 2975 — Algorithms, Dynamics, and Information Flow in Networks).

Parallel and I/O-efficient Randomization of Massive Networks using Global Curveball Trades

4

joint work with C.J. Carstens, M. Hamann, U. Meyer, M. Penschuck, 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 [81].
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* [47]. 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* [47] 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 [91] nearly one order of magnitude faster.

This chapter is based on the peer-reviewed conference article [48]:

[48] 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

Manuel Penschuck and I are main authors of this paper. Together, we contributed most of the algorithms and their implementations.

4.1 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 [102, 146, 173]. While simple models like Erdős-Rényi graphs [64] are easy to generate and analyze, they are too different from commonly observed powerlaw degree sequences [146, 143, 173]. Thus, random graphs with the same degree sequence as the given graph are frequently used [55, 102, 163]. 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 [164, 163]. The randomization of a given graph is commonly implemented as an *Edge Switching* [55, 137].

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 [114, 116]. [91] 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 [139] was studied to reduce the initial bias and the number of *ES* steps necessary [91]. 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 [174, 180] and has been adopted for graphs [46, 47]: 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 node[1] in a graph once [47]. 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.

4.1.1 Our Contributions

We present the first efficient algorithms for Curveball: the (sequential) internal memory and external memory algorithms IM-CB[2] and EM-CB for the Simple Undirected Curveball algorithm (see Section 4.4). Experiments in Section 4.5, indicate that they are faster than the established edge switching approaches in practice.

[1]For an odd number n of nodes, a single node is left out

[2]We 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 4.3, we show that random global trades lead to uniform samples of simple, undirected graphs and demonstrate experimentally in Section 4.5 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 4.5).

4.2 Preliminaries and Notation

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

Graphs and Degree Sequences

A graph $G = (V, E)$ has $n = |V|$ sequentially numbered nodes $V = \{v_1, \ldots, v_n\}$ 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 $\forall v_i \in V\colon\ \deg(v_i) = d_i$.

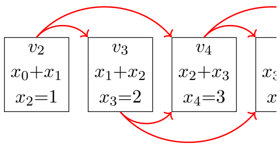
Randomization and Distributions

$\textsc{Pld}\,([a, b), \gamma)$ 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 $\textsc{Pld}\,([a, b), \gamma)$ 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 $\operatorname{rank}_\sigma(x)$ as the number of elements positioned in front of x by σ.

4.2.1 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 [1] 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 4: 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 ← 0
4     while PQ.min.key == $i$ do
5         sum ← sum + PQ.removeMin().val              // Receive all messeges 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 $\mathrm{scan}(n) := \Theta(n/B)$ I/Os. Sorting n consecutive items triggers $\mathrm{sort}(n) := \Theta((n/B) \cdot \log_{M/B}(n/B))$ I/Os. For all realistic values of n, B and M, $\mathrm{scan}(n) < \mathrm{sort}(n) \ll n$. Sorting complexity constitutes a lower bound for most intuitively non-trivial EM tasks [133]. EM queues use amortized $\mathcal{O}(1/B)$ I/Os per operation and require $\mathcal{O}(B)$ main memory [123]. An external priority queue (PQ) requires $\mathcal{O}(\mathrm{sort}(n))$ I/Os to push and pop n items [14, 13].

4.2.2 TFP: Time Forward Processing

Time Forward Processing (*TFP*) is a generic technique to manage data dependencies of external memory algorithms [123]. 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 4). 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 [14, 13], *TFP* incurs $\mathcal{O}(\mathrm{sort}(k))$ I/Os, where k is the number of messages sent.

4.3 Randomization Schemes

Here, we summarize the randomization schemes *ES* [137] and Curveball for simple undirected graphs [46], 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. [120]). Its *mixing time* indicates the number of steps necessary to reach the stationary distribution.

4.3.1 Edge Switching

Edge Switching is a state-of-the-art randomization method with a wide range of applications, e.g. the generation of graphs [91, 116], or the randomization of biological datasets [101]. 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 [62], aperiodic and symmetric [81] and hence converges to the uniform distribution on the space of simple graphs with fixed degree sequence. While analytic bounds on the mixing time [84, 85] are impractical, usually a number of steps linear in the number of edges is used in practice [154].

4.3.2 Simple Undirected Curveball Algorithm

Curveball is a novel randomization method. In each step, two nodes trade their neighborhoods, possibly yielding faster mixing times [46, 174, 180].

Definition 4.1 (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 $\sigma\colon D_{ij} \to D_{ij}$ 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 4.1, performing t on G results in

$$
\begin{aligned}
B_i &= (A_i \setminus A_{i-j}) \cup \{x \mid x \in D_{ij}, \mathrm{rank}_\sigma(x) \leq |A_{i-j}\} \quad \text{and} \\
B_j &= (A_j \setminus A_{j-i}) \cup \{x \mid x \in D_{ij}, \mathrm{rank}_\sigma(x) > |A_{i-j}|\}\,.
\end{aligned}
$$

Since edges are undirected, symmetry has to be preserved: for all $u \in A_i \backslash B_i$ the label j in adjacency list B_u is changed to i and analogously for $A_j \setminus B_j$. ◀

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 [47].

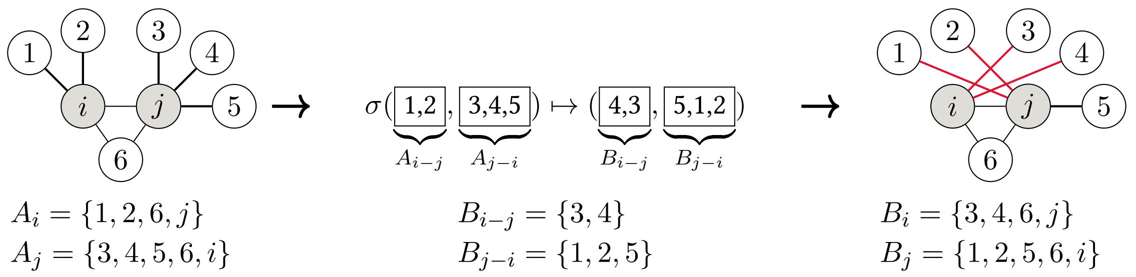


Figure 4.1: The trade (i, j, σ) between nodes i and j only considers edges to the disjoint neighbors $\{1, \ldots, 5\}$. For the reassigned disjoint neighbors we use the short-hand $B_{i-j} := \{x \mid x \in D_{ij}, \mathrm{rank}_\sigma(x) \le |A_{i-j}|\}$ and $B_{j-i} := \{x \mid x \in D_{ij}, \mathrm{rank}_\sigma(x) > |A_{i-j}|\}$. 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.

4.3.3 Undirected Global Trades

Trade sequences typically consist of pairs in which each constituent is drawn uniformly at random. While it is a well known fact[3] 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 4.5 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 [47]. 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 node[4]). Dependencies are then resolved by the order of this sequence.

Definition 4.2 (Undirected Global Trade). Let $G = (V, E)$ be a simple undirected graph and $\pi\colon V \to V$ be a permutation on the set of nodes. A *global trade* $T = (t_1, \ldots, t_\ell)$ 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 $t_1, \ldots, t_\ell$ are applied successively starting with G. ◀

Theorem 4.3 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 [47], which shows convergence of global trades in bipartite or directed graphs, to undirected graphs and uses similar techniques.

Theorem 4.3. 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.

[3]For instance studied as the coupon collector problem.
[4]This 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 [45] 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 [45] carries over.

It remains to show that the transition probabilities are symmetric. Let $\mathcal{T}^g_{AB}$ 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 $\mathcal{T}^g_{AB}$. That is $P_{AB} = \sum_{T \in \mathcal{T}^g_{AB}} \mathbf{P}_A(T)$ 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 [47]. We now define the reverse global trade of $T = (t_1, \ldots, t_\ell)$ as $\tilde{T} = (\tilde{t}_\ell, \ldots, \tilde{t}_1)$. It is straight-forward to check that this gives a bijection between the sets $\mathcal{T}^g_{AB}$ and $\mathcal{T}^g_{BA}$.

It remains to show that the middle equality holds in

$$P_{AB} = \sum\nolimits_{T \in \mathcal{T}^g_{AB}} \mathbf{P}_A(T) \overset{!}{=} \sum\nolimits_{\tilde{T} \in \mathcal{T}^g_{BA}} \mathbf{P}_B(\tilde{T}) = P_{BA}.$$

Let $T = (t_1, \ldots, t_\ell)$ be a global trade from state A to state B as implied by π and $A = A_1, \ldots, A_{\ell+1} = B$ be the intermediate states. We denote the reversal of T and π as $\tilde{T}$ and $\tilde{\pi}$ respectively and obtain

$$P_A(T) = \mathbb{P}[\pi]\, \mathbf{P}_{A_1}(t_1) \ldots \mathbf{P}_{A_\ell}(t_\ell) = \mathbb{P}[\tilde{\pi}]\, \mathbf{P}_B(\tilde{t}_\ell) \ldots \mathbf{P}_{A_2}(\tilde{t}_1) = P_B(\tilde{T}).$$

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. □

4.4 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 4.3.2) 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

[5]Since 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 $\mathcal{A}_{v_i}$, shuffle[6] 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 $\operatorname{scan}(|\mathcal{A}_{u_i}| + |\mathcal{A}_{v_i}|)$ I/Os (see also proof of Lemma 4.6 if the neighborhoods fit into RAM). Hence we focus on the harder task of obtaining and updating the adjacency information.

4.4.1 EM-CB: A Sequential I/O-efficient Curveball Algorithm

EM-CB (Algorithm 5) 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 4.4.2 and 4.4.4 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* principle, 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 $\mathscr{S}(u)$ of trades in which u is actively processed, i.e. $\mathscr{S}(u) := \big[\, i \mid u \in t_i \text{ for } i \in [\ell] \,\big] \circ [\infty]$.

[6]In contrast to Definition 4.2, 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.

[7]If an edge connects two nodes that are both actively traded we implicitly perform an arbitrary tie-break.

Algorithm 5: 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 $\mathscr{S}(u) = [t_1, \ldots, t_k]$ 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 PqTtoT
               else                      Send message $\langle t_1^v, v, u, t_1 \rangle$ via PqTtoT
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 $\mathscr{A}_G(u)$, $\mathscr{A}_G(v)$ and their successors $\tau(\cdot)$ from PqTtoT
18     Randomly reassign disjoint neighbors, yielding new neighbors $\mathscr{A}'_G(u)$ and $\mathscr{A}'_G(v)$.
19     foreach $(a, b) \in (\{u\} \times \mathscr{A}'_G(u)) \cup (\{v\} \times \mathscr{A}'_G(v))$ 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 PqTtoT
           else                                             Send message $\langle \tau_b, b, a, \tau_a \rangle$ via PqTtoT
```

Example 4.4. Let $G = (V, E)$ be a simple graph with $V = \{v_1, v_2, v_3, v_4\}$ and trade sequence $T = [t_1\colon \{v_1, v_2\}, t_2\colon \{v_3, v_4\}, t_3\colon \{v_1, v_3\}, t_4\colon \{v_2, v_4\}, t_5\colon \{v_1, v_4\}]$. Then, the successors $\mathscr{S}$ follow as $\mathscr{S}(v_1) = [1, 3, 5, \infty]$, $\mathscr{S}(v_2) = [1, 4, \infty]$, $\mathscr{S}(v_3) = [2, 3, \infty]$, $\mathscr{S}(v_4) = [2, 4, 5, \infty]$. ◀

This information is then spread via two channels:

- After preprocessing, EM-CB scans $\mathscr{S}$ 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 $\mathscr{S}$ conjointly. Later, it can be inductively computed since each trade receives the successors of all nodes involved.

Lemma 4.5. For an arbitrary trade sequence T of length ℓ, EM-CB has a worst-case I/O complexity of $\mathcal{O}\left[\operatorname{sort}(\ell) + \operatorname{sort}(n) + \operatorname{scan}(m) + \ell d_{\max}/B \log_{M/B}(m/B)\right]$. For r global trades, the worst-case I/O complexity is $\mathcal{O}(r[\operatorname{sort}(n) + \operatorname{sort}(m)])$. ◀

Proof. Refer to Section 4.A (Appendix) for the proof. □

4.4.2 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 4.5, 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 sorters[8] by unstructured I/O into main memory (see Algorithm 6 (Appendix) for details):

- Instead of sending neighborhood information in a *TFP* fashion, we now rely on a classical adjacency vector data structure $\mathscr{A}_G$ (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 $\mathscr{S}$ in a contiguous block of memory. The algorithm additionally maintains the vector $\mathscr{S}_{\text{idx}}[1 \dots n]$ 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 4.6. 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 4.B (Appendix) for the proof. □

4.4.3 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 4.3.3, a global trade can be encoded as a permutation $\pi\colon [n] \to [n]$ 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 $\mathscr{S}$ and communicate successors.

[8]The 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 [24].

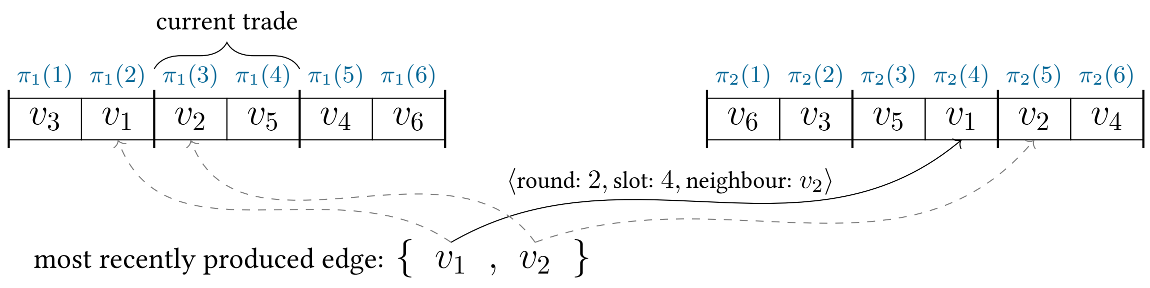

current trade
$\pi_1(1)$ $\pi_1(2)$ $\pi_1(3)$ $\pi_1(4)$ $\pi_1(5)$ $\pi_1(6)$
v_3 v_1 v_2 v_5 v_4 v_6
$\pi_2(1)$ $\pi_2(2)$ $\pi_2(3)$ $\pi_2(4)$ $\pi_2(5)$ $\pi_2(6)$
v_6 v_3 v_5 v_1 v_2 v_4
$\langle$round: 2, slot: 4, neighbour: $v_2\rangle$
most recently produced edge: $\{ v_1 , v_2 \}$

Figure 4.2: 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 4.2, 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 $\mathscr{S}_{j,i_2}[v_x] = (j, \pi_j(x))$ if v_x is processed later in round j (i.e. $\pi_j(x)/2 > i_2$) and otherwise $\mathscr{S}_{j,i_2}[v_x] = (j{+}1, \pi_{j+1}(x))$. 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 $f\colon X \to Y$ 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 4.C (Appendix)).

In practice, we use functions from the family of linear congruential maps H_p where p is the smallest prime number $p \geq n$:

linear congruential map

$$H_p := \{ h_{a,b} \mid 1 \leq a < p \text{ and } 0 \leq b < p \} \tag{4.1}$$
$$h_{a,b}(x) \equiv (ax + b) \mod p, \tag{4.2}$$

As detailed in Section 4.D (Appendix) random choices from H_p are well suited for EM-GCB since they are 2-universal[9] and contain only $\mathcal{O}(\log(n))$ gaps. They are also bijections with an easily computable inverse $h^{-1}_{a,b}$ that allows EM-GCB to determine the active node $h^{-1}_{a,b}(i)$ 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*.

4.4.4 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 4.3, we split a global trade into *microchunks* each containing a similar

[9]i.e. given one node in a single trade, the other is uniformly chosen among the remaining nodes.

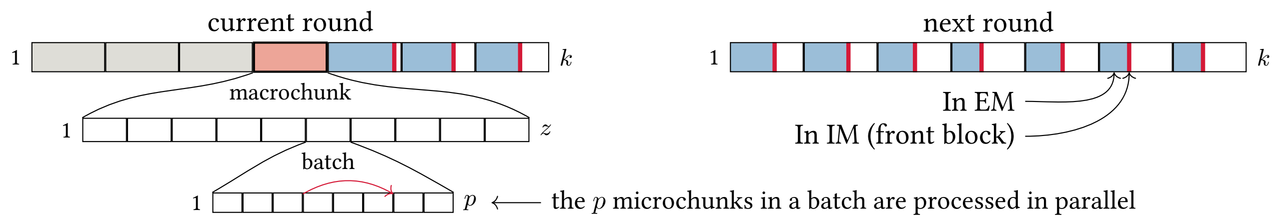


Figure 4.3: EM-PGCB splits each global trade into k *macrochunks* and maintains an external memory queue for each. Before processing a macrochunk, the buffer is loaded into IM and sorted, and further subdivided into z batches each consisting of p microchunks. A type (ii) message is visualized by the 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 4.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 $m = \mathcal{O}\left(M^2/B\right)$ edges[10], 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 4.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 4.F).

Theorem 4.7. 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 4.5) carries over, requiring $\text{sort}(m)$ I/Os for a global trade. □

[10]Even with as little as 1 GiB of internal memory, several billion edges are supported.

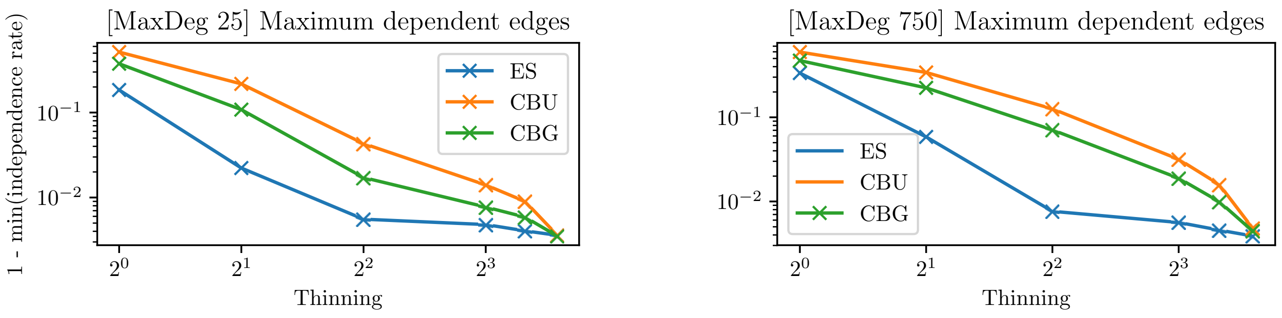


Figure 4.4: Fraction of edges still correlated as a function of the thinning parameter k for graphs with $n = 2{\cdot}10^3$ nodes and degree distribution $\textsc{Pld}\left([a,b),\gamma\right)$ with $\gamma = 2$, $a = 5$, and $b \in \{25, 750\}$. The (not thinned) long Markov chains of edge switching (ES), Curveball with uniform trades (CBU) and Curveball with global trades (CBG) contain 6000 super steps each.

4.5 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* [171]; 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* [60].

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 $\textsc{Pld}\left([a,b),\gamma\right)$ 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 [91] 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 [91]. Runtimes are measured on the following off-the-shelf machine: Intel Xeon E5-2630 v3 (8 cores at 2.40GHz), 64GB RAM, 2$\times$ Samsung 850 PRO SATA SSD (1 TB), Ubuntu Linux 16.04, GCC 7.2.

4.5.1 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

[11]Code 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 [155]. 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 [91].

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 $(Z^k_{e,t})_{t\geq 0}$ 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 $(Z^k_{e,t})_{t\geq 0}$ 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 4.4 contains a selection of results obtained for small powerlaw graph instances using this method (see Section 4.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 4.3.2).

In all settings *ES* shows the fastest decay. The gap towards global trades growths 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.

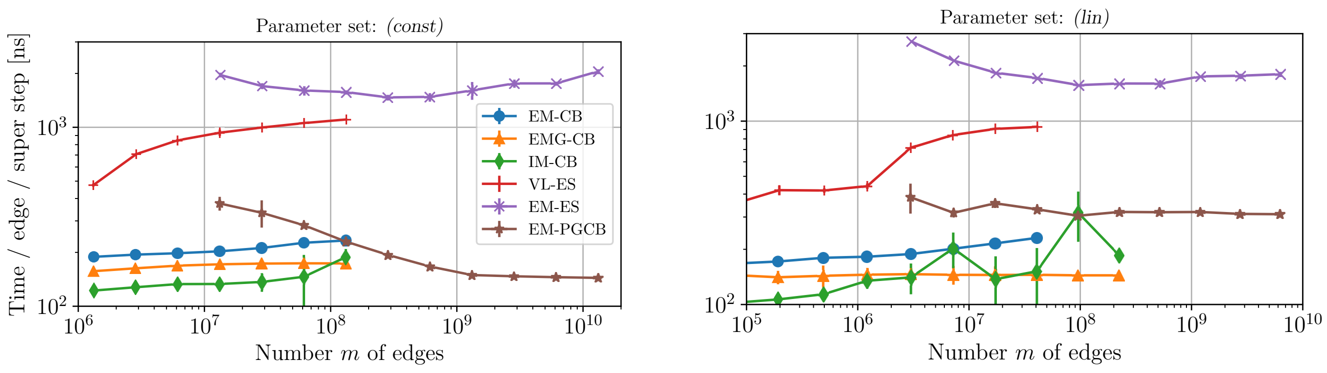


Figure 4.5: Runtime per edge and super step (global trade or m edge swaps) of the proposed algorithms IM-CB, EM-CB and EM-PGCB compared to state-of-the-art IM edge switching VL-ES and EM edge switching EM-ES. Each data point is the median of $S \geq 5$ runs over 10 super steps each. The left plot contains 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).

4.5.2 Runtime Performance Benchmarks

We measure the runtime of the algorithms proposed in Section 4.4 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 [181]. 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 [91].

EM-ES: ☞ *Section 2.5*

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 4.5 presents the wall-time per edge and super step including precomputation[12] 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 4.5.1). 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

[12]For 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 4.9 (Section 4.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.

4.6 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 4.A EM-CB

Lemma 4.5. 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 5, 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 PqTtoT 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 $\mathcal{O}(\deg(u_i) + \deg(v_i))$ messages are retrieved shuffled and redistributed causing $\mathcal{O}[\text{sort}(\deg(u_i) + \deg(v_i))]$ I/Os. The bound can be improved to $\mathcal{O}\Big((\deg(u_i) + \deg(v_i))/B \log_{M/B}(m/B)\Big)$ by observing that $\mathcal{O}(m)$ items are stored in the PQ at any time. For a worst-case analysis we set $\deg(u_i) = \deg(v_i) = d_{\max}$ 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 4.B IM-CB

Lemma 4.6. 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 6, the computation of $\mathscr{S}[\cdot]$ and its auxiliary structures involves scanning over T and V resulting in $\mathcal{O}(n + \ell)$ operations. Inserting all edges into $\mathscr{A}_G$ requires another $\mathcal{O}(n + m)$ steps.

The i-th trade takes $\mathcal{O}(\deg(v_i) + \deg(u_i))$ 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 $\mathcal{O}(\deg(v_i) + \deg(u_i))$. Since T's constituents are drawn uniformly at random, we estimate the neighborhood sizes as $\mathbb{E}[\deg(u_i)] = \mathbb{E}[\deg(v_i)] = m/n$ 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 $r\sum_i(\deg(u_i) + \deg(v_i)) = r\sum_{v\in V}\deg(v) = \mathcal{O}(rm)$. □

Appendix 4.C EM-GCB

Recall that a global trade can be encoded by a permutation $\pi\colon V \to V$ on the nodes or node indices (see Section 4.3.2). Consequently, generating a uniform random permuta-

Algorithm 6: IM-CB as detailed in Section 4.4.2.

```
Data: Trade sequence $T$, simple graph $G$
$\mathscr{S}_{\text{IDX}}[1 \dots n{+}1] \leftarrow 0$    // Compute $\mathscr{S}$: First count how often a node is active, ...
foreach $\{u, v\} \in T$ do
    $\mathscr{S}_{\text{IDX}}[u] \leftarrow \mathscr{S}_{\text{IDX}}[u] + 1$
    $\mathscr{S}_{\text{IDX}}[v] \leftarrow \mathscr{S}_{\text{IDX}}[v] + 1$
$\mathscr{S}_{\text{BEGIN}}[i] \leftarrow 1 + \sum_{j=1}^{i-1} \mathscr{S}_{\text{IDX}}[j] \;\; \forall 1 \leq i \leq n{+}1$  // Exclusive prefix sum with stop marker
copy $\mathscr{S}_{\text{IDX}} \leftarrow \mathscr{S}_{\text{BEGIN}}$
Allocate $\mathscr{S}[1 \dots 2\ell]$
foreach $t_i = \{u_i, v_i\} \in T$ for increasing $i$ do
    $\mathscr{S}[\mathscr{S}_{\text{IDX}}[u_i]] \leftarrow i$    // Compute $\mathscr{S}$: ...when it is active
    $\mathscr{S}_{\text{IDX}}[u_i] \leftarrow \mathscr{S}_{\text{IDX}}[u_i] + 1$

    $\mathscr{S}[\mathscr{S}_{\text{IDX}}[v_i]] \leftarrow i$
    $\mathscr{S}_{\text{IDX}}[v_i] \leftarrow \mathscr{S}_{\text{IDX}}[v_i] + 1$
reset $\mathscr{S}_{\text{IDX}} \leftarrow \mathscr{S}_{\text{BEGIN}}$
$\tau_{v_i}$ := if $(\mathscr{S}_{\text{IDX}}[i] == \mathscr{S}_{\text{BEGIN}}[i+1])$ then $\infty$ else $\mathscr{S}[\mathscr{S}_{\text{IDX}}[i]]$  // Short for read successor

// Fill $\mathscr{A}_G$
$\mathcal{A}_{\text{BEGIN}}[i] \leftarrow 1 + \sum_{j=1}^{i-1} \deg(v_j) \;\; \forall 1 \leq i \leq n{+}1$    // Prefix sum with stop marker
copy $\mathcal{A}_{\text{IDX}} \leftarrow \mathcal{A}_{\text{BEGIN}}$
Allocate $\mathscr{A}_G[1 \dots 2m]$
foreach $\{a, b\} \in E$ do
    if $\tau_a \leq \tau_b$ then push $b$ into $\mathscr{A}_G(a)$: $\mathscr{A}_G[\mathcal{A}_{\text{IDX}}[a]] \leftarrow b$;   $\mathcal{A}_{\text{IDX}}[a] \leftarrow \mathcal{A}_{\text{IDX}}[a] + 1$
    else            push $a$ into $\mathscr{A}_G(b)$: $\mathscr{A}_G[\mathcal{A}_{\text{IDX}}[b]] \leftarrow a$;   $\mathcal{A}_{\text{IDX}}[b] \leftarrow \mathcal{A}_{\text{IDX}}[b] + 1$

// Trade
foreach trade $t_i = (u, v) \in T$ for increasing $i$ do
    Gather neighbors $\mathscr{A}_G(u)$, $\mathscr{A}_G(v)$ from $\mathscr{A}_G$ using $\mathcal{A}_{\text{BEGIN}}$
    Reset $\mathcal{A}_{\text{IDX}}[u] \leftarrow \mathcal{A}_{\text{BEGIN}}[u]$, $\mathcal{A}_{\text{IDX}}[v] \leftarrow \mathcal{A}_{\text{BEGIN}}[v]$
    Advance $\mathscr{S}_{\text{IDX}}[u]$ and $\mathscr{S}_{\text{IDX}}[v]$, s.t. $\tau_u$ and $\tau_v$ gets next trades
    Randomly reassign disjoint neighbors, yielding new neighbors $\mathcal{A}_u$ and $\mathcal{A}_v$.
    foreach $(a, b) \in (\{u\} \times \mathscr{A}'_G(u)) \cup (\{v\} \times \mathscr{A}'_G(v))$ do
        // Push node edge into $\mathscr{A}_G$; same as line 18
        if $\tau_a < \tau_b$ then Push $b$ in $\mathscr{A}_G(a)$
        else            Push $a$ in $\mathscr{A}_G(b)$
```

tion on $[n]$ yields a uniform random global trade. Injective hash functions have several computational advantages and can substitute the random permutation:

Definition 4.8 (Relaxed global trade). Let $h\colon [n] \to \mathbb{N}$ 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 4.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 $0 \equiv p \mod p$. Additionally for a map $h : X \to Y$ we denote the image of h as $\mathbf{im}(h) = \{h(x) : x \in X\}$.

Definition 4.9 (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 $h \in H$

$$\mathbb{P}[h(x_1) = y_1 \wedge h(x_2) = y_2] = |Y|^{-2}.$$ ◀

Proposition 4.10. A linear congruential map $h_{a,b}\colon \mathbb{Z}_p \to \mathbb{Z}_p, x \mapsto ax + b \mod p$ for $a \neq 0$ and p prime is a bijection. ◀

Proof. The translation $\tau_b(x) = x + b \mod p$ and multiplication $\chi_a(x) = ax \mod p$ 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 $h_{a,b}^{-1}(y) = a^{-1}(y - b) \mod p$. □

Lemma 4.11. The ensemble $H = \{h_{a,b}\colon a \in \mathbb{Z}_p^*,\ b \in \mathbb{Z}_p\}$ is 2-universal. ◀

Proof. see Proposition 7 of [49]. □

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 4.16): 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 4.12 (Sorted rank-map). Let $n \in \mathbb{N}$. Further, let $h\colon [n] \to \mathbb{N}$ be an injective map restricted to $[n]$ and π_h be the permutation that sorts $[\, h(i) \,]_{i=1}^{n}$ ascendingly. Denote

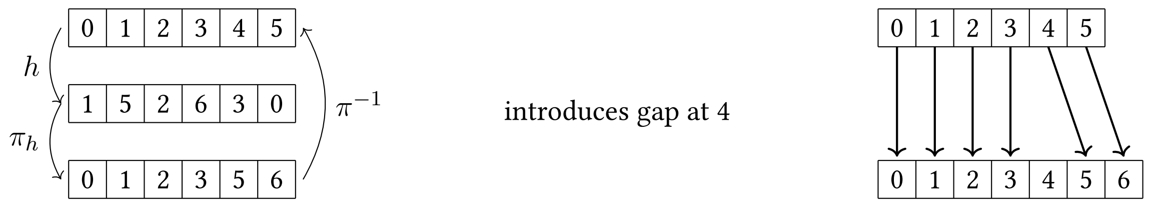


Figure 4.6: The sorted rank-map for $n = 6$ and $h\colon [n] \to \mathbb{Z}_7, x \mapsto 4x + 1$. For the set $\{0, 1, 2, 3\}$ the sorted rank-map π is just the identity. In contrast for $x \in \{4, 5\}$ the value x is mapped to $\pi(x) = x + 1$.

with $\pi = (h \circ \pi_h)\colon [n] \to \mathbf{im}(h)$ 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 4.6. ◀

Remark 4.13. The sorted rank-map π can only shift the original values and is thus monotonically increasing, see Figure 4.6. 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 4.6. ◀

Proposition 4.14. Let $n \in \mathbb{N}$ and $p \geq n$ be a prime number. Further, let $h\colon [n] \to \mathbb{Z}_p$ 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 4.6. 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 $s \in \{k, \ldots, k'\}$, resulting in inequalities

$$\pi^{-1}(y) + k \quad \leq \quad y \quad \leq \quad \pi^{-1}(y) + k'.$$

By subtracting k and k' on both sides, the claim follows. □

With Proposition 4.14 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 4.4.4).

Example 4.15. Let n and h be given from Figure 4.6. 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 4.14

$$2 - (\pi(5) - 5) \leq \pi^{-1}(2) \leq 2 - (\pi(0) - 0),$$
$$1 \leq \pi^{-1}(2) \leq 2.$$

Thus, the rank of 2 in $\mathbf{im}(h)$ is either 1 or 2. ◀

Appendix 4.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 4.F.

4.E.1 Data Structure for Message Transportation

Recall in Section 4.4.4 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 send 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 4.D) we can do better:

Proposition 4.16. 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 4.14 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 4.F we argue that this happens rarely. The processor that inserts the last message for that particular trade will perform it instead.

4.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 $m = \mathcal{O}\big(M^2/B\big)$ 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 7.

Algorithm 7: EM-PGCB as detailed in Section 4.4.4 and Section 4.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 $\mathscr{A}_G$ 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 $\mathscr{A}_G$
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
                             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 4.F Analysis of EM-PGCB

4.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 $n/k \cdot \mathrm{Var}(D)$ 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 $\max_{1\le i\le k} S_i$. The largest macrochunk may receive many high-degree nodes exceeding the size of the internal memory M. We thus bound its number in Corollaries 4.18 and 4.20.

Lemma 4.17. Let $Y = \max_{1\le i\le k} X_i$, where the X_i are iid r.v. distributed as $\mathcal{N}(0,\sigma^2)$. Then, $\mathbb{E}[Y] \le \sigma\sqrt{2\log(k)}$. ◀

Proof. The following chain of inequalities holds

$$e^{t\mathbb{E}[Y]} \le \mathbb{E}\left[e^{tY}\right] = \mathbb{E}\left[\max_{1\le i\le k} e^{tX_i}\right] \le \sum_{i=1}^{k} \mathbb{E}\left[e^{tX_i}\right] = ke^{t^2\sigma^2/2},$$

where in order Jensen's inequality[13] 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 $t \ne 0$) yields $\mathbb{E}[Y] \le \frac{\log(k)}{t} + \frac{t\sigma^2}{2}$, which is minimized by $t = \sqrt{2\log(k)}/\sigma$. The above proof is a special case in a proof of [125]. □

Corollary 4.18. Let $Y = \max_{1\le i\le k} S_i$. By approximating S_i with a Gaussian r.v. N_i with $\mu = \mathbb{E}[S_i]$ and $\sigma^2 = \operatorname{Var}(S_i)$, one gets an approximate upper bound on Y:

$$\mathbb{E}[Y] \approx \mathbb{E}\left[\max_{1\le i\le k} N_i\right] \le \mathbb{E}[S_1] + \sqrt{2\log(k)\operatorname{Var}(S_1)} = \mathbb{E}[S_1] + \sqrt{\frac{n\log(k)}{2k}\operatorname{Var}(D)}.$$

◀

Proof. Since $\max_{1\le i\le k} N_i$ is centered around μ, it is identically distributed to $\mu + \max_{1\le i\le k} N_i'$ where N_i' has the same variance but is centered around 0. By applying Lemma 4.17 to $\max_{1\le i\le k} N_i'$ the claim follows, since $\mathbb{E}[\max_{1\le i\le k} N_i] = \mu + \mathbb{E}[\max_{1\le i\le k} N_i']$. □

Lemma 4.19. Let $X_1, \dots, X_k$ be iid and $Y = \max\limits_{1\le i\le k} X_i$. Then, $\operatorname{Var}(Y) \le k\operatorname{Var}(X_1)$. ◀

Proof. For Z, Z' iid. $\mathbb{E}\left[(Z-Z')^2\right] = 2\operatorname{Var}(Z)$ holds, since

$$\mathbb{E}\left[Z^2 - 2ZZ' + Z'^2\right] = 2\mathbb{E}\left[Z^2\right] - 2\mathbb{E}[Z]^2.$$

Now, let $Y' = \max_{1\le i\le k} X_i'$ be an independent copy of Y and $r > 0$. First, the inequality $\mathbb{P}\left[|Y-Y'|^2 > r\right] \le \sum_{i=1}^{k}\mathbb{P}\left[|X_i - X_i'|^2 > r\right]$ 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 $X_i > X_j' \ge X_i'$ implies $|X_i > X_i'| > r$ and consequently $\mathbb{P}\left[|Y-Y'|^2 > r\right] \le \mathbb{P}[\exists i\colon X_i - X_i'| > r]$.

A union bound yields $\mathbb{P}\left[|Y-Y'|^2 > r\right] \le \sum_{i=1}^{k}\mathbb{P}\left[|X_i - X_i'|^2 > r\right]$. Integrating r from 0 to ∞ yields $2\operatorname{Var}(Y) = \mathbb{E}\left[(Y-Y')^2\right] \le k\mathbb{E}\left[(X_1 - X_1')^2\right] = 2k\operatorname{Var}(X_1)$, which concludes the proof. □

[13]Let f be convex. For a non-negative λ_i with $\sum_{i=1}^{n}\lambda_i = 1$ it follows $f(\sum_{i=1}^{n}\lambda_i x_i) \le \sum_{i=1}^{n}\lambda_i f(x_i)$.

Corollary 4.20. Let $Y = \max_{1 \leq i \leq k} S_i$. Then, $\mathrm{Var}(Y) \leq k \, \mathrm{Var}(S_1) = n \, \mathrm{Var}(D)$. ◀

Proof. This is a special case of Lemma 4.19. □

The probability mass of a Gaussian r.v. is concentrated around its mean, e.g. the tails vanish very quickly, see Proposition 4.21. This heuristically additionally holds true for the maximum macrochunk size (Lemma 4.22).

Proposition 4.21. 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

$$\mathbb{P}[X > t] \leq \exp(-t^2/2)/\sqrt{2\pi}/t = \mathcal{O}\left(\frac{e^{-t^2/2}}{t}\right).$$ ◀

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 4.22. 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 4.21 the claim follows. □

4.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 4.23. 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 $\operatorname{avg}\deg \frac{n}{kzp} = \frac{2m}{kzp}$ 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

$$\mathbb{E}[V_i] = \sum_{i=1}^{n} \mathbb{P}[i \in A] \, \mathbb{P}\left[i \in h^{-1}(\mathcal{B})\right] = n \frac{\deg_{\text{avg}}}{n} \frac{\frac{n(p-i)}{kzp}}{n} = \deg_{\text{avg}} \frac{p - i}{kzp}.$$

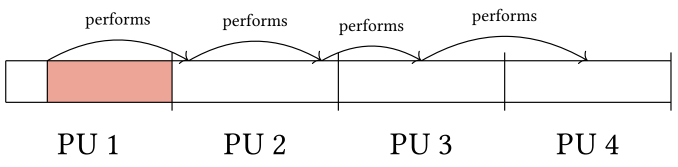


Figure 4.7: Work stealing of PU 1. The arrows represent a long work stealing chain of trades. The red marked area represents still untouched trades of the first microchunk that will get processed *after* the long chain by the first PU.

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

$$\mathbb{E}[H_i] = \frac{n}{kzp}\mathbb{E}[V_i] = \frac{2m(p-i)}{k^2z^2p^2}.$$

By summing over all p microchunks, e.g. $\sum_{i=1}^{p} \mathbb{E}[H_i]$ the claim follows. □

Example 4.24. Consider Lemma 4.23 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 4.23 predicts a count of less than 4 critical messages on average in a batch. ◀

Theoretically by Lemma 4.23 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 4.7. 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 4.25.

Proposition 4.25. Let X be geometrically distributed with parameter $(1 - 1/z^2)$ for $z > 1$. Then, $\mathbb{P}[X > t] = \frac{1}{z^{2t}} = e^{-2\ln(z)t}$. ◀

Proof. The claim follows by $\mathbb{P}[X > t] = 1/z^{2t}$ and setting $t = \mathcal{O}(1)$. □

Appendix 4.G Additional Experimental Results

4.G.1 Swaps Performed by Curveball and Global Curveball

In Figure 4.8 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.

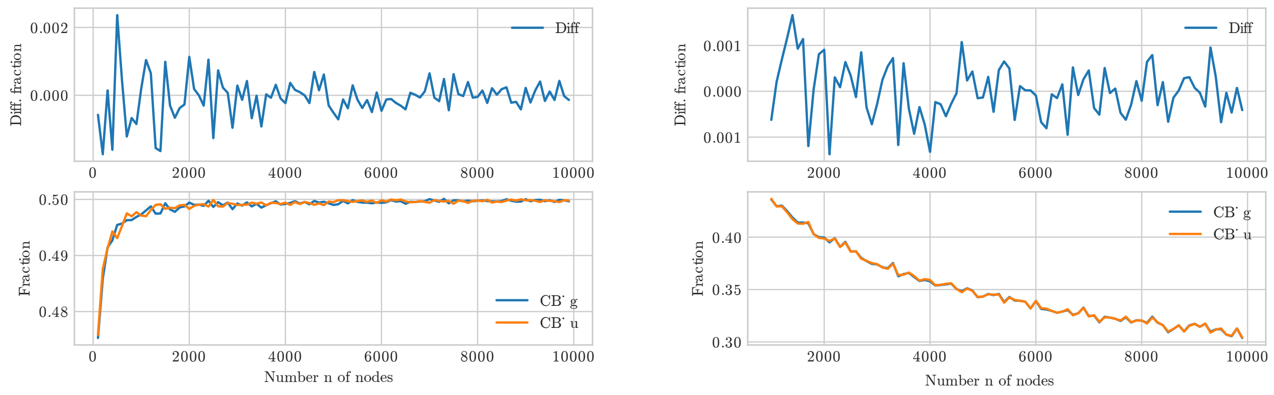


Figure 4.8: The average fraction of performed neighborhood swaps of $n/2$ uniform trades and a single global trade.
Left: 10-regular graphs for increasing n.
Right: powerlaw graphs realized from PLD $([10, n/20), 2)$ for increasing n by the HAVEL-HAKIMI algorithm.

4.G.2 Autocorrelation Time of Curveball and Edge Switching

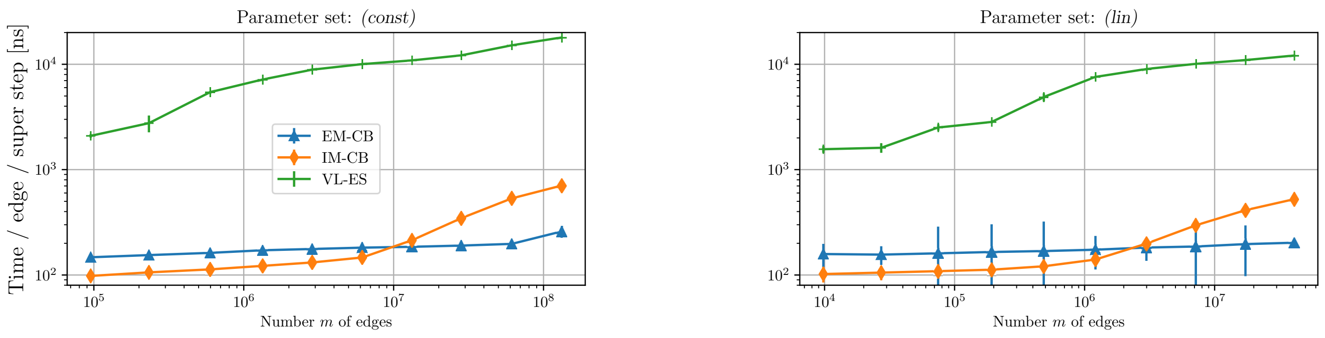


Figure 4.9: Runtime per edge and super step of IM-CB and EM-CB compared to state-of-the-art IM edge switching VL-ES. Each data point is the median of $S \geq 5$ runs over 10 super steps each. The left plot contains the (const)-parameter set, the right one (linear). Machine: Intel i7-6700HQ CPU (4 cores), 64 GB RAM, Ubuntu Linux 17.10.

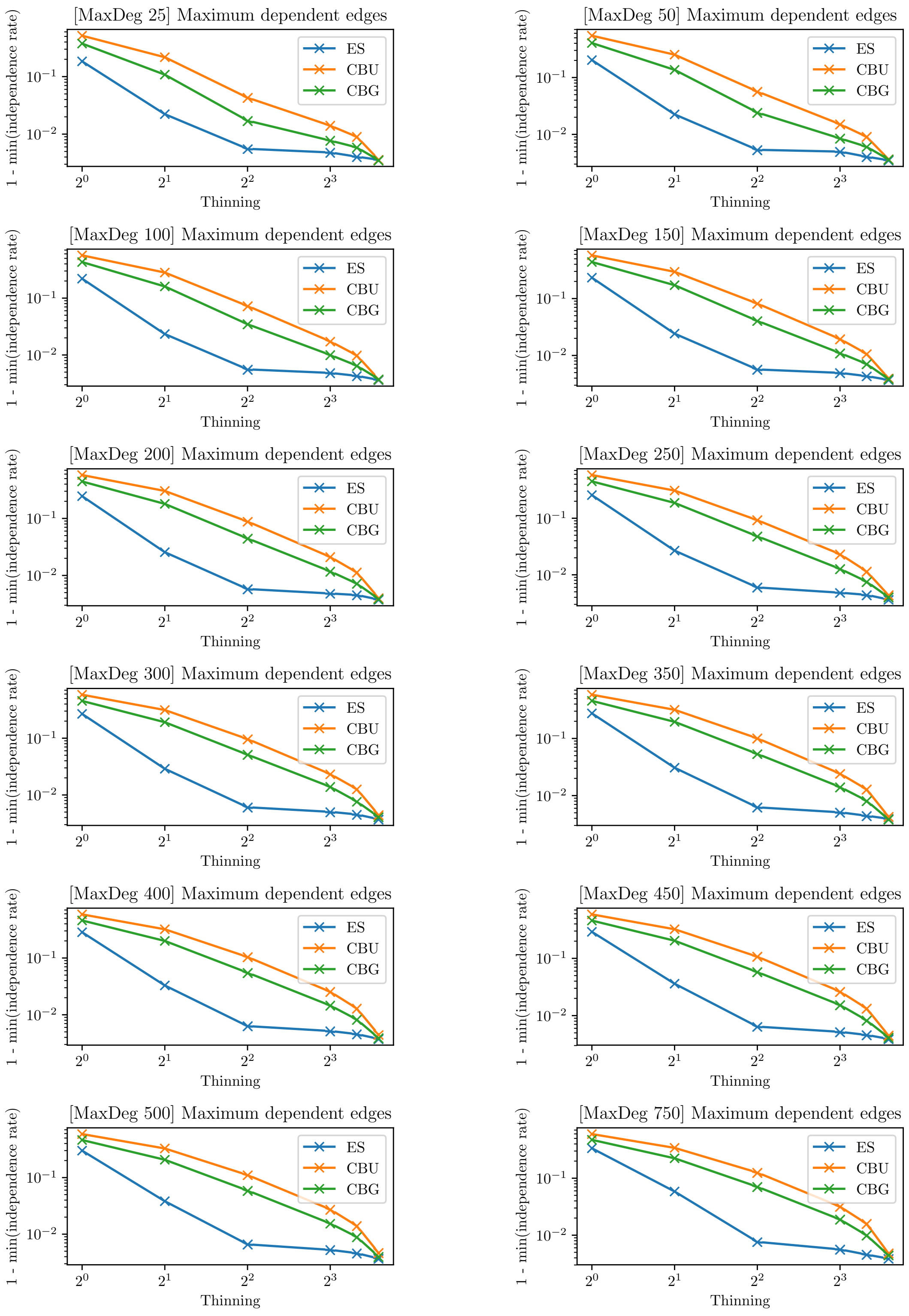


Figure 4.10: Fraction of edges still correlated as function of the thinning parameter k for graphs with $n = 2{\cdot}10^3$ nodes and degree distribution $\textsc{Pld}([a, b), \gamma)$ with $\gamma = 2$, $a = 5$, and several different values for b. The (not thinned) long Markov chains contain 6000 super steps each.

5 Engineering Uniform Sampling of Graphs with a Prescribed Power-law Degree Sequence

joint work with D. Allendorf, U. Meyer, M. Penschuck, and N. Wormald

We consider the following common network analysis problem: given a degree sequence $\mathcal{D} = (d_1, \ldots, d_n) \in \mathbb{N}^n$ return a uniform sample from the ensemble of all simple graphs with matching degrees. In practice, the problem is typically solved using Markov Chain Monte Carlo approaches, such as Edge-Switching or Curveball, even if no practical useful rigorous bounds are known on their mixing times. In contrast, Arman et al. sketch Inc-Powerlaw, a novel and much more involved algorithm capable of generating graphs for power-law bounded degree sequences with $\gamma \gtrapprox 2.88$ in expected linear time.
For the first time, we give a complete description of the algorithm and add novel switchings. To the best of our knowledge, our open-source implementation of Inc-Powerlaw is the first practical generator with rigorous uniformity guarantees for the aforementioned degree sequences. In an empirical investigation, we find that for small average-degrees Inc-Powerlaw is very efficient and generates graphs with one million nodes in less than a second. For larger average-degrees, parallelism can partially mitigate the increased running-time.

This chapter is based on the peer-reviewed conference article [9]:

[9] D. Allendorf, U. Meyer, M. Penschuck, H. Tran, and N. Wormald. Engineering uniform sampling of graphs with a prescribed power-law degree sequence. In C. A. Phillips and B. Speckmann, editors, *Proceedings of the Symp. on Algorithm Engineering and Experiments ALENEX*, pages 27–40. Society for Industrial and App. Math. SIAM, 2022. doi:10.1137/1.9781611977042.3 .

My contribution

I substantially contributed to the new booster switchings.

5.1 Introduction

A common problem in network science is the sampling of graphs matching prescribed degrees. It is tightly related to the random perturbation of graphs while keeping their degrees. Among other things, the problem appears as a building block in network models (e.g., [114]). It also yields null models used to estimate the statistical significance of observations (e.g., [136, 83]).

The computational cost and algorithmic complexity of solving this problem heavily depend on the exact requirements. Two relaxed variants with linear work sampling algorithms are Chung-Lu graphs [52] and the configuration model [26, 144, 38]. The Chung-Lu model produces the prescribed degree sequence only in expectation and allows for simple and efficient generators [135, 3, 140, 73]. The configuration model (see Section 5.2), on the other hand, exactly matches the prescribed degree-sequence but allows loops and multi-edges, which introduce non-uniformity into the distribution [144, p.436] and are inappropriate for certain applications; however, erasing them may lead to significant changes in topology [164, 181].

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

5.1.1 Related Work

An early uniform sampler with unknown algorithmic complexity was given by Tinhofer [177]. Perhaps the first practically relevant algorithm was implicitly given by graph enumeration methods (e.g. [25, 26, 39]) using the configuration model with rejection-sampling. While its time complexity is linear in the number of nodes, it is exponential in the maximum degree squared and therefore already impractical for relatively small degrees.

McKay and Wormald [128] increased the permissible degrees. Instead of repeatedly rejecting non-simple graphs, their algorithm may remove multi-edges using switching operations. For d-regular graphs with $d = \mathcal{O}\big(n^{1/3}\big)$, its expected time complexity is $\mathcal{O}\big(d^3 n\big)$ where n is the number of nodes; later, Gao and Wormald [77] improved the result to $d = o(\sqrt{n})$ with the same time complexity, and also considered sparse non-regular cases (e.g. power-law degree sequences) [78]. Subsequently, Arman et al. [17] present[1] the algorithms Inc-Gen, Inc-Powerlaw and Inc-Reg based on *incremental relaxation*. Inc-Gen runs in expected linear time provided $\Delta^4 = \mathcal{O}(m)$ where Δ is the maximum degree and m is the number of edges.

In the relaxed setting, where the generated graph is approximately uniform, Jerrum and Sinclair [105] gave an algorithm using Markov Chain Monte Carlo (MCMC) methods. Since then, further MCMC-based algorithms have been proposed and analyzed (e.g. [56, 81, 85, 107, 122, 170, 174, 180, 181]). While these algorithms allow for larger families of degree sequences, topological restrictions (e.g., connected graphs [81, 181]), or more general characterizations (e.g., joint degrees [170, 122]), theoretically proven upper

[1]Implementations of Inc-Gen and Inc-Reg are available at https://users.monash.edu.au/~nwormald/fastgen_v3.zip

bounds on their mixing times are either impractical or non-existent. Despite this, some of these algorithms found wide use in several practical applications and have been implemented in freely available software libraries [114, 171, 181] and adapted for advanced models of computation [30, 90, 48].

As generally fast alternatives, asymptotic approximate samplers (e.g. [78, 23, 110, 172, 186]) have been proposed. These samplers provide a weaker approximation than MCMC: the error tends to 0 as n grows but cannot be improved for any particular n.

5.1.2 Our Contribution

Arman et al. [17] introduce incremental relaxation and, as a corollary, obtain Inc-Powerlaw by applying the technique to the Pld algorithm [78]. Crucial details of Inc-Powerlaw were left open and are discussed here for the first time. For the parts of the algorithm that use incremental relaxation (see Section 5.2), we determine the order in which the relevant graph substructures should be relaxed, how to count the number of those substructures in a graph and find new lower bounds on the number of substructures, or adjust the ones used in Pld.

Our investigation also identified two cases where incremental relaxation compromised Inc-Powerlaw's linear running-time as it implied too frequent restarts. We solved this issue in consultation with the authors of [17] by adding new switchings to Phase 4 (t_a-, t_b-, and t_c-switchings, see Section 5.2.6) and Phase 5 (switchings where $\max(m_1, m_2, m_3) = 2$, see Section 5.2.7).

We engineer and optimize an Inc-Powerlaw implementation and discuss practical parallelization possibilities. In an empirical evaluation, we study our implementation's performance, provide evidence of its linear running-time, and compare the running-time with an implementation of the popular approximately uniform *Edge Switching* algorithm.

5.1.3 Preliminaries and Notation

For consistency, we use notation in accordance with prior descriptions of Pld and Inc-Powerlaw. A graph $G = (V, E)$ has n nodes $V = \{1, \ldots, n\}$ and $|E|$ edges. An edge connecting node i to itself is called a *loop* at i. Let $m_{i,j}$ denote the multiplicity of edge $e = \{i, j\}$ (often abbreviated as ij); for $m_{i,j} = 0, 1, 2, 3$ we refer to e as a *non-edge*, *single-edge*, *double-edge*, *triple-edge*, respectively, and for $m_{i,j} > 1$ as *multi-edge* (analogously for loops). A graph is *simple* if it only contains simple edges (i.e., no multi-edges or loops).

Given a graph G, define the *degree* $\deg(i) = 2m_{i,i} + \sum_{j \in V/\{i\}} m_{i,j}$ as the number of edges incident to node $i \in V$. Let $\mathcal{D} = (d_1, \ldots, d_n) \in \mathbb{N}^n$ be a *degree sequence* and denote $\mathcal{G}(\mathcal{D})$ as the set of simple graphs on n nodes with $\deg(i) = d_i$ for all $i \in V$. The degree sequence $\mathcal{D}$ is *graphical* if $\mathcal{G}(\mathcal{D})$ is non-empty. If not stated differently, $\mathcal{D}$ is non-increasing, i.e., $d_1 \geq d_2 \geq \ldots \geq d_n$. We say $\mathcal{D}$ is *power-law distribution-bounded (plib)* with exponent $\gamma > 1$ if $\mathcal{D}$ is strictly positive and there exists a constant K (independent of n) such that for all $i \geq 1$ there are at most $Kni^{1-\gamma}$ entries of value i or larger [78].

$\deg(\cdot)$: *degree*

$\mathcal{D}$: *degree sequence*

graphical

plib: power-law distribution-bounded

Denote the k-th factorial moment as $[x]_k = \prod_{i=0}^{k-1}(x-i)$ and define $M_k = \sum_{i=1}^{n}[d_i]_k$, $H_k = \sum_{i=1}^{h}[d_i]_k$, and $L_k = M_k - H_k$, where h is a parameter defined in Section 5.2.1 (roughly speaking, h is the number of nodes with high degrees).

5.2 Algorithm Description

Inc-Powerlaw takes a degree sequence $\mathcal{D} = (d_1, \dots, d_n)$ as input and outputs a uniformly random simple graph $G \in \mathcal{G}(\mathcal{D})$. The expected running-time is $\mathcal{O}(n)$ if $\mathcal{D}$ is a plib sequence with exponent $\gamma > 21/10 + \sqrt{61}/10 \approx 2.88102$.

The algorithm starts by generating a random graph G using the *Configuration Model* [26]. To this end, let G be a graph with n nodes and no edges, and for each node $i \in V$ place d_i marbles labeled i into an urn. We then draw two random marbles without replacement, connect the nodes indicated by their labels, and repeat until the urn is empty. The resulting graph G is uniformly distributed in the set $\mathcal{S}(\mathbf{m}_{h,l,d,t}(G))$ where $\mathbf{m}_{h,l,d,t}(G)$ is a vector specifying the multiplicities of all edges between, or loops at heavy nodes (as defined below), as well as the total numbers of other single-loops, double-edges, and triple-edges. In particular, if G is simple, then it is uniformly distributed in $\mathcal{G}(\mathcal{D})$. Moreover, if $\mathcal{D}$ implies $M_2 < M_1$, the degrees are rather small, and with constant probability G is a simple graph [104]. Hence, *rejection sampling* is efficient; the algorithm returns G if it is simple and restarts otherwise.

For $M_2 \geq M_1$, the algorithm goes through five *phases*. In each phase, all non-simple edges of one kind, e.g., all single-loops, or all double-edges, are removed from the graph by using *switchings*. A switching replaces some edges in the graph with other edges while preserving the degrees of all nodes. Phases 1 and 2 remove multi-edges and loops with high multiplicity on the highest-degree nodes. In Phases 3, 4 and 5, the remaining single-loops, triple-edges and double-edges are removed. To guarantee the uniformity of the output and the linear running-time, the algorithm may restart in some steps. A restart always resets the algorithm back to the first step of generating the initial graph.

Note that the same kind of switching can have different effects depending on which edges are selected for participation in the switching. In general, we only allow the algorithm to perform switchings that have the intended effect. Usually, a switching should remove exactly one non-simple edge without creating or removing other non-simple edges. A switching that has the intended effect is called *valid*.

Uniformity of the output is guaranteed by ensuring that the expected number of times a graph G in $\mathcal{S}(\mathbf{m}_{h,l,d,t}(G))$ is produced in the algorithm depends only on $\mathbf{m}_{h,l,d,t}(G)$. This requires some attention since, in general, the number of switchings we can perform on a graph and the number of switchings that produce a graph can vary between graphs in the same set (i.e., some graphs are more likely reached than others). To remedy this, there are *rejection steps*, which restart the algorithm with a certain probability. *Before* a switching is performed on a graph G, the algorithm accepts with a probability proportional to the number of valid switchings that can be performed on G, and *forward-rejects (f-rejects)* otherwise. We do this by selecting an uniform random switching on G, and accepting if it is valid, or rejecting otherwise. Then, *after* a switching produced a graph G', the algorithm accepts with a probability inversely proportional to the number of valid switchings that can produce G', and *backward-rejects (b-rejects)* otherwise. This is done by computing a quantity $b(G')$ that is proportional to the number of valid switchings that can produce G', and a lower bound $\underline{b}(G')$ on $b(G')$ over all G' in the same set, and then accepting with probability $\underline{b}(G')/b(G')$.

5.2.1 Phase 1 and 2 Preconditions

In Phases 1 and 2, the algorithm removes non-simple edges with high multiplicity between the highest-degree nodes. To this end, define a parameter $h = n^{1-\delta(\gamma-1)}$ where δ is chosen so that $1/(2\gamma-3) < \delta < (2-3/(\gamma-1))/(4-\gamma)$ (e.g. $\delta \approx 0.362$ for $\gamma \approx 2.88103$). The h highest-degree nodes are then called *heavy*, and the remaining nodes are called *light*. An edge is called *heavy* if its incident nodes are heavy, and *light* otherwise. A *heavy multi-edge* is a multi-edge between heavy nodes, and a *heavy loop* is a loop at a heavy node.

Now, let W_i denote the sum of the multiplicities of all heavy multi-edges incident with i, and let $W_{i,j} = W_i + 2m_{i,i} - m_{i,j}$. Finally, let $\eta = \sqrt{M_2^2 H_1 / M_1^3}$. There are four preconditions for Phase 1 and 2: (1) for all nodes $i \neq j$ connected by a heavy multi-edge, we have $m_{i,j}W_{i,j} \leq \eta d_i$ and $m_{i,j}W_{j,i} \leq \eta d_j$, (2) for all nodes i that have a heavy loop, we have $m_{i,i}W_i \leq \eta d_i$, (3) the sum of the multiplicities of all heavy multi-edges is at most $4M_2^2/M_1^2$, and (4) the sum of the multiplicities of all heavy loops is at most $4M_2/M_1$.

If any of the preconditions is not met, the algorithm restarts, otherwise it enters Phase 1.

5.2.2 Phase 1: Removal of Heavy Multi-Edges

A heavy multi-edge ij with multiplicity $m = m_{i,j}$ is removed with the *heavy-m-way switching* shown in Figure 5.1. Note that the switching is defined on *pairs* instead of edges. An edge ij of multiplicity m is treated as m distinct pairs (i,j). Adding a pair (i,j) increases the multiplicity m, and similarly, removing (i,j) decreases m. The heavy-m-way switching switching removes the m pairs (i,j) and m additional pairs (v_k, v_{k+1}), $1 \leq k \leq m$, and replaces them with $2m$ new pairs between i and v_k, and j and v_{k+1}.

In Phase 1, we iterate over all heavy multi-edges ij and each time execute:

1. Pick a uniform random heavy-m-way switching $S = (G, G')$ at nodes i and j as follows: for all $1 \leq k \leq m$, sample a uniform random pair (v_k, v_{k+1}) in random orientation. Then remove the pairs (i,j) and (v_k, v_{k+1}), and add (i, v_k) and (j, v_{k+1}). The graph that results after all pairs have been switched is G'.

2. Restart the algorithm (f-reject) if S is not valid. The switching is valid if for all $1 \leq k \leq m$: (a) v_k and v_{k+1} are distinct from i and j, (b) if v_k is heavy, it is not already connected to i, and if v_{k+1} is heavy, it is not connected to j, and (c) at

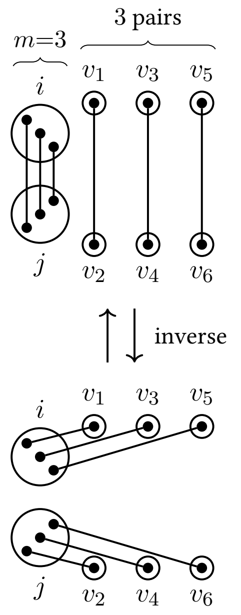


Figure 5.1: *A heavy-m-way switching where $m = 3$.*

least one of v_k and v_{k+1} is light. (This ensures that only the heavy multi-edge ij is removed, and no other heavy multi-edges or loops are added or removed.)

3. Restart the algorithm (b-reject) with probability $1 - \dfrac{\underline{b}_{hm}(G', i, j, m)}{b_{hm}(G', i, j, m)}$.
4. Set $G \leftarrow G'$ and continue to the next iteration with probability $\dfrac{1}{1 + \dfrac{\overline{b}_{hm}(G', i, j, 1)}{\underline{f}_{hm}(G', i, j, 1)}}$.

   Otherwise, re-add ij as a single-edge with the following steps:

   (a) Pick a uniform random inverse heavy-1-way switching $S' = (G', G'')$ at nodes i, j as follows: pick one simple neighbor v_1 of i (i.e., edge $v_1 i$ is simple) uniformly at random, and analogously v_2 for j. Then remove the pairs (i, v_1), (j, v_2), and add (i, j), (v_1, v_2).

   (b) Restart the algorithm (f-reject) if S' is not valid. The switching is valid unless both v_1 and v_2 are heavy.

   (c) Restart the algorithm (b-reject) with probability $1 - \dfrac{\underline{f}_{hm}(G'', i, j, 1)}{f_{hm}(G'', i, j, 1)}$.

   (d) Set $G \leftarrow G''$.

To compute the b-rejection probability for step 3, let Y_1 and Y_2 be the number of heavy nodes that are neighbors of i and j, respectively, in the graph G'. Then, set:

$$\begin{aligned}\underline{b}_{hm}(G', i, j, m) = {} & [d_i - W_{i,j}]_m [d_j - W_{j,i}]_m \\ & - mh^2 [d_i - W_{i,j}]_{m-1} [d_j - W_{j,i}]_{m-1}\end{aligned} \tag{5.1}$$

$$\begin{aligned}b_{hm}(G', i, j, m) = \sum_{l=0}^{m} \Bigg[& (-1)^l \binom{m}{l} [Y_1]_l [Y_2]_l \\ & \cdot [d_i - W_{i,j} - l]_{m-l} [d_j - W_{j,i} - l]_{m-l} \Bigg]\end{aligned} \tag{5.2}$$

For step 4:

$$\overline{b}_{hm}(G', i, j, 1) = (d_i - W_{i,j})(d_j - W_{j,i}) \tag{5.3}$$

$$\underline{f}_{hm}(G', i, j, 1) = M_1 - 2H_1 \tag{5.4}$$

For step 4c: let Z_1 be the number of ordered pairs between light nodes in the graph G'', let Z_2 be the number of pairs between one light and one heavy node, where the heavy node is not adjacent to i, and let Z_3 be the analogous number for j. Then set:

$$f_{hm}(G'', i, j, 1) = Z_1 + Z_2 + Z_3 \tag{5.5}$$

$$\tag{5.6}$$

$$\underline{f}_{hm}(G'', i, j, 1) = M_1 - 2H_1 \tag{5.7}$$

Phase 1 ends if all heavy multi-edges are removed. Then Inc-Powerlaw enters Phase 2.

5.2.3 Phase 2: Removal of Heavy Loops

Phase 2 removes all heavy loops using the *heavy-m-way loop switching* shown in Figure 5.2. The algorithm iterates over all heavy nodes i that have a heavy loop, and for each performs the following steps:

1. Pick a uniform random heavy-m-way loop switching $S = (G, G')$ at node i (cf. Phase 1).
2. Restart (f-reject) if S is not valid. The switching is valid if for all $1 \leq k \leq m$: a) $v_k \neq i$ and $v_{k+1} \neq i$, b) iv_k and iv_{k+1} are non-edges or light, c) at least one of v_k and v_{k+1} is light.
3. Restart the algorithm (b-reject) with probability $1 - \dfrac{\underline{b}_{hl}(G', i, m)}{b_{hl}(G', i, m)}$.
4. Set $G \leftarrow G'$.

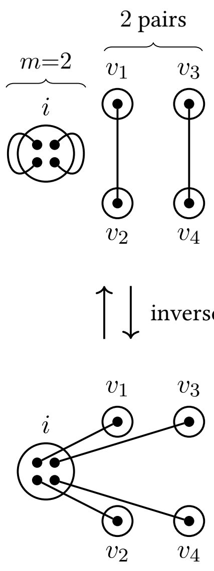


Figure 5.2: *A heavy-m-way loop switching where $m = 2$.*

Let Y be the number of heavy neighbors of i in G'. The quantities needed in step 3 are:

$$\underline{b}_{hl}(G', i, m) = [d_i]_{2m} - mh^2[d_i]_{2m-2} \tag{5.8}$$

$$b_{hl}(G', i, m) = \sum_{l=0}^{m} (-1)^l \binom{m}{l} [Y]_{2l} [d_i - 2l]_{2m-2l} \tag{5.9}$$

Phase 2 ends if all heavy loops are removed. We then check preconditions for the next phases.

5.2.4 Phase 3, 4 and 5 Preconditions

After Phases 1 and 2, the only remaining non-simple edges in the graph G are all incident with at least one light node, i.e., with one of the low-degree nodes. With constant probability, the only remaining non-simple edges are single loops, double-edges, and triple-edges, and there are not too many of them [78]. Otherwise, the algorithm restarts. Let m_l denote the number of single loops, m_t the number of triple-edges, and m_d the number of double-edges in the graph G. Then, the preconditions are: (1) $m_l \leq 4L_2/M_1$, (2) $m_t \leq 2L_3M_3/M_1^3$, (3) $m_d \leq 4L_2M_2/M_1^2$, and (4) there are no loops or multi-edges of higher multiplicity.

If all preconditions are met, the algorithm enters Phase 3 to remove all remaining loops.

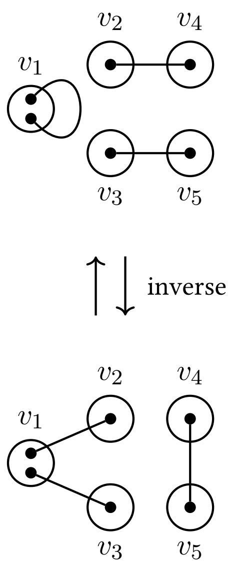


Figure 5.3: *The l-switching used in Phase 3.*

5.2.5 Phase 3: Removal of Light Loops

Phase 3 removes all light loops, i.e., loops at lower degree nodes, with the *l-switching* depicted in Figure 5.3. We repeat the following steps until all loops are removed:

1. Pick a uniform random l-switching $S = (G, G')$ as follows. Sample a uniform random loop on some node v_1 in G. Then, sample two uniform random pairs

(v_2, v_4) and (v_3, v_5) in random orientation. Replace (v_1, v_1), (v_2, v_4), (v_3, v_5) with (v_1, v_2), (v_1, v_3), (v_4, v_5).

2. Restart (f-reject) if S is not valid. The switching is valid if it removes the targeted loop without adding or removing other multi-edges or loops.
3. Restart the algorithm (b-reject) with probability $1 - \dfrac{\underline{b}_l(G'; 0)\underline{b}_l(G'; 1)}{b_l(G', \emptyset)b_l(G', v_1v_2v_3)}$.
4. Set $G \leftarrow G'$.

To accelerate the computation of the b-rejection probabilities, *incremental relaxation* [17] is used. Let $v_1v_2v_3$ denote a *two-star* centered at v_1, i.e. three nodes v_1, v_2, v_3 where v_1v_2 and v_1v_3 are edges. We call a two-star $v_1v_2v_3$ *simple*, if both edges are simple, and we call the star *light*, if the center v_1 is a light node. Finally, we speak of *ordered* two-stars if each permutation of the labels for the outer nodes v_2 and v_3 implies a distinct two-star. Then, the l-switching creates a light simple two-star $v_1v_2v_3$ and a simple pair v_4v_5.

With incremental relaxation, the b-rejection is split up into two sub-rejections, one for each structure created by the switching. First, set $b_l(G', \emptyset)$ to the number of light simple ordered two-stars in G'. Then, initialize $b_l(G', v_1v_2v_3)$ to the number of simple ordered pairs in G'. Now, subtract all the simple ordered pairs that are incompatible with the two-star $v_1v_2v_3$ created by the switching. The incompatible pairs a) share nodes with the two-star $v_1v_2v_3$ or b) have edges v_2v_4 or v_3v_5. Let $A_2 = \sum_{i=1}^{d_1} d_i$. Then, we use the following lower bounds on these quantities:

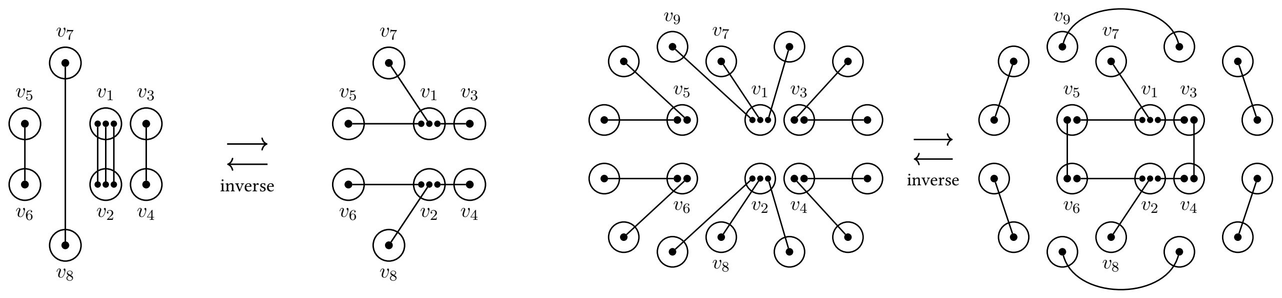
$$\underline{b}_l(G'; 0) = L_2 - 12m_td_h - 8m_dd_h - m_ld_h^2 \tag{5.10}$$

$$\underline{b}_l(G'; 1) = M_1 - 6m_t - 4m_d - 2m_l - 2A_2 - 4d_1 - 2d_h \tag{5.11}$$

Next, the algorithm removes triple-edges in Phase 4.

5.2.6 Phase 4: Removal of Light Triple-Edges

In Phase 4, the algorithm uses multiple different switchings. Similarly to the previous phases, there is one switching that removes the multi-edges. The other switchings, called *boosters*, lower the probability of a b-rejection. In total, there are four different switchings. The t-switching removes a triple-edge (see Figure 5.4a). The t_a-, t_b- and t_c-switchings create structures consisting of a simple three-star $v_1v_3v_5v_7$, and a light simple three star $v_2v_4v_6v_8$, that do not share any nodes. We call these structures *triplets*. Note that the t-switching creates a triplet where none of the edges v_1v_2, v_3v_4, v_5v_6 or v_7v_8 are allowed to exist. The t_a-switching creates the triplets where either one of the edges v_3v_4, v_5v_6 or v_7v_8 exist. The t_b-switching (see Figure 5.4b) creates the triplets where two of those edges exist. The t_c-switching creates the triplet where all three of those edges exist.



(a) The t-switching.

(b) The t_b-switching.

Figure 5.4: Switchings used in Phase 4. Both produce a triplet with the three-stars $v_1v_3v_5v_7$ and $v_2v_4v_6v_8$ centered at v_1 and v_2, respectively.

Phase 4 removes all triple-edges. In each iteration, the algorithm first chooses a switching type τ from $\{t, t_a, t_b, t_c\}$, where type τ has probability ρ_τ. The sum of these probabilities can be less than one, and the algorithm restarts with the remaining probability. Overall, we have the following steps (where the constants ρ_τ, defined below, ensure uniformity – cf. [77]):

1. Choose switching type τ with probability ρ_τ, or restart with probability $1-\sum_\tau \rho_\tau$.

2. Pick a uniform random τ-switching $S = (G, G')$. If $\tau = t$, sample a uniform random triple-edge and three uniform random pairs, and switch them as shown in Figure 5.4a. If $\tau \neq t$, sample some uniform random k-stars, as exemplified for t_b in Figure 5.4b, and switch them into the intended triplet.

3. Restart the algorithm (f-reject) if S is not valid. The switching is valid if, for $\tau = t$, it removes the targeted triple-edge, or if, for $\tau \neq t$, it creates the intended triplet, without adding or removing (other) multi-edges or loops.

4. Restart the algorithm (b-reject) with probability $1 - \dfrac{\underline{b}_t(G'; 0)\underline{b}_t(G'; 1)}{b_t(G', \emptyset)b_t(G', v_1v_3v_5v_7)}$.

5. Restart the algorithm (b-reject) with probability $1 - \dfrac{\underline{b}_\tau(G')}{b_\tau(G')}$.

6. Set $G \leftarrow G'$.

For the b-rejection, incremental relaxation is used. In step 4, there are two sub-rejections for the triplet, and in step 5, there are sub-rejections for any additional pairs created (e.g. pairs that are not part of the triplet). The t-switching creates no additional pairs, the t_a-switching creates three, the t_b-switching shown in Figure 5.4b creates six, and the t_c-switching nine.

The probability for step 4 is computed as follows: first, set $b_t(G', \emptyset)$ to the number of simple ordered three-stars in G'. Then, set $b_t(G', v_1v_3v_5v_7)$ to the number of light simple ordered three-stars that a) do not share any nodes with the three-star $v_1v_3v_5v_7$ created by S, b) have no edge v_1v_2 and no multi-edges v_3v_4, v_5v_6, v_7v_8. Let $B_k = \sum_{i=1}^{d_1}[d_{h+i}]_k$. Then, the lower bounds are:

$$\underline{b}_t(G';0) = M_3 - 18m_td_1^2 - 12m_dd_1^2 \tag{5.12}$$

$$\underline{b}_t(G';1) = L_3 - 18m_td_h^2 - 12m_dd_h^2 - B_3 - 3(m_t+m_d)B_2 - d_h^3 - 9B_2 \tag{5.13}$$

For step 5: let k be the number of additional pairs created by the switching. Then, for pair $1 \leq i \leq k$, set $b_\tau(G', \overline{V}_{i+1}(S))$ to the number of simple ordered pairs in G', that a) do not share nodes with the triplet or the previous $i-1$ pairs, and b) have no edges that should have been removed by the switching (e.g., in Figure 5.4b, v_1v_9 cannot be an edge). Finally, set $b_\tau(G') = \prod_{i=1}^{k} b_\tau(G', \overline{V}_{i+1}(S))$. The lower bound is:

$$\underline{b}_\tau(G') = \prod\nolimits_{i=1}^{k} \underline{b}_\tau(G';i+1) \tag{5.14}$$

$$\underline{b}_\tau(G';i+1) = M_1 - 6m_t - 4m_d - 16d_1 - 4(i-1)d_1 - 2A_2 \tag{5.15}$$

The type probabilities as computed as follows. When initializing Phase 4, set $\rho_t = 1-\varepsilon$ where $\varepsilon = 28M_2^2/M_1^3$, and set $\rho_\tau = 0$ for $\tau \in \{t_a, t_b, t_c\}$. In each subsequent iteration, the probabilities are only updated after a t-switching $S = (G, G')$ is performed. Then, first, let i be the number of triple-edges in the graph G', and let i_1 be the initial number of triple-edges after first entering Phase 4. Now, define a parameter x_i:

$$x_i = x_{i+1}\rho_t \frac{\underline{b}_t(G';0)\underline{b}_t(G';1)}{\overline{f}_t(i+1)} + 1, \tag{5.16}$$

where $x_{i_1} = 1$ and $\overline{f}_t(i) = 12iM_1^3$.

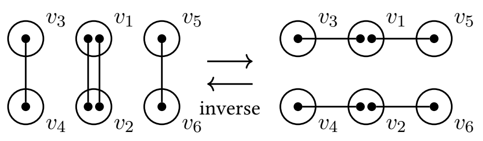
Define $\overline{f}_{t_a} = 3M_3L_3M_2^2$, $\overline{f}_{t_b} = 3M_3L_3M_2^4$, and $\overline{f}_{t_c} = M_3L_3M_2^6$. Then, update the probability ρ_τ for $\tau \in \{t_a, t_b, t_c\}$ as follows:

$$\rho_\tau = \frac{x_{i+1}}{x_i}\rho_t \frac{\overline{f}_\tau}{\underline{b}_\tau(G')\overline{f}_t(i+1)} \tag{5.17}$$

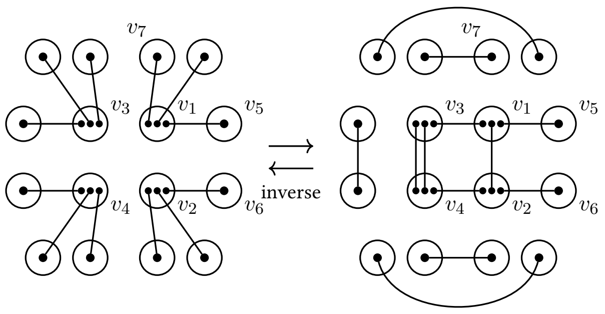
Finally, the algorithm enters Phase 5.

5.2.7 Phase 5: Removal of Light Double-Edges

Similar to Phase 4, Phase 5 uses multiple different switchings. The d-switching (see Figure 5.5a) removes double-edges. The booster switchings create so-called *doublets* consisting of a simple two-star $v_1v_3v_5$ and a light simple two-star $v_2v_4v_6$ that do not share any nodes. Let m_1, m_2, and m_3 denote the multiplicities of the edges v_1v_2, v_3v_4 and v_5v_6 in a doublet, respectively. Then, the d-switching creates the doublet with $\max(m_1, m_2, m_3) = 0$. The booster switchings create all other doublets where $\max(m_1, m_2, m_3) \leq 2$, i.e., where some of the edges are single-edges or double-edges. We identify each booster switching by the doublet created, e.g., type $\tau = (1, 2, 0)$ shown in Figure 5.5b creates a doublet where $m_1 = 1$, $m_2 = 2$, and $m_3 = 0$.



(a) d-switching.



(b) The type $\tau = (1, 2, 0)$ booster switching.

Figure 5.5: Switchings used in Phase 5.

Phase 5 repeats the following steps until all double-edges are removed:

1. Choose switching type τ with probability ρ_τ or restart with the probability $1 - \sum_\tau \rho_\tau$.
2. Pick a uniform random τ-switching $S = (G, G')$. If $\tau = d$, sample a uniform random double-edge and two uniform random pairs, and switch them as shown in Figure 5.5a. If $\tau \neq d$, sample a number of uniform random k-stars, as exemplified in Figure 5.5b, and switch them into the intended doublet and a number of additional simple edges.
3. Restart the algorithm (f-reject) if S is not valid. The switching is valid if, for $\tau = d$, it removes the targeted double-edge or, for $\tau \neq d$, it creates the intended doublet without adding or removing (other) multi-edges or loops.
4. Restart the algorithm (b-reject) with probability $1 - \dfrac{\underline{b}_d(G'; 0)\underline{b}_d(G'; 1)}{b_d(G', \emptyset)b_d(G', v_1v_3v_5)}$.
5. Restart the algorithm (b-reject) with probability $1 - \dfrac{\underline{b}_\tau(G')}{b_\tau(G')}$.
6. Set $G \leftarrow G'$.

For the b-rejection, incremental relaxation is used. Step 4 contains the sub-rejections for the doublet and step 5 the rejections for any additional pairs created by the switching. In general, the number of additional pairs created by type $\tau = (m_1, m_2, m_3)$ is $k = I_{m_1 \geq 1} m_1 + I_{m_2 \geq 1}(m_2 + 2) + I_{m_3 \geq 1}(m_3 + 2)$ where I denotes the indicator function.

For step 4: first, set $b_d(G', \emptyset)$ to the number of simple ordered two-stars in G'. Then, set $b_d(G', v_1v_3v_5)$ to the number of light simple ordered two-stars in G' that do not share any nodes with the two-star $v_1v_3v_5$ created by S. The lower bounds are:

$$\underline{b}_d(G'; 0) = M_2 - 8m_d d_1 \tag{5.18}$$

$$\underline{b}_d(G'; 1) = L_2 - 8m_d d_h - 6B_1 - 3d_h^2 \tag{5.19}$$

For step 5: let k be the number of additional pairs created by the switching. Then, for pair $1 \leq i \leq k$, set $b_\tau(G', \overline{V}_{i+1}(S))$ to the number of simple ordered pairs in G', that a) do not share nodes with the doublet or the previous $i-1$ pairs, and b) have no edges that should have been removed by the switching (e.g., in Figure 5.5b, v_1v_7 cannot be an edge). Finally, set $b_\tau(G') = \prod_{i=1}^{k} b_\tau(G', \overline{V}_{i+1}(S))$. The lower bound is:

$$\underline{b}_\tau(G') = \prod_{i=1}^{k} \underline{b}_\tau(G'; i+1) \tag{5.20}$$

$$\underline{b}_\tau(G'; i+1) = M_1 - 4m_d - 12d_1 - 4(i-1)d_1 - 2A_2 \tag{5.21}$$

The type probabilities are computed as follows. When initializing Phase 5, set $\rho_d = 1 - \xi$ where

$$\xi = \frac{32M_2^2}{M_1^3} + \frac{36M_4L_4}{M_2L_2M_1^2} + \frac{32M_3^2}{M_1^4}, \tag{5.22}$$

and set $\rho_\tau = 0$ for all types $\tau \neq d$. The probabilities are updated after a switching $S = (G, G')$ is performed that changes the number of double-edges. Then, first, let i denote the new number of double-edges in G', let i_1 denote the initial number of double-edges after first entering Phase 5, and let $\rho_d(i)$ denote the probability of type d on a graph with i double-edges. Now, define a parameter x_i:

$$x_i = x_{i+1}\rho_d(i+1)\frac{\underline{b}_d(G';0)\underline{b}_d(G';1)}{\overline{f}_d(i+1)} + 1, \tag{5.23}$$

where $x_{i_1} = 1$ and $\overline{f}_d(i) = 4iM_1^2$.

Then, to update the probability of a booster switching type τ, there are two cases: (1) if $\tau = (m_1, m_2, m_3)$ adds double-edges (i.e., $\max(m_1, m_2, m_3) = 2$) and the number of double-edges if a switching of this type was performed would be higher than $i_1 - 1$, set $\rho_\tau = 0$. Otherwise, (2) let i' denote the new number of double-edges if a switching of this type was performed, and define

$$\overline{f}_\tau = M_{k_1}L_{k_1}(I_{k_2\geq 2}M_{k_2}^2 + I_{k_2\leq 1}1) \cdot (I_{k_3\geq 2}M_{k_3}^2 + I_{k_3\leq 1}1), \tag{5.24}$$

where $k_1 = m_1 + 2$, $k_2 = m_2 + 1$, $k_3 = m_3 + 1$.

Then set:

$$\rho_\tau = \frac{x_{i'+1}}{x_i}\rho_{d_{i'+1}}\frac{\overline{f}_\tau}{\underline{b}_\tau(i')\overline{f}_d(i'+1)} \tag{5.25}$$

$$\rho_d = 1 - \rho_{1,0,0} - \xi. \tag{5.26}$$

When Phase 5 terminates, all non-simple edges are removed, and the final graph G is output.

5.3 Adjustments to the Algorithm

In this section, we describe our adjustments to the Inc-Powerlaw algorithm sketched in [17].

5.3.1 New Switchings in Phase 4

Phase 4 of Pld only uses the t-switching [78]. There, the rejection probability is small enough so that no booster switchings are needed. The overall running-time of Phase 4 in Pld however, is superlinear, as computing the probability of a b-rejection requires counting the number of valid t-switchings that produce a graph. By using incremental relaxation [17], the cost of computing the b-rejection probability becomes sublinear, as it only requires us to count simpler structures in the graph. However, when applying incremental relaxation to Phase 4, the probability of a b-rejection increases, as the lower bounds on the number of those structures are less tight, and the overall running-time remains superlinear.

We address this issue by using booster switchings in Phase 4 to reduce the b-rejection probability (analogous to Phase 5 of Pld). To this end, we add three new switchings: the t_a, t_b and t_c switching (see Section 5.2.6). This is done entirely analogous to Phase 5 of Pld, which also uses multiple switchings in the same Phase. We first derive an equation for the expected number of times that a graph is produced by a type $\tau \in \{t, t_a, t_b, t_c\}$ switching. Then we set equal the expected number of times for each graph in the same set $\mathcal{S}(\mathbf{m}_{h,l,d,t})$. The resulting system of equations is fully determined by choosing an upper bound ε on the probability of choosing a type $\tau \neq t$. We can then derive the correct probabilities ρ_τ for each type as a function of $\mathbf{m}_{h,l,d,t}$.

Lemma 5.1. Let $\mathcal{D}$ be a plib sequence with exponent $\gamma > 21/10 + \sqrt{61}/10 \approx 2.88102$. Then, given $\mathcal{D}$ as input, the probability of a b-rejection in Phase 4 of Inc-Powerlaw is $o(1)$. ◀

Proof. Refer to Section 5.A (Appendix) for the proof. □

5.3.2 New Switchings in Phase 5

Phase 5 of Pld uses the type-I switching (this is the same as the d-switching in Inc-Powerlaw), as well as a total of six booster switchings called type-III, type-IV, type-V, type-VI and type-VII. These booster switchings create the doublets where each of the "bad edges" can either be a non-edge or single-edge, i.e. $\max\{m_1, m_2, m_3\} = 1$. For Phase 5 of Pld, this suffices to ensure that the b-rejection probability is small enough. However, similar to Phase 4, applying incremental relaxation to reduce the computational cost increases the rejection probability, leading to a superlinear running-time overall.

To further reduce the probability of a b-rejection, we add booster switchings that create the doublets where one or more of the "bad edges" is a double-edge, i.e. $\max\{m_1, m_2, m_3\} = 2$ (see Section 5.2.7). We then integrate the new switchings to Phase 5 of Inc-Powerlaw by deriving the correct probabilities ρ_τ and increasing the constant ξ used to bound the probabilities of the types $\tau \notin \{d, (1,0,0)\}$.

Lemma 5.2. Let $\mathcal{D}$ be a plib sequence with exponent $\gamma > 21/10 + \sqrt{61}/10 \approx 2.88102$. Then, given $\mathcal{D}$ as input, the probability of a b-rejection in Phase 5 of Inc-Powerlaw is $o(1)$. ◀

Proof. Refer to Section 5.A (Appendix) for the proof. □

5.3.3 Expected Running-time

We now use Lemma 5.1 and Lemma 5.2 to bound the expected running-time of the algorithm.

Theorem 5.3. Let $\mathcal{D}$ be a plib sequence with exponent $\gamma > 21/10+\sqrt{61}/10 \approx 2.88102$. Then, given $\mathcal{D}$ as input, the expected running-time of Inc-Powerlaw is $\mathcal{O}(n)$. ◀

Proof. From [17], we know that the running-time of each individual phase (e.g. computation of rejection parameters, etc.) is at most $\mathcal{O}(n)$ and in addition, we know that the probability of an f-rejection in any Phase is $o(1)$ and the probability of a b-rejection in Phases 1, 2 or 3 is $o(1)$. By Lemma 5.1 and Lemma 5.2, the probability of a b-rejection in Phase 4 or 5 is $o(1)$. Therefore, the expected number of restarts is $\mathcal{O}(1)$, and the overall running-time of Inc-Powerlaw is $\mathcal{O}(n)$. □

5.4 Implementation

In this section we highlight some aspects of our Inc-Powerlaw implementation. The generator is implemented in modern C++ and relies on Boost Multiprecision[2] to handle large integer and rational numbers that occur even for relatively small inputs.

5.4.1 Graph Representation

Inc-Powerlaw requires a dynamic graph representation capable of adding and removing edges, answering edge existence and edge multiplicity queries, enumerating a node's neighborhood, and sampling edges weighted by their multiplicity. A careful combination of an adjacency vector and a hash map yields expected constant work for all operations.

In practice, however, we find that building and maintaining these structures is more expensive than using a simpler, asymptotically sub-optimal, approach. To this end, we exploit the small and asymptotically constant average degree of plib degree sequences and the fact that most queries do not modify the data structure. This allows us to only use a compressed sparse row (CSR) representation that places all neighborhoods in a contiguous array A_C and keep the start indices A_I of each neighborhood in a separate array; neighborhoods are maintained in sorted order and neighbors may appear multiple times to encode multi-edges.

Given an edge list, we can construct a CSR in time $\mathcal{O}(n+m)$ using integer sorting. A subsequent insertion or deletion of edge uv requires time $\mathcal{O}(\deg(u)+\deg(v))$; these operations, however, occur at a low rate. Edge existence and edge multiplicity queries for edge uv are possible in time $\mathcal{O}(\log\min(\deg(u),\deg(v)))$ by considering the node with the smaller neighborhood (as $\mathcal{D}$ is ordered $u \leq v$ implies $\deg(u) \geq \deg(v)$). Assuming plib degrees, these operations require constant expected work. Randomly drawing an

[2]https://github.com/boostorg/multiprecision (V 1.76.0)

edge weighted by its multiplicity is implemented by drawing a uniform index j for A_C and searching its incident node in A_I in time $\mathcal{O}(\log n)$[3].

Several auxiliary structures (e.g., indices to non-simple edges, numbers of several sub-structures, et cetera) are maintained requiring $\mathcal{O}(m)$ work in total. Where possible, we delay their construction to the beginning of Phase 3 in order to not waste work in case of an early rejection in Phases 1 or 2.

5.4.2 Parallelism

While Inc-Powerlaw seems inherently sequential (e.g. due to the dependence of each switching step on the previous steps), it is possible to parallelize aspects of the algorithm. In the following we sketch two non-exclusive strategies. These approaches are practically significant, but are not designed to yield a sub-linear time complexity.

Intra-Run

As we discuss in Section 5.5, the implementation's runtime is dominated by the sampling of the initial multigraph and construction of the CSR. These in turn spend most time with random shuffling and sorting. Both can be parallelized [18, 159].

Inter-Run

If Inc-Powerlaw restarts, the following attempt is independent of the rejected one. Thus, we can execute P instances of Inc-Powerlaw in parallel and return the "first" accepted result. Synchronization is only used to avoid a selection bias towards quicker runs: all processors assign globally unique indices to their runs and update them after each restart. We return the accepted result with smallest index and terminate processes working on results with larger indices prematurely. The resulting speed-up is bounded by the number of restarts which is typically a small constant.

5.4.3 Configuration Model

As the majority of time is spend sampling the initial graph G and building its CSR representation, we carefully optimize this part of our implementation. First, we give an extended description of the configuration model that remains functionally equivalent to Section 5.2.

Given a degree-sequence $\mathcal{D} = (d_1, \ldots, d_n)$, let G be a graph with n nodes and no edges. For each node $u \in V$ place d_u marbles labeled u into an urn. Then, randomly draw without replacement two marbles with labels a and b, respectively. Append label a to an initially empty sequence A and analogously label b to B. Finally, add for each $1 \leq i \leq m$ the edge $\{A[i], B[i]\}$ to G.

We adopt a common strategy [150] to implement sampling without replacement. First produce a sequence $N[1 \ldots 2m]$ representing the urn, i.e., the value i is contained

[3]Observe that constant time look-ups are straight-forward by augmenting each entry in A_C with the neighbor. We, however, found the contribution of the binary search non-substantial.

d_i times. Then randomly shuffle N and call the result N'. Finally, partition N' arbitrarily to obtain the aforementioned sequences A and B of equal sizes. For our purpose, it is convenient to choose the first and second halves of N', i.e., $A = N'[1 \dots m]$ and $B = N'[m{+}1 \dots 2m]$. [4]

Our parallel implementation shuffles N with a shared memory implementation based on [159]. We then construct a list of all pairs in both orientations and sort it lexicographically in parallel [18]. In the resulting sequence, each neighborhood is a contiguous subsequence. Hence, we can assign the parallel workers to independent subproblems by aligning them to the neighborhood boundaries. The sequential algorithm follows the same framework to improve locality of reference in the data accesses. It uses a highly tuned Fisher-Yates shuffle based on [118] and the integer sorting SkaSort[5].

Both shuffling algorithms are modified almost halving their work. The key insight is that the distribution of graphs sampled remains unchanged if we only shuffle A and allow an arbitrary permutation of B (or vice versa). This can be seen as follows. Assume we sampled A and B as before and computed graph $G_{A,B}$. Then, we let an adversary choose an arbitrary permutation π_B of B without knowing A. If we apply π_B to B before adding the edges, the resulting $G_{A,\pi_B(B)}$ is in general different from G_{AB}. We claim, however, that $G_{A,B}$ and $G_{A,\pi_B(B)}$ both are equally distributed samples of the configuration model. We can recover the original graph by also applying π_B to A, i.e., $G_{\pi_B(A),\pi_B(B)} = G_{A,B}$. Let P_m denote the set of all $m!$ permutations of a sequence of length m, and note that the composition $\circ \colon P_m \times P_m \to P_m$ is a bijection. Further recall that A is randomly shuffled and all its permutations $\pi_A \in P_m$ occur with equal probability. Thus, as π_A is uniformly drawn from P_m, so is $(\pi_A \circ \pi_B) \in P_m$. In conclusion, the distribution of edges is independent of the adversary's choice.

To exploit this observation, we partition N into two subsequences $N'[1 \dots k]$ and $N'[k+1 \dots 2m]$. Each element is assigned to one subsequence using an independent and fair coin flip. While partitioning and shuffling are both linear time tasks, in practice, the former can be solved significantly faster (in the parallel algorithm [159], it is even a by-product of the assignment of subproblems to workers). Observe that with high probability both sequences have roughly equal size, i.e., $|k - m| = \mathcal{O}(\sqrt{m})$. We then only shuffle the larger one (arbitrary tie-breaking if $k = m$), and finally output the pairs $(N'[i], N'[m+i])$ for all $1 \le i \le m$.

5.5 Empirical Evaluation

In the following, we empirically investigate our implementation of Inc-Powerlaw.

To reaffirm the correctness of our implementation and empirically support the uniformity of the sampled graphs, we used unit tests and statistical tests. For instance, we carried out χ^2-tests over the distribution of all possible graphs for dedicated small degree sequences $\mathcal{D}$ where it is feasible to fully enumerate $\mathcal{G}(\mathcal{D})$. Additionally, we assert

[4] To "shuffle" or "random permute" refers to the process of randomly reordering a sequence such that any permutation occurs with equal probability.

[5] https://github.com/skarupke/ska_sort

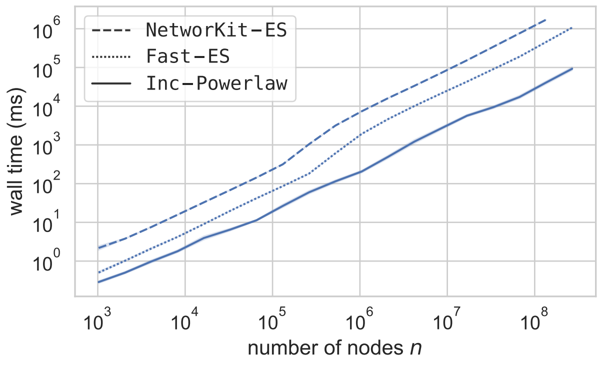


Figure 5.6: Average running-time of sequential Inc-Powerlaw and Edge-Switching.

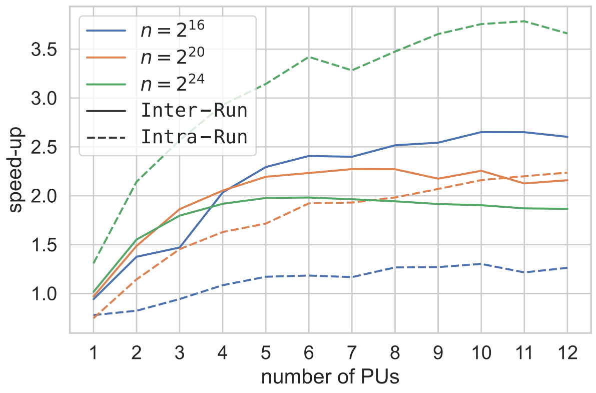


Figure 5.7: Speed-up of Inter-Run and Intra-Run.

the plausibility of rejection parameters.

The widely accepted, yet approximate, *Edge Switching* MCMC algorithm provides a reference to existing solutions. We consider two implementations: NetworKit-ES, included in *NetworKit* [171] and based on [81], was selected for its readily availability and flexibility. Fast-ES is our own solution that is at least as fast as all open sequential implementations we are aware of. For the latter, we even exclude the set-up time for the graph data structure. To their advantage, we execute an average of 10 switches per edge (in practice, common choices [137, 81, 154] are 10 to 30). Increasing this number improves the approximation of a uniform distribution, but linearly increases the work.

In each experiment below, we generate between 100 and 1000 random power-law degree sequences with fixed parameters n, γ, and minimal degree d_{min} analogously to the PowerlawDegreeSequence generator of *NetworKit*. Then, for each sequence, we benchmark the time it takes for the implementations to generate a graph and report their average. In the plots, a shaded area indicates the 95%-confidence interval. The benchmarks are built with GNU g++-9.3 and executed on a machine equipped with an AMD EPYC 7452 (32 cores) processor and 128 GB RAM running Ubuntu 20.04.

Running-time Scaling in n

In Figure 5.6 we report the performance of Inc-Powerlaw and the *Edge Switching* implementations for degree sequences with $\gamma \approx 2.88$, $d_{min} = 1$, and $n = 2^k$ for integer values $10 \leq k \leq 28$. Our Inc-Powerlaw implementation generates a graph with $n \approx 10^6$ nodes in 0.26 seconds. The plot also gives evidence towards Inc-Powerlaw's linear work complexity. Comparing with the Edge-Switching implementations, we find that Inc-Powerlaw runs faster. We can conclude that in this setting, the provably uniform Inc-Powerlaw runs just as fast, if not faster, than the approximate solution.

Table 5.1: *The average number of runs until acceptance and peak speed-ups as observed in Figure 5.7.*

n	runs	Inter-Run	Intra-Run
2^{16}	3.9	2.7 for $p = 10$	1.3 for $p = 10$
2^{20}	3.3	2.3 for $p = 7$	2.2 for $p = 12$
2^{24}	3.0	2.0 for $p = 5$	3.8 for $p = 11$

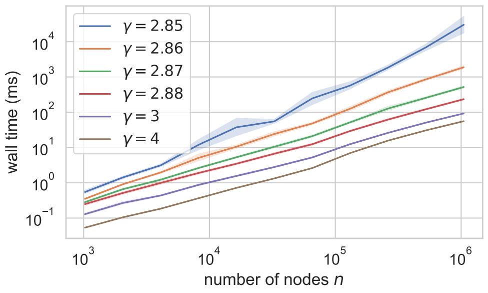


Figure 5.8: The average running-time of Inc-Powerlaw in dependence of n for different values of γ.

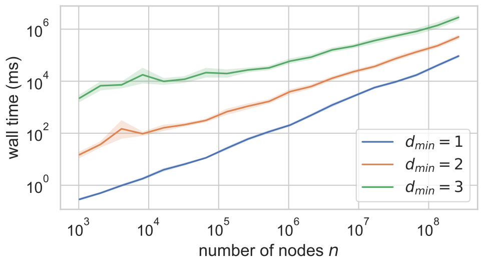


Figure 5.9: The average running-time of Inc-Powerlaw for sequences with minimum degree $d_{min} \in \{1, 2, 3\}$.

Speed-up of the Parallel Variants

Figure 5.7 shows the speed-up of our Inter-Run and Intra-Run parallelizations over sequential Inc-Powerlaw. We generate degree sequences with $n = 2^k$ for $k \in \{16, 20, 24\}$, measure the average running-time of the parallel variants when using $1 \leq p \leq 12$ PUs (processor cores), and report the speed-up in the average running-time of the parallel variants over sequential Inc-Powerlaw.

For $p = 5$ and $n = 2^{24}$, we observe an Inter-Run parallelization speed-up of 2.0; more PUs yield diminishing returns as the speed-up is limited by the number of runs until a graph is accepted which is 3.0 on average for the aforementioned parameters. Another limiting factor is the fact that rejected runs stop prematurely. Hence, the accepting run (i) requires on average more work and (ii) forms the critical path that cannot be accelerated by Inter-Run.

For the same n, Intra-Run achieves a speed-up of 3.8 for $p = 11$ PUs; here, the remaining unparallelized sections limit the scalability as governed by Amdahl's law [156]. Overall, Inter-Run yields a better speed-up if the the number of restarts is high (smaller n), whereas Intra-Run yields a better speed-up for larger n if the overall running-time is dominated by generating the initial graph (see Table 5.1).

Different Values of the Power-law Exponent γ

Next, we investigate the influence of the power-law exponent γ. The guarantees on Inc-Powerlaw's running-time only hold for sequences with $\gamma \gtrsim 2.88102$, so we expect a superlinear running-time for $\gamma \leq 2.88$. For $\gamma \geq 3$, the expected number of non-simple edges in the initial graph is much lower, so we expect the running-time to remain linear but with decreased constants. Figure 5.8 shows the average running-time of Inc-Powerlaw for sequences for various γ.

For $\gamma \leq 2.88$, we observe an increase in the running-time. The slope of the curve for

$\gamma = 2.85$ also suggests that the running-time becomes non-linear for lower values of γ. Overall, the requirement of $\gamma \gtrapprox 2.88102$ appears to be relatively strict. In particular, we observe that the higher maximum degrees of sequences with $\gamma \leq 2.85$ greatly increase the rejection probability in Phases 1 and 2.

For $\gamma \geq 3$, the average running-time decreases somewhat but remains linear. For these values of γ, we observe that the initial number of non-simple edges in the graph is small, and that the algorithm almost always accepts a graph on its first run, so the overall running-time approaches the time required to sample the initial graph with the configuration model.

Higher Average Degrees

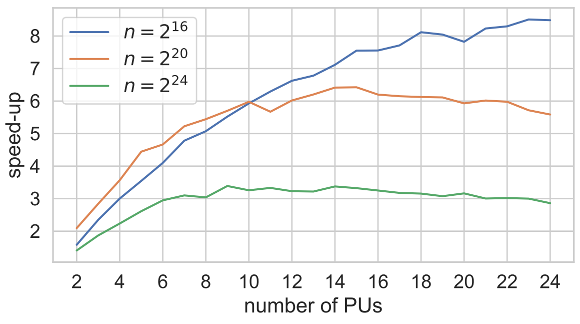
The previously considered sequences drawn from an unscaled power-law distribution tend to have a rather small average degree of approximately 1.44. On the other hand, many observed networks feature higher average degrees [22, 157]. To study Inc-Powerlaw on such networks, we sample degree sequences with minimum degree $d_{min} \in \{1, 2, 3\}$. For $d_{min} = 2$ and $d_{min} = 3$, the average degree $\overline{d}$ of the sequences increases to $\overline{d} = 3.39$ and $\overline{d} = 5.44$ respectively. We then let the implementation generate graphs for each choice of $d_{min} \in \{1, 2, 3\}$, and report the average time as a function of n in Figure 5.9.

As a higher average degree increases the expected number of non-simple edges in the initial graph, we observe a significant increase in running-time. For instance, for $n = 2^{20}$ we find that the average number of double-edges in the initial graph are 6.5, 41.6 and 98.8 for $d_{min} = 1, 2$ and 3, respectively, and the overall number of switching steps until a simple graph is obtained increases from 10.2 for $d_{min} = 1$ to 49.6 for $d_{min} = 2$ and to 110.8 for $d_{min} = 3$. This in turn greatly increases the chance for a rejection to occur and the number of runs until a graph is accepted (see Section 5.6).

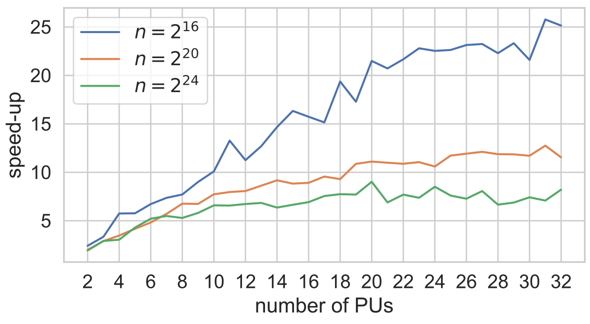
However, for large values of $n \geq 2^{24}$ the effect of the higher average degrees on the running-time becomes less pronounced. This is because the probability of a rejection at any step in the algorithm decreases quite fast with n, thus even if the number of switching steps increases, the number of runs decreases. We can conclude that Inc-Powerlaw is efficient when generating graphs that are either very sparse ($\overline{d} \lessapprox 5$) or very large ($n \gtrapprox 2^{24}$), but the algorithm is much less efficient when generating small to medium sized graphs ($n \lessapprox 2^{24}$) with medium average degree ($\overline{d} \gtrapprox 5$).

Speed-up of Inter-Run for Higher Average Degrees

While Inc-Powerlaw's sequential work increases with a higher average degree, so do the number of independent runs that can be parallelized by Inter-Run. Figure 5.10 shows the speedup of Inter-Run over sequential Inc-Powerlaw for sequences with $d_{min} = 2$ when using $2 \leq p \leq 24$ PUs and $d_{min} = 3$ using $2 \leq p \leq 32$ PUs. For $n = 2^{20}$ nodes, Inter-Run yields a speed-up of 6.4 with $p = 14$ PUs for $d_{min} = 2$ and 12.8 for $p = 31$ PUs for $d_{min} = 3$ (see Section 5.6).



(a) $d_{min} = 2$.



(b) $d_{min} = 3$.

Figure 5.10: Speed-up of Inter-Run for $d_{min} \in \{2, 3\}$.

Table 5.2: Average number of runs until acceptance and average number of switching steps in an accepting run.

	$d_{min} = 1$		$d_{min} = 2$		$d_{min} = 3$	
n	runs	steps	runs	steps	runs	steps
2^{16}	3.9	4.9	53.0	24.1	1160.7	51.0
2^{20}	3.3	10.2	18.1	49.6	164.0	110.8
2^{24}	3.0	19.0	9.6	92.8	43.3	208.5
2^{28}	2.5	32.3	5.6	161.6	16.7	375.3

Table 5.3: Average number of runs until acceptance and peak speed-ups as observed in Figure 5.10.

	$d_{min} = 2$		$d_{min} = 3$	
n	runs	speedup	runs	speedup
2^{16}	53.0	8.5 for $p = 24$	1160.7	25.8 for $p = 32$
2^{20}	18.1	6.4 for $p = 14$	164.0	12.8 for $p = 31$
2^{24}	9.6	3.4 for $p = 10$	43.3	9.0 for $p = 20$

As expected, we can achieve a higher speed-up for higher d_{min}, so we can partially mitigate the increase in running-time by taking advantage of the higher parallelizability. On the other hand, we still experience the limited scaling due to accepting runs being slower than rejecting runs.

5.6 Conclusions

For the first time, we provide a complete description of Inc-Powerlaw which builds on and extends previously known results [17, 78]. To the best of our knowledge, Inc-Powerlaw is the first practical implementation to sample provably uniform graphs from prescribed power-law-bounded degree sequences with $\gamma \geq 2.88102$. Our implementation is freely available.

In an empirical study, we find that Inc-Powerlaw is very efficient for small average degrees; for larger average degrees, we observe significantly increased constants in Inc-Powerlaw's running-time which are partially mitigated by an improved parallelizability.

While the expected running-time of Inc-Powerlaw is asymptotically optimal, we expect practical improvements for higher average degrees by improving the acceptance probability in Phases 3, 4 and 5 of the algorithm (e.g., by finding tighter lower bounds or by adding new switchings). It is also possible that the requirement on γ could be lowered; our experiments indicate that the acceptance probability in Phases 1 and 2 should be improved to this end. Our measurements also suggest that more fine-grained parallelism may be necessary to accelerate accepting runs.

Acknowledgments

This work was partially supported by the Deutsche Forschungsgemeinschaft (DFG) under the grants ME 2088/4-2 (SPP 1736 Algorithms for BIG DATA) and ME 2088/5-1 (FOR 2975 Algorithms, Dynamics and Information Flow in Networks). It was also supported in part by the Australian Research Council under grant DP160100835.

Appendix 5.A Proofs

Lemma 5.1. Let $\mathcal{D}$ be a plib sequence with exponent $\gamma > 21/10 + \sqrt{61}/10 \approx 2.88102$. Then, given $\mathcal{D}$ as input, the probability of a b-rejection in Phase 4 of Inc-Powerlaw is $o(1)$. ◀

Proof. We first show that the number of iterations in Phase 4 is at most $\mathcal{O}\big(L_3 M_3 / M_1^3\big)$. First, recall that the algorithm only enters Phase 4 if the graph satisfies the Phase 3, 4 and 5 preconditions. In particular, the graph may contain at most $L_3 M_3 / M_1^3$ triple-edges. Phase 4 terminates once all triple-edges are removed. In each iteration, a triple-edge is removed if we choose a type t-switching. The probability of choosing a t-switching is $\rho_t = 1 - \varepsilon$, where $\varepsilon = 28 M_2^2 / M_1^3$, and it can be verified that the probability of choosing any other switching is bounded by ε. In addition, we know that $M_k = \mathcal{O}\big(n^{k/\gamma - 1}\big)$ for $k \geq 2$ and $M_1 = \Theta(n)$ [78], and we have $\varepsilon = \mathcal{O}\big(n^{2/(\gamma-1)}/n^3\big) = o(1)$ assuming that $\gamma > 21/10 + \sqrt{61}/10 \approx 2.88102$. Thus, a triple-edge is removed in each iteration with probability $1 - o(1)$, and as the graph may contain at most $L_3 M_3 / M_1^3$ triple-edges, the total number of iterations is at most $\mathcal{O}\big(L_3 M_3 / M_1^3\big)$.

Now, we show that the probability of a b-rejection vanishes with n. First, it is easy to verify that the probability of a b-rejection in step 4 dominates the probability of a rejection in step 5 (compare Section 5.2.6). The probability of a b-rejection in step 4 is $1 - \underline{b}_t(G';0)\underline{b}_t(G';1)/(b_t(G',\emptyset)b_t(G',v_1v_3v_5v_7))$. It can be shown that $\underline{b}_t(G';0) = \Omega(M_3)$, $\underline{b}_t(G';1) = \Omega(L_3 - B_3)$, and $b_t(G',\emptyset)b_t(G',v_1v_3v_5v_7) \leq M_3 L_3$. Thus, the probability of a b-rejection is at most $\mathcal{O}(M_3 B_3 / M_3 L_3)$. In addition, as shown above, the number of iterations of Phase 4 is at most $\mathcal{O}\big(L_3 M_3 / M_1^3\big)$. Then, the overall probability of a b-rejection during all of Phase 4 is at most $\mathcal{O}\big(M_3 B_3 / M_1^3\big) = \mathcal{O}\big(n^{3/(\gamma-1)} n^{1-\delta(\gamma-4)}/n^3\big) = o(1)$ for $\gamma > 21/10 + \sqrt{61}/10 \approx 2.88102$. □

Lemma 5.2. Let $\mathcal{D}$ be a plib sequence with exponent $\gamma > 21/10 + \sqrt{61}/10 \approx 2.88102$. Then, given $\mathcal{D}$ as input, the probability of a b-rejection in Phase 5 of Inc-Powerlaw is $o(1)$. ◀

Proof. Analogous to the proof of Lemma 5.1, we first bound the number of iterations in Phase 5. In each iteration, a double-edge is removed if we chose the d-switching, and a d-switching is chosen with probability $1 - \rho_{1,0,0} - \xi$, where $\xi = 32 M_2^2 / M_1^3 + 36 M_4 L_4 / M_2 L_2 M_1^2 + 32 M_3^2 / M_1^4$. It can be verified that the probability of choosing any of the other switchings is bounded by ξ. In Phase 5 of Pld $([, , t), h)$e probability of choosing a type-I switching is set to $1 - \rho_{III} - \xi'$, where $\xi' = 32 M_2^2 / M_1^3$. As the probability of not choosing a type-I switching in Pld $([v, a), n)$ishes with n, we know that the terms $\rho_{1,0,0} = \rho_{III}$ and $32 M_2^2 / M_1^3$ vanish with n. For the remaining two terms, first note that $L_{k+1} \leq L_k d_h = \mathcal{O}\big(L_k n^{\delta}\big)$ and $M_{k+1} \leq M_k d_1 = \mathcal{O}\big(M_k n^{1/(\gamma-1)}\big)$ for $k \geq 2$ [78]. Then, we have $36 M_4 L_4 / M_2 L_2 M_1^2 \leq M_2 d_1^2 L_2 d_h^2 / M_2 L_2 M_1^2 = \mathcal{O}\big(n^{2/(\gamma-1)+2\delta}/n^2\big) = o(1)$, and $32 M_3^2 / M_1^4 = \mathcal{O}\big(n^{6/(\gamma-1)}/n^4\big) = o(1)$ assuming $\gamma > 21/10 + \sqrt{61}/10 \approx 2.88102$. Thus, a double-edge is removed in each iteration with probability $1 - o(1)$, and as a graph satisfying the Phase 3, 4 and 5 preconditions may contain at most $L_2 M_2/M_1^2$ double-edges, the total number of iterations in Phase 5 is at most $\mathcal{O}(L_2 M_2/M_1^2)$.

We now show that the probability of a b-rejection is small enough. Again, the probability of a b-rejection in step 4 dominates the probability of a b-rejection in step 5 (compare Section 5.2.7). The rejection probability in step 4 is $1-\underline{b}_d(G';0)\underline{b}_d(G';1)/(b_d(G',\emptyset)b_d(G',v_1v_3v_5))$. Note that $\underline{b}_d(G';0) = \Omega(M_2)$, $\underline{b}_d(G';1) = \Omega(L_2 - A_2)$, and $b_d(G',\emptyset)b_d(G',v_1v_3v_5) \leq M_2 L_2$. Then, the overall probability of a b-rejection in Phase 5 is at most $\mathcal{O}(M_2 A_2/M_1^2) = \mathcal{O}\left(n^{2/(\gamma-1)}n^{(2\gamma-3)/(\gamma-1)^2}/n^2\right) = o(1)$ for $\gamma > 21/10 + \sqrt{61}/10 \approx 2.88102$. □

Appendix 5.B Correctness proofs of lower bounds

For Phases 3, 4 and 5, we use new lower bounds on the number of structures in the graph created by a valid switching.

For Phase 3, we factorize the lower bound on the number of inverse l-switchings used in PLD to obtain two new lower bounds $\underline{b}_l(G';0)$ and $\underline{b}_l(G';1)$.

Lemma 5.4. Let $\mathcal{S}$ be the class of graphs with m_t light triple-edges, m_d light double-edges and m_l light single loops (and no other non-simple edges). For all $G \in \mathcal{S}$, and all light simple two-stars $v_1v_2v_3$ in G that are created by a valid l-switching, we have

$$\underline{b}_l(\mathcal{S};0) \leq b_l(G,\emptyset) \tag{5.27}$$

$$\underline{b}_l(\mathcal{S};1) \leq b_l(G,v_1v_2v_3). \tag{5.28}$$

◀

Proof. We have $\underline{b}_l(\mathcal{S};0) = L_2 - 12m_t d_h - 8m_d d_h - m_l d_h^2$, and $b_l(G,\emptyset)$ is equal to the number of light simple ordered two-stars in G. We now show that $\underline{b}_l(\mathcal{S};0)$ is a lower bound on $b_l(G,\emptyset)$. First, each graph G matching the sequence contains exactly L_2 light ordered two-stars. We then overestimate the number of two-stars that are not simple, and subtract this from L_2: a two-star $v_1v_2v_3$ is not simple if one of the edges v_1v_2 or v_1v_3 is a triple-edge, a double-edge or a loop. There are at most $12m_t d_h$ that contain a triple-edge, as there are m_t ordered triple-edges ($6m_t$ ordered pairs), at most d_h choices for the remaining node of the two-star (any light node has degree smaller than d_h), and 2 ways to combine the selected pairs into the two-star as shown in Figure 5.3. Similarly, there are at most $8m_d d_h$ two-stars that contain a double-edge, as there are m_d double-edges ($4m_d$ ordered pairs), at most d_h choices for the remaining node and 2 ways to combine the selected pairs into the two-star, and there are at most $m_l d_h^2$ two-stars that contain a loop, as there are m_l loops and at most d_h^2 choices for the outer nodes of the two-star.

For the second bound, we have $\underline{b}_l(\mathcal{S};1) = M_1 - 6m_t - 4m_d - 2m_l - 2A_2 - 4d_1 - 2d_h$ and $b_l(G,v_1v_2v_3)$ is set to the number of simple ordered pairs (v_4,v_5) that (a) do share nodes with the two-star $v_1v_2v_3$ and (b) where v_2v_4 and v_3v_5 are non-edges. Each graph G matching the sequence contains exactly M_1 ordered pairs. There are at most $6m_t$

ordered pairs that contain a triple-edge, at most $4m_d$ ordered pairs that contain a double-edge and at most $2m_l$ ordered pairs that contain a loop. For case (a), there are at most $4d_1$ ordered pairs where $v_4 \in \{v_2, v_3\}$ or $v_5 \in \{v_2, v_3\}$, and at most $2d_h$ ordered pairs where $v_4 = v_1$ or $v_5 = v_1$. For case (b), we know that $A_2 = \sum_{i=1}^{d_1} d_i$ is an upper bound on the number of two-paths $v_2v_4v_5$ or $v_3v_5v_4$ [78], so there are at most $2A_2$ such pairs. □

For Phase 4, we use three new lower bounds $\underline{b}_t(G;0)$, $\underline{b}_t(G;1)$ and $\underline{b}_\tau(G;i+1)$.

Lemma 5.5. Let $\mathcal{S}$ be the class of graphs with m_t light triple-edges and m_d light double-edges (and no other non-simple edges). For all $G \in \mathcal{S}$, all simple three-stars $v_1v_3v_5v_7$ in G, and all triplets with $1 \leq i \leq k$ additional pairs $\overline{V}_{i+1}(S) = (v_1v_3v_5v_7v_2v_4v_6v_8, \ldots, v_{6+2i-1}v_{6+2i})$ in G that are created by a valid Phase 4 switching S, we have

$$\underline{b}_t(\mathcal{S};0) \leq b_t(G,\emptyset) \tag{5.29}$$

$$\underline{b}_t(\mathcal{S};1) \leq b_t(G, v_1v_3v_5v_7) \tag{5.30}$$

$$\underline{b}_\tau(\mathcal{S};i+1) \leq b_\tau(G,\overline{V}_{i+1}(S)). \tag{5.31}$$

◀

Proof. We have $\underline{b}_t(\mathcal{S};0) = M_3 - 18m_td_1^2 - 12m_dd_1^2$ and $b_t(G,\emptyset)$ is set to the number of simple ordered three-stars in G. Analogously to Lemma 1, we show that $\underline{b}_t(\mathcal{S};0)$ is a lower bound on $b_t(G,\emptyset)$ by starting with M_3, the number of ordered three-stars in a graph G matching the sequence and then subtracting an overestimate of the number of non-simple three-stars. The only non-simple three-stars contain a triple-edge or a double-edge. There are at most $18m_td_1^2$ non-simple three-stars that contain a triple-edge, as there are m_t triple-edges in G, at most d_1 choices for each of the two remaining outer nodes, and 18 ways to label the star as shown in Figure 5.4a. Similarly, there are at most $12m_dd_1^2$ three-stars that contain a double-edge.

For the second bound, we have $\underline{b}_t(G';1) = L_3 - 18m_td_h^2 - 12m_dd_h^2 - B_3 - 3(m_t + m_d)B_2 - d_h^3 - 9B_2$, and $b_t(G, v_1v_3v_5v_7)$ is equal to the number of light simple ordered three-stars that a) do not share any nodes with the three-star $v_1v_3v_5v_7$ created by S, b) have no edge v_1v_2 and no multi-edges v_3v_4, v_5v_6, v_7v_8. Each graph matching the sequence contains exactly L_3 light ordered simple three-stars. Analogous to $\underline{b}_t(\mathcal{S};0)$, there are at most $18m_td_h^2 + 12m_dd_h^2$ light three-stars that are not simple. There are at most $d_h^3 + 9B_2$ light simple ordered three-stars $v_2v_4v_6v_8$ of case a): first, if $v_2 = v_1$, then there are at most d_h^3 choices for the outer nodes. In addition, we know that for each node v_4 in G, there are at most $B_2 = \sum_{i=1}^{d_1}[d_{h+i}]_2$ light simple two-stars $v_2v_6v_8$ where v_2v_4 is an edge [78], so there are at most $9B_2$ three-stars where $v_4, v_6, v_8 \in \{v_3, v_5, v_7\}$. The only remaining case is if $v_2 \in \{v_3, v_5, v_7\}$, or if any of $v_4, v_6, v_8 = v_1$, but in this case v_1v_2 is an edge, so this falls under case b). For case b), it suffices to subtract $B_3 + 3(m_t + m_d)B_2$ three-stars: we know that for each node v_1 in G, there are at most $B_3 = \sum_{i=1}^{d_1}[d_{h+i}]_3$ light simple three-stars $v_2v_4v_6v_8$ where v_2v_1 is an edge. For a three-star where any of v_3v_4, v_5v_6, v_7v_8 is a multi-edge, we have at most $3(m_t + m_d)B_2$

choices, as there are $m_t + m_d$ multi-edges in G and choices for the first outer node, and at most B_2 choices for the center and the two remaining outer nodes.

For the third bound, we have $\underline{b}_\tau(\mathcal{S}; i+1) = M_1 - 6m_t - 4m_d - 16d_1 - 4(i-1)d_1 - 2A_2$, and $b_\tau(G, \overline{V}_{i+1}(S))$ is equal to the number of simple ordered pairs in G, that a) do not share any nodes with the triplet, or the previous $i-1$ pairs, and b) have no forbidden edges with the triplet. First, each graph matching the sequence contains exactly M_1 ordered pairs. At most $6m_t$ of those pairs are in a triple-edge, and at most $4m_d$ pairs are in a double-edge. For case a), there are at most $16d_1$ ordered pairs that share a node with the triplet, as for each of the 8 nodes of the triplet, there are at most d_1 choices for the second node of the simple pair and 2 ways to label the pair. Similarly, there are at most $4(i-1)$ pairs that share a node with the $i-1$ pairs relaxed in the previous steps. Finally, there are at most $2A_2$ pairs of case b): each of the two nodes in the pair cannot have an edge with one designated node of the triplet, and starting from that node, there are at most A_2 pairs connected to it via an edge. □

In Phase 5, we use three new lower bounds $\underline{b}_d(G; 0)$, $\underline{b}_d(G; 1)$ and $\underline{b}_\tau(G; k+1)$.

Lemma 5.6. Let $\mathcal{S}$ be the class of graphs with m_d light double-edges (and no other non-simple edges). For all $G \in \mathcal{S}$, all simple two-stars $v_1v_3v_5$ in G, and all doublets with $1 \le i \le k$ additional pairs $\overline{V}_{i+1}(S) = (v_1v_3v_5v_2v_4v_6, \dots, v_{4+2i-1}v_{4+2i})$ in G that are created by a valid Phase 5 switching S, we have

$$\underline{b}_d(\mathcal{S}; 0) \le b_d(G, \emptyset) \tag{5.32}$$
$$\underline{b}_d(\mathcal{S}; 1) \le b_d(G, v_1v_3v_5) \tag{5.33}$$
$$\underline{b}_\tau(\mathcal{S}; i+1) \le b_\tau(G, \overline{V}_{i+1}(S)). \tag{5.34}$$

◀

Proof. We have $\underline{b}_d(G'; 0) = M_2 - 8m_d d_1$, and $b_d(G, \emptyset)$ is set to the number of simple ordered two-stars in G. We now show that $\underline{b}_d(G'; 0)$ is a lower bound on $b_d(G, \emptyset)$. There are M_2 ordered two-stars in a graph G matching the sequence. Of these, the only ones that are not simple are the ones that contain a double-edge, and G can contain at most $8m_d d_1$ such two-stars.

For the second bound, we have $\underline{b}_d(G; 1) = L_2 - 8m_d d_h - 6B_1 - 3d_h^2$, and $b_d(G, v_1v_3v_5)$ is equal to the number of light simple ordered two-stars in that do not share any nodes with the two-star $v_1v_3v_5$. Similar to the first step above, G contains exactly L_2 light ordered two-stars, and at most $8m_d d_h$ light ordered two-stars that are not simple. The only remaining cases are two-stars $v_2v_4v_6$ that share any nodes with the first two-star. First, there are at most $6B_1$ two-stars where $v_4, v_6 \in \{v_1, v_3, v_5\}$, as B_1 is an upper bound on the number of pairs v_2v_4 or v_2v_6 where v_4 or v_6 are connected to one of v_1, v_3, v_5 via an edge. The other remaining case is $v_2 \in \{v_1, v_3, v_5\}$. In this case, there are at most d_h^2 choices for the remaining nodes of the two-star, so in total there are at most $3d_h^2$ such two-stars.

The proof for $\underline{b}_\tau(\mathcal{S}; i+1)$ is analogous to the proof for the similar bound in Phase 4 (see above). □

6 An Experimental Study of External Memory Algorithms for Connected Components

joint work with G.S. Brodal, R. Fagerberg, D. Hammer, U. Meyer, and M. Penschuck

We empirically investigate algorithms for solving *Connected Components* in the external memory model. In particular, we study whether the randomized $\mathcal{O}(\mathrm{sort}(E))$ algorithm by Karger, Klein, and Tarjan can be implemented to compete with practically promising and simpler algorithms having only slightly worse theoretical cost, namely Borůvka's algorithm and the algorithm by Sibeyn and collaborators.
For all algorithms, we develop and test a number of tuning options. Our experiments are executed on a large set of different graph classes including random graphs, grids, geometric graphs, and hyperbolic graphs.
Among our findings are: The Sibeyn algorithm is a very strong contender due to its simplicity and due to an added degree of freedom in its internal workings when used in the *Connected Components* setting. With the right tunings, the Karger-Klein-Tarjan algorithm can be implemented to be competitive in many cases. Higher graph density seems to benefit Karger-Klein-Tarjan relative to Sibeyn. Borůvka's algorithm is not competitive with the two others.

This chapter is based on the peer-reviewed conference article [41]:

[41] G. S. Brodal, R. Fagerberg, D. Hammer, U. Meyer, M. Penschuck, and H. Tran. An experimental study of external memory algorithms for connected components. In D. Coudert and E. Natale, editors, *Int. Symp. on Experimental Algorithms SEA*, volume 190 of *LIPIcs*, pages 23:1–23:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.SEA.2021.23 .

My contribution

David Hammer and I contributed an over-proportionally amount of material.

6.1 Introduction

CC: Connected Components

MSF: Minimum Spanning Forest

The *Connected Components* (*CC*) problem is a fundamental algorithmic task on undirected graphs and has a large number of applications including web graph analysis, communication network design, image analysis, and clustering in computational biology. *CC* may be viewed as a smaller sibling of the *Minimum Spanning Forest* (*MSF*) problem defined on weighted, undirected graphs — any algorithm solving *MSF* and able to return the trees of the forest one by one can be used to solve *CC* by first assigning arbitrary edge weights.

In internal memory, *CC* is simple to solve in linear time by DFS or BFS. A long-standing open problem is whether *MSF* can also be solved deterministically in linear time. The large body of work devoted to the question (see e.g., the references in [151]) indicates that in internal memory, *MSF* is harder to tackle than *CC*, at least in terms of the algorithmic sophistication needed (and potentially also in terms of the asymptotic complexity of the problem).

In external memory (see Section 6.2 for the definition of the model and its parameters), the I/O-complexity of *CC* and *MSF* is bounded from below by $\Omega(E/V \cdot \mathrm{sort}(V))$ [142] and a number of algorithms come within at most a logarithmic factor of $\mathcal{O}(\mathrm{sort}(E))$. No deterministic algorithm is known to match the lower bound, but a randomized algorithm with $\mathcal{O}(\mathrm{sort}(E))$[1] expected cost exists [108, 51]. Unlike in internal memory, the known external memory *CC* algorithms are essentially the same as the known algorithms for *MSF*, either exactly or as close variants. The largest discrepancy between the two settings is for the randomized $\mathcal{O}(\mathrm{sort}(E))$ algorithm, where a fairly involved subroutine in its *MSF* variant becomes straight-forward for *CC*.

It seems that the randomized $\mathcal{O}(\mathrm{sort}(E))$ external memory algorithm was never empirically investigated. One aim of this paper is to carry out such an investigation in the *CC* setting where the discrepancy mentioned above gives the algorithm the largest opportunity of being competitive in practice. Due to the large size of internal memory in most current computer systems, it is not clear whether a small asymptotic advantage of at most a logarithmic factor will materialize in practice for graphs of very large, but still plausible, sizes. In more detail, we want to investigate implementations and tuning options for the randomized $\mathcal{O}(\mathrm{sort}(E))$ *CC* algorithm, as well as for the practically most promising of the remaining (asymptotically slightly worse, but often simpler) external *CC* algorithms, and then compare the best implementations of each algorithm on a broad range of graph classes. More generally, the aim of this paper is to investigate the best algorithmic choices for solving the *CC* problem in external memory.

Previous work In the semi-external case, where $V \leq M$, scanning the edges and maintaining the components via a Union-Find data structure in internal memory will solve *CC* in $\mathcal{O}(\mathrm{scan}(E))$ I/Os. The classic Borůvka *MSF* algorithm was externalized by

[1]Using sparsification, the algorithm can be implemented to use $\mathcal{O}(E/V \cdot \mathrm{sort}(V))$ I/Os [51], matching the lower bound exactly. In this paper, we will consider its $\mathcal{O}(\mathrm{sort}(E))$ version as the two bounds are very close and in practice their difference is unlikely to outweigh the added algorithmic complication.

Chiang et al. [51] by showing how to implement a Borůvka step in $\mathcal{O}(\mathrm{sort}(E))$ I/Os, leading to $\mathcal{O}(\log(V/M) \cdot \mathrm{sort}(E))$ I/Os for the entire algorithm. A simpler method for implementing a Borůvka step in $\mathcal{O}(\mathrm{sort}(E))$ I/Os was later given by Arge et al. [15]. Munagala and Ranade [142] gave a *CC* algorithm using $\mathcal{O}(\log\log(VB/E) \cdot \mathrm{sort}(E))$ I/Os and also proved the above-mentioned lower bound. The algorithm was generalized to *MSF* by Arge et al. [15], keeping the I/O bound. The algorithm of [15] was further developed by Bhushan and Gopalan [33], slightly improving the I/O bound.

Karger, Klein, and Tarjan [108] gave an internal *MSF* algorithm with expected $\mathcal{O}(E)$ running time using a linear time *MSF* verification algorithm as its central subroutine. The algorithm can be externalized to use expected $\mathcal{O}(\mathrm{sort}(E))$ I/Os [51] by using external Borůvka steps and the external *MSF* verification algorithm by Chiang et al. [51]. For *CC*, it is an easy observation (already made by [4]) that the *MSF* verification can be substituted by a contraction step, which simplifies the implementation considerably. To the best of our knowledge, neither the *CC* nor the *MSF* variant of this external memory algorithm has been studied empirically.

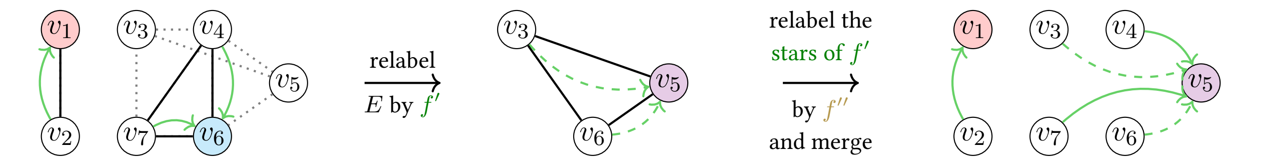
A very simple randomized *MSF* algorithm using expected $\mathcal{O}(\log(V/M) \cdot \mathrm{sort}(E))$ I/Os was developed by Sibeyn and Meyer. It was first reported by Schultes [165], and further described and empirically tested by Dementiev et al. [61]. A *CC* variant was theoretically and empirically studied by Sibeyn [167] (and to a lesser extent by Schultes [166]). Due to its simplicity, the algorithm is likely to have very competitive constants in its I/O bound, which is argued theoretically in [167] and substantiated by the experiments in [165, 166, 61, 167]; however, none of these experiments include comparisons to other external memory algorithms.

Our contribution We implement the *CC* version of the $\mathcal{O}(\mathrm{sort}(E))$ randomized and external algorithm by Karger, Klein, and Tarjan [108] and develop and investigate a number of tuning options. We then compare it to tuned versions of what we consider the practically most promising other algorithms for the *CC*, namely external Borůvka and the algorithm by Sibeyn et al. [165, 166, 61, 167]. Our experiments are executed on numerous graph classes, including $\mathcal{G}(n, p)$ graphs, grids, geometric graphs, and hyperbolic graphs (see Section 6.6).

Among our findings are: Sibeyn's algorithm is a very strong contender due to its simplicity and due to an added degree of freedom in its internal workings when used in the *CC* setting. With the right tunings, the Karger-Klein-Tarjan algorithm can be implemented to be competitive. Higher graph density seems to benefit Karger-Klein-Tarjan relative to Sibeyn, as does larger graph sizes. The latter observation is in line with its better (expected) asymptotic I/O bound. Borůvka's algorithm is not competitive compared to its contenders.

6.2 Definitions

The *Connected Components* (*CC*) problem on an undirected graph $G = (V, E)$ is to partition V such that two nodes are in the same subset iff they are connected by a path



v	v_1	v_2	v_4	v_6	v_7
$f'(v)$	v_1	v_1	v_6	v_6	v_6

(a) Obtain f' by solving CC on E'

v	v_3	v_5	v_6
$f''(v)$	v_5	v_5	v_5

(b) Obtain f'' by solving CC on E relabeled by f'

v	v_1	v_2	v_3	v_4	v_5	v_6	v_7
$f(v)$	v_1	v_1	v_5	v_5	v_5	v_5	v_5

(c) Obtain f by merging the star of f'' and the star of f' relabeled with f''

Figure 6.1: Relabeling and contraction. The input E and the subset $E' \subseteq E$ are illustrated in **(a)** where E' corresponds to solid black edges and $E \setminus E'$ to dotted lines. Solving *CC* on E' yields f' which represents the two CCs $\{v_1, v_2\}$ and $\{v_4, v_6, v_7\}$ by v_1 and v_6, respectively. This corresponds to the two stars indicated by the directed green edges. Since v_3 and v_5 are not covered by E', they are also not included in f'. The result of the contraction E/E' is shown in **(b)** with solid lines and is obtained by relabeling E by f'. Solving *CC* on E/E' yields f'' indicated by the dashed directed edges. In **(c)**, we merge the stars of f' relabeled with f'' (solid) together with the stars of f'' (dashed) and obtain the final result f. Observe that the star of f may contain edges (e.g. (v_7, v_5)) that were not part of the original input E.

in G. We overload the symbols V and E: depending on the context, V may represent either the *set* or the *number* of nodes, and E may similarly represent either the set or the number of edges.

We analyze the cost of algorithms in the I/O-model of Aggarwal and Vitter [1] where M denotes the size of the internal memory, B denotes the block size, and $\mathrm{scan}(N) = \Theta(N/B)$ and $\mathrm{sort}(N) = \Theta(N/B \log_{M/B}(N/M))$ denote the costs of scanning and sorting N elements.

As input, we assume the standard external memory representation of a graph as a list of its edges. This means that isolated nodes cannot be represented and should be handled separately by the user, which is straight-forward as they constitute their own connected components. We denote by $V(E)$ the set of nodes contained in an edge set E. Hence, an input is formally a graph $G = (V(E), E)$, but for simplicity we denote it just by E. We require the input E to be given in lexicographical order, as all our algorithms need this. We thereby avoid an initial sorting step in the algorithms, which would only make their relative differences in running times less clear. Unless otherwise stated, we also assume that each unordered edge $\{u, v\}$ is stored only once in its *normalized* form $(\min(u, v), \max(u, v))$.

As output, we require a mapping $f\colon V(E) \to V(E)$ where $f(v) = f(u)$ iff u and v are in the same connected component. In other words, for each connected component one node is chosen as its representative. Concretely, the mapping shall be returned as the list of pairs $\{(v, f(v)) \mid v \in V(E)\}$, except that all identities (v, v) are omitted. Note that we can interpret this output as the edge list of a directed graph composed of disjoint *stars*, where a star is a set of nodes pointing to a common center node. Each star represents a connected component in E.

A *relabeling* of a graph E by a mapping $f\colon V(E) \to V(E)$ means applying f to all edge endpoints and then removing parallel edges and self-loops in the resulting edge list. If f is given by a graph of oriented stars as described above, a relabeling can be implemented in $\mathcal{O}(\mathrm{sort}(E))$ I/Os by $\mathcal{O}(1)$ sorting and scanning steps on E. A *contraction* E/E' of a graph E by a subset $E' \subseteq E$ of its edges means solving *CC* on E' and then relabeling E by the returned mapping f'. The concepts of relabeling and contraction are illustrated in Figure 6.1 **(a)** and **(b)**, respectively.

Note that if we next obtain a mapping f'' by solving *CC* on the contracted graph E/E', we can solve *CC* on the original graph E as follows: use the mapping f'' to relabel the graph of stars representing f' (only the target of each star edge is affected by the relabeling) and then return the union of those relabeled edges and the edges of the graph of stars representing f''. The process is illustrated in Figure 6.1 **(c)**. It is easy to verify that it will produce a graph of stars representing the solution f to *CC* on the original graph E. All recursive algorithms in the current paper use this process as their framework.

6.3 Algorithms

In this section, we describe the basic versions of the implemented algorithms.

Union-Find In the semi-external case, where $V(E) \leq M$, scanning the edges once while maintaining a Union-Find data structure on $V(E)$ in internal memory solves *CC* in $\mathcal{O}(\mathrm{scan}(E))$ I/Os and $\mathcal{O}(E\alpha(E, V(E)))$ time [175], where α is the inverse Ackermann function. We use this as a base case.

Borůvka A Borůvka step in the *MSF* setting means letting each node choose an incident edge of minimum weight and then contracting the graph by the set E' of chosen edges. In E', each node is in a connected component of size at least two, so the number of nodes is at least halved in the step. As a Borůvka step requires $\mathcal{O}(\mathrm{sort}(E))$ I/Os (see below), this leads to a recursive algorithm which will use $\mathcal{O}(\log_2(V/M)\,\mathrm{sort}(E))$ I/Os before the semi-external base case is reached. This constitutes Borůvka's algorithm.

The first part of a Borůvka step finds E' with $\mathcal{O}(\mathrm{sort}(E))$ I/Os as follows: double E during a scan to make it contain both directions of each undirected edge. Then for all nodes choose an incident edge of minimum weight via a single sort and scan of this version of E.

To implement the remainder of a Borůvka step, one can exploit that E' is a graph where each connected component has exactly one cycle, as seen by repeatedly following paths of chosen edges until all nodes have been visited. Assuming that all edge weights are unique (otherwise, use node IDs as tie-breakers), the weights along any such path are strictly decreasing, except when traversing the lightest undirected edge $\{u, v\}$ of the component in two directions $(u, v), (v, u)$, implying that the cycle is a two-cycle. Both directions have the same normalized representation, hence can be identified and de-duplicated by sorting E', after which the connected component corresponds to a tree rooted in v. This can be done for all such pairs in the same sorting step, making the edges E' form a forest where each tree coincides with a connected component. We select the roots as the components' IDs.

In order to return the star graph of the mapping f', we have to inform each node of its tree's root. Early external methods [51, 4] used algorithms for Euler tours of trees based on list ranking. We use a simpler method described in [15]. It requires $\mathcal{O}(\mathrm{sort}(V(E)))$ I/Os and is based on the fact that edge weights are strictly increasing on root-to-leaf paths in the trees, i.e., if we address messages to nodes by the weight of their incoming edge, parents will be processed before their children. This allows edge weights to be used as a “time line” in a general technique known as *Time Forward Processing* [123]. The propagation is done for all trees simultaneously by maintaining a set of signals in an external priority queue. The data structure is initialized by inserting signals for all children of all roots. Using sorting steps, we also create a list L of tree edges not incident to a root. In L, all child edges of a node v are grouped together, and the order between groups is determined by the weight of the parent edge of v. We then repeatedly remove the signal with smallest key from the priority queue and forward the information contained to the next block of children from L.

In the *CC* setting, the above algorithm for a Borůvka step can be implemented by (formally) assigning to all edges their unique normalized identity as their weight. Note that in the first part of the step, this is equivalent to each node simply choosing the edge to the neighbor with the lowest ID.

Karger-Klein-Tarjan The *CC* version [4] of the $\mathcal{O}(\mathrm{sort}(E))$ randomized, external algorithm based on Karger, Klein, and Tarjan [108] has the following recursive structure:

1. Perform three Borůvka steps on the input graph. Let the result be E.
2. Let E' contain each edge of E independently with probability $1/2$.
3. Compute the connected components of E' recursively.
4. Form the contraction $E'' = E/E'$.
5. Compute the connected components of E'' recursively.
6. Relabel the result of step 3 by the result of step 5 and merge with the result of step 5, as detailed in Section 6.2.
7. Perform the relabelings and merges corresponding to the contraction in each of the initial Borůvka steps (as detailed in Section 6.2) and return the result.

In step 4, only the edges in $E \setminus E'$ need to be processed as contraction by E' eliminates all edges in E'. The crux of the Karger-Klein-Tarjan algorithm is that the number of edges in E'' is $\mathcal{O}(V(E))$ in expectation. The argument for this is as follows (adapted from [108] to the *CC* setting).

Consider building a spanning tree F for E' by the standard Union-Find based algorithm *while* performing the sampling. That is, consider each edge e of E sequentially and include it in F *iff* it is sampled *and* it does not form a cycle with edges already in F. Case 1: e forms a cycle. Then e will not appear in E'' due to the contraction.

Case 2a: e does not form a cycle, and is sampled. Then e will not appear in E'' due to the contraction (as it is included in F). Case 2b: e does not form a cycle, and is not sampled. Then e may appear in E''. Since the final F is a spanning tree of E', we have $|F| \leq V(E') - 1$ and hence $|F| < V(E)$. Thus, the number of Case 2b edges is a stochastic variable upper-bounded by a negative binomial distribution with $p = 1/2$ and $r = V(E)$ (the number of tails before $V(E)$ heads have appeared when flipping a fair coin). Therefore the expected number of Case 2b edges is at most $V(E)$, implying the same for the expected number of edges in E''.

This statement is analogous to Lemma 2.1 of [108] for the *MSF* version. The rest of the argument in [108] for the expected cost carries over[2] almost verbatim, with $\mathcal{O}(E)$ time substituted by $\mathcal{O}(\text{sort}(E))$ I/Os.

Sibeyn The *MSF* algorithm presented in [61] is a surprisingly simple I/O-efficient algorithm. It works by repeatedly letting some node select its minimum incident edge and contracting that edge. These contractions are done in a lazy fashion using the time-forward processing method with node IDs as the "time" dimension. The original algorithm is described in two versions: one using buckets and the other using a priority queue.

We here describe the version based on priority queues. The algorithm represents the undirected edges only in their normalized form (oriented from lower to higher ID). All edges are initially inserted into a priority queue (PQ) which is ordered by source first and edge weight second. This ordering allows the algorithm to perform node contractions by repeatedly extracting the minimum edge in the PQ. When the extracted edge (u, v, w) has a new source u compared to the previous extracted edge, $\{u, v\}$ is the lightest edge incident to u (after the contractions done so far) and is output as an MSF edge. The edge $\{u, v\}$ is then contracted and u's remaining edges are forwarded to (i.e., taken over by) v. In detail, all subsequent edges (u, v', w') with source u extracted from the PQ become $\{v, v'\}$ by inserting $(\min\{v, v'\}, \max\{v, v'\}, w')$ into the PQ, except that edges with $v' = v$ (i.e., self-loops) are skipped. In this MSF version of the algorithm, forwarded edges need to be annotated with the original node IDs of their endpoints, in order for the output to be a correct MSF. When the number V' of source IDs remaining in the PQ can fit in internal memory, i.e., when $V' \leq M$, the rest of the edges in the PQ are extracted and a semi-external version of Kruskal's algorithm is run on them. If using randomized node IDs, the algorithm requires expected $\mathcal{O}(E \log(V/V'))$ priority queue operations to contract the original node set V to a smaller node set V' (i.e. for contracting $V - V'$ nodes) [61]. This implies a total cost of $\mathcal{O}(\log(V/M) \cdot \text{sort}(E))$ I/Os.

In our setting, the goal is to compute connected components. This allows the algorithm to be simplified in a number of ways (some described in [166]). The tree that the algorithm outputs should only capture connectivity, hence its edges need not be edges from the original input E, so there is no need to annotate forwarded edges with original node IDs. Additionally, one can choose an *arbitrary* edge out of the "current"

[2]The argument in [108] allows for using only two initial Borůvka steps. We here follow the description of the *CC* algorithm in [4], which uses three.

source u as the new target to forward edges to. A natural heuristic is to send the information as far forward in time as possible. This is achieved by simply ordering the PQ by source in increasing order and by target in decreasing order as the first edge out of each new source will then go to the furthest neighbor (or known reachable node due to forwarded edges) immediately.

As the final CC information should be represented as a set of stars, some post-processing has to be done on the rooted trees output by the modified node contraction algorithm. As node IDs give a topological ordering of the tree edges, one can simply reverse the tree edges and use time-forward processing in the opposite direction relative to the node contraction phase. This post-processing only incurs $\mathcal{O}(\mathrm{sort}(V))$ additional I/Os.

The bucket version of the algorithm replaces the priority queue with a set of unsorted buckets. Two variants are described in the CC setting in [167, Section 3.4]: one which processes each bucket in internal memory and one which uses the semi-external Union-Find algorithm on each bucket. Choosing bucket sizes ahead of time for the former variant is non-trivial as the density tends to increase during computation. We therefore focus on the latter variant in this paper.

Randomized-Borůvka A standard Borůvka step has a first part where each node selects an incident edge, and a second part where the connected components of this edge set E' are found via time-forward processing and returned as a mapping represented by a star graph.

We now describe a novel randomized method for the second part which is simpler than time-forward processing, at the cost of a worse bound on the contraction factor. In Section 6.7, we empirically investigate whether this trade-off is beneficial for the overall I/O cost when using Borůvka steps (as part of Borůvka's or Karger-Klein-Tarjan's algorithm).

We consider the selected edge of a node as an outgoing oriented edge. The method is simple: 1) Let each node keep its selected edge with probability p, resulting in the edge set E''. 2) Mark all edges (u, v) in E'' for which E'' contains an edge (w, u), then remove all marked edges to give the final edge set E'''. Step 1) can be done during the edge selection process at no cost, and step 2) can be done in one additional sort and scan step. No (oriented) path in E''' has length more than one, hence E''' is a star graph itself (it represents its own connected components) and can just be returned. Note that while the star-graph computation discussed for the original Borůvka algorithm requires the cycle of a connected component to be a two-cycle, and therefore requires nodes to choose minimum incident edges according to some assigned unique edge weights, this is not the case for our randomized variant.

Lemma 6.1. E''' has expected size of at least $p(1-p)V(E)$. ◀

Proof. E' has size $V(E)$, so the expected size of E'' is $pV(E)$. If we for each edge (w, u) in E' count a mark whenever (w, u) was kept *and* (u, v) was kept (where (u, v) is u's chosen edge), then we have an upper bound on the total number of marks (it is an upper bound, as (u, v) could also be counted as marked via another edge (w', u), but (u, v) can only hold one mark). Hence, the expected number of edges removed from E'' to E''' is less than $p^2 V(E)$. Thus, the expected size of E''' is at least $p(1-p)V(E)$, which is maximized for $p = 1/2$. □

When contracting using the star graph E''', each edge of E''' will remove at least one node, so at least $1/2(1-1/2)V(E) = V(E)/4$ nodes are removed in expectation. Thus, the expected contraction factor is at least $1/(1-1/4) = 4/3$. The contraction for a given graph may be larger than this (just as for standard Borůvka steps and its lower bound of two on the contraction factor). In Section 6.7, we empirically study contraction factors.

6.4 Tuning Options

We suggest and experimentally evaluate several variations of the algorithms with potential for impact on their practical running times and I/O costs.

Pipelining Pipelining is the concept of one algorithmic sub-routine handing its output directly to another sub-routine without storing the intermediate data on disk. Applying this where possible can save I/Os, and our implementation platform *STXXL* offers tools for this type of programming. Before settling on using it, however, we want to investigate its impact.

Contraction Sub-routine In Borůvka's algorithm, and in the first step of the Karger-Klein-Tarjan algorithm, nodes are contracted. We investigate if *Time Forward Processing* based Borůvka steps or the proposed randomized version will be the fastest. The general form of the I/O cost argument in [61, 165] states that if Sibeyn's algorithm is run until the number of nodes has been contracted from V to V', it uses expected $\mathcal{O}(\log(V/V') \cdot \mathrm{sort}(E))$ I/Os. Thus, another possible contraction sub-routine in Karger-Klein-Tarjan is to use Sibeyn.

Omitting Node Contractions at the Root in Karger-Klein-Tarjan From the details of the cost analysis of Karger-Klein-Tarjan [108], it seems likely that the initial contraction in the root node of its recursion tree will dominate the running time in practice. The asymptotic result of expected $\mathcal{O}(\mathrm{sort}(E))$ cost still holds if this contraction (but not the contractions in other nodes of the recursion tree) is omitted. Then the algorithm will simply start with a scan of the input edge list when sampling edges before the first recursive call. If the returned mapping happens to contract nodes and edges well, the second recursive call will not contribute much to the total I/O cost, either. In this case, the dominating part will be the contraction after the first recursive call, which comprises two sorting steps and two scannings steps on E (if we enter the base case in the second recursive call, we can even save one of the sorting steps, because the edges do not need to be sorted before making the call).

Sampling Parameter in Karger-Klein-Tarjan The original sampling probability for edges before the first recursive call in Karger-Klein-Tarjan was set to $p = 1/2$, but other values are possible. Lowering p makes the first recursive call cheaper, and for denser graphs, we may still have a good effect of the contraction before the second recursive call, because a sparser subset of edges may still span large portions of the connected components. If this turns out to be true, one could make p depend on the density (lower p when the density is higher).

Approximate Counting Algorithms for Size Estimation In the recursive algorithms, there is a need to estimate V in order to know when the semi-external base case can be entered. One idea is to use approximate counting algorithms [10, 21, 68] from the streaming community to determine an estimate on the number of unique nodes in the edge list. In the streaming model this problem is referred to as the *Distinct Elements* problem and most solutions only provide a (δ, ε) guarantee, meaning that the estimate is within a $(1 + \varepsilon)$-multiplicative error with probability at least $(1 - \delta)$. As smaller values of ε and δ require more internal work (mostly in the form of more evaluations of independent hash-functions), we investigate if we can benefit from these methods while staying I/O-bound.

Which Neighbor to Contract in Sibeyn In each step of Sibeyn, the *MSF* version of the algorithm must choose to contract the current node and its neighbor given by its incident edge of minimum weight. In the *CC* version, it is free to choose any neighbor. As argued in [61], it may be beneficial to choose the neighbor with largest node ID. We investigate what is the best choice and the gains possible, and we empirically compare choosing a neighbor with largest node ID, a neighbor with smallest node ID, and a random neighbor (which corresponds to the *MSF* version).

Minimizing the PQ in Sibeyn When running the PQ version of Sibeyn, we may exploit that the input edges are sorted. This allows us to skip the initial insertion of all edges into the PQ: while running the algorithm, the list of original edges can just be merged with the output of the PQ, which then only needs to contain reinserted edges, not original edges.

Influence of Relinking in Sibeyn with Buckets In the bucket version of Sibeyn's algorithm, the connected components for a bucket are computed and signals are sent to later buckets. Sibeyn [61] introduces a *relinking* variant which restructures the signals before sending them to reduce the number of signals between buckets.

6.5 Implementation

All algorithms are implemented in C++ using the *STXXL* library [60], which offers highly tuned external memory versions of fundamental algorithmic building blocks like sorting and priority queues. It also supports pipelining, as described in Section 6.4. The external priority queue of *STXXL*, which we use in several places in the algorithms implemented, is based on [160].

In order to accommodate different contraction schemes in the contraction of the recursive Karger-Klein-Tarjan algorithm, we implemented a generic framework for performing the sampling, contraction, relabeling and merging during the algorithm's execution. The supported conraction schemes are Sibeyn, Karger-Klein-Tarjan and Randomized-Borůvka contractions. The framework comes in two flavors: as a purely vector-based and a pipelined stream-based implementation. This allows us to evaluate to what degree pipelining is beneficial.

Edge Representation In our implementation we store each undirected edge by its ordered pair (u, v) where $u < v$. For sorted edges we additionally employ a more I/O-efficient data structure: consecutive edges with the same source u are compressed to a single entry u followed by all its adjacent nodes and a delimiter.

Data Structures The pipelined implementations make use of several *STXXL* data structures. In these, generated data is not saved in an explicit vector but fed to a container then functions as a data stream with read-only access. An example for this is *STXXL*'s *sorter*: in the first phase, items are 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 sorted stream which can be rewound at any time. While a sorter is functionally equivalent to filling, sorting, and reading back an external memory vector, the restricted access model reduces constant factors in the running time and I/O cost [24].

Semi-external Base Case While we assume that the number of nodes in the original input is known exactly, this is not necessarily true for recursive calls. As we aim to switch to a semi-external base case algorithm, keeping track of the number of remaining nodes is essential. For the Karger-Klein-Tarjan algorithm, the node contraction step contracts a specified number of nodes and as such, a good estimate for the number of nodes remaining after initial contraction is simply the original node count minus the number contracted[3]. The same holds for the contracted edge set passed on to the second recursive call: the number of connected components returned from the first recursive call is known and corresponds to the number additional nodes contracted. This leaves the first recursive call operating on the sampled edges E'. The number of nodes here is trivially bounded both by the bound known before sampling and by $2|E'|$. The latter bound can be improved somewhat; as edges are kept sorted, the number of unique sources can be counted while sampling and only $|E'|$ is added for an upper bound. Taking the best of these bounds at different stages, we maintain an upper bound on the node count. By using these upper bounds we can save I/Os in the relabeling as relabeled edges may immediately be piped into the semi-external base case without the

[3]This gives an exact count except when a connected component is contracted to a singleton — at which point it will not appear in the edge list.

otherwise required final sorting step. Note that while computing the exact number of nodes requires only a few scanning and sorting steps, this is too costly in practice for competitive results.

6.6 Graph Classes

For our experiments, we use a variety of different synthetic graph models. We consider four types: the *Gilbert* type classic random graphs, *Random Geometric Graph*s and *Random Hyperbolic Graph*s, both belonging to the class of spatial network models, and finally deterministically generated grid graphs. Using scalable graph generators, we generate fully external ($M < V$) graphs with a range of different parameters. For a recent overview of such generators, see [150].

Gilbert Graphs In the $\mathcal{G}(n, p)$ model of Gilbert [79], each edge is present independently with probability p. The $\mathcal{G}(n, p)$ model can generate graphs with a varying number of connected components for sufficiently small p. It is widely used in empirical work, but its degree distribution is often considered atypical compared to real-world instances.

Random Geometric Graphs *Random Geometric Graph*s (*RGG*s) [80, 148] are a simple case of spatial networks where graphs are projected onto Euclidean space. In *RGG*s n points are placed uniformly at random into a d-dimensional unit-cube $[0, 1)^d$ where any two points are connected if their Euclidian distance is below a given threshold r. To generate graphs in this model, we use the generator available in KaGen [74].

Random Hyperbolic Graphs *Random Hyperbolic Graph*s (*RHG*s) [112, 87] are a special case of spatial networks where graphs are projected onto hyperbolic space. We describe the threshold model, the simplest *RHG* variant [87]. The points are randomly placed onto a two-dimensional disk in hyperbolic space where the radial probability density function increases exponentially towards the border. The angular coordinate is sampled uniformly at random from $[0, 2\pi)$ and points are connected if their hyperbolic distance is less than a given threshold R. The density of points near the center is controllable by setting a dispersion parameter α. One interesting feature of *RHG*s is that the node degrees follow a power law distribution which is often found in real-world graph instances, in particular when generated via human activities and choices. In the threshold model the exponent is $\gamma = 1 + 2\alpha$ with high probability [87]. To generate graphs in this model, we use the HyperGen generator [149].

Grid Graphs We consider two different types of square grid graphs. In both versions, the nodes are seen as points in a two-dimensional grid; (x, y) for $1 \leq x \leq w$ and $1 \leq y \leq h$. For the simpler version, nodes are connected horizontally and vertically to their neighbors. All nodes except for boundary nodes thus have degree 4. To achieve higher degree, we additionally consider generalized grids in which nodes are connected to all nodes within distance d under the infinity norm. That is, node (x, y) has edges to nodes $(x+i, y+j)$ where $-d \leq i \leq d$ and $-d \leq j \leq d$, except where this exceeds the grid boundary. Internal nodes in these graphs have degree $4d(d+1)$. To investigate the effects of increasing the number of components, we additionally generate graphs which we refer to as *cubes* consisting of multiple disjoint layers, each of which is a generalized grid graph.

6.7 Experiments

Our experiments were carried out in two phases. In the first phase, we investigated the impact of the various algorithmic variants and proposals for tuning described in Sections 6.3 to 6.5. This was done on subsets of the test graphs of Section 6.6 and selected other test cases. The aim of this phase was to develop a set of well-engineered implementations of the most promising contenders. In the second phase, we then compared those on a large set of test graphs of Section 6.6 — the compute time of this phase alone comprised one third of a year. Below, we describe our experimental setup and our learnings from each of the two phases. For space reasons, we mainly include plots for the second phase. The full set of plots are in Section 6.A (Appendix) (in the plots, the numbers V and E are denoted by n and m, respectively).

6.7.1 Experimental Setup

The experiments were run on individual nodes of the Goethe-HLR cluster at Goethe University Frankfurt, as this allowed us to run many experiments simultaneously (note that our algorithms all are sequential, parallel algorithms for *CC* are beyond the scope of this paper). The nodes each have Intel Xeon Skylake Gold 6148 CPUs and 192 GB of RAM. Each node has a HGST Ultrastar HUS726020ALA610 hard drive which was used for the *STXXL* disk file. The code was compiled using GCC version 8.3.1 with the optimization parameters `O3` and `march=native`.

In each run, the input graph was first loaded onto the local hard drive in the appropriate *STXXL* data structure: an edge stream for the stream-based implementations and an *STXXL* vector for the vector-based implementations. The threshold for switching to the semi-external base case was for all the algorithms set to 33,554,432 nodes which corresponds to 256 MiB of node IDs. To capture wall-time and I/O volume, we used the `iostats` module provided by the *FOXXLL* library (a component of *STXXL*). The main timing plots in Figures 6.10 to 6.17 show the wall time (bars) and total I/O volume (bytes read plus bytes written during the execution of the algorithm) reported by the `iostats`.

To keep the combined compute time of the experiments from becoming infeasible (even when executing experiments in parallel on a cluster), we reduced the RAM used by the CPUs to a few GB, which allowed us values of V/M up to 80 and graph densities E/V up to 20 (although not both maximal values at the same time) while keeping individual experiments under half a day of compute time. Our hypothesis was that if the algorithms are I/O-bound, the relative running times of the algorithms would stay approximately the same even if moving to larger sizes of RAM and from the hard disks of the cluster nodes to solid-state disks. With the set of final contenders, we conducted experiments on selected graph classes on a single machine having 16 GB of RAM and a RAID with six solid-state disks of 480 GB each. Those experiments confirmed our hypothesis, as the relative running times changed less than 20% in almost all cases tested.

To limit the amount of memory used on the cluster nodes, we limited the internal memory allowed for *STXXL* primitives used (sorting streams and priority queues were limited to 1 GB of RAM each). With the base case threshold (accounting for overhead), and the above limits, the implementations should be able to run with approximately 2 GB of memory. We did not have a mechanism to enforce a strict bound on the memory actually allocated, but monitored the amount of RAM actually used, which was in the range of 2 GB to 5 GB. To force disk accesses rather than additional buffering, the `direct` flag was used for the *STXXL* disk file.

6.7.2 Phase 1 – Initial Findings

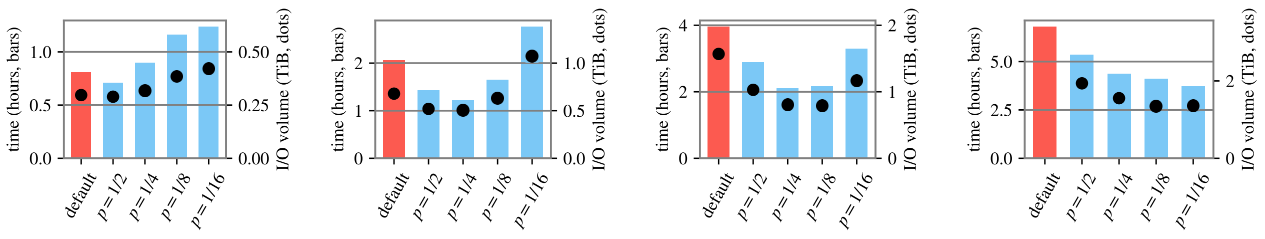
We now describe our main findings in phase one of our experiments. Unless otherwise mentioned, the measure compared is wall clock time.

Randomized-Borůvka and Borůvka

For our suggestion for randomized Borůvka steps, we first investigated the impact on the observed contraction ratio of a number of different edge representations and of various sampling parameters. On most graph classes, sampling parameters much closer to one than to 1/2 gave better contraction ratios (see Figure 6.5), in line with Lemma 6.1 only giving a lower bound. There was correlation among the graph classes between increased contraction efficiency of the randomized Borůvka steps and increased contraction efficiency of standard Borůvka (past the lower bound of two on the ratio). However, the ratio was consistently worse for the randomized version, and its simpler code did not make up for this when considering the total time of Borůvka's algorithm. Additionally, both of the two versions of Borůvka's algorithm were clearly worse than Sibeyn's algorithm based on PQs, both before and after adding pipelining. For instance, when doing node contraction until the base case is reached, we found that Sibeyn's algorithm was approximately 59% faster than ordinary Borůvka and we likewise found that one variant of our Karger-Klein-Tarjan implementation using Sibeyn's algorithm for node contraction was around 58% faster than one using the randomized Borůvka steps (results vary across graphs, numbers given here are averages). We therefore left Borůvka's algorithm out of the final race.

Pipelining

Adding pipelining in *STXXL* turned out to improve our implementations of Sibeyn and of Karger-Klein-Tarjan. Introducing pipelining (including compressed edge streams) reduced the running time of one of our Karger-Klein-Tarjan variants approximately



(a) $m/n = 2.04$ (b) $m/n = 5.00$ (c) $m/n = 10.00$ (d) $m/n = 20.00$

Figure 6.2: (Subset of Figure 6.10) Running times and I/O volumes for $\mathcal{G}(n, p)$ graphs with a node set size of 5GiB and varying density. The *default* variant always contracts and has a sampling probability of $p = 1/2$. The remaining variants skip contraction in the root and have fixed sampling probabilites. For $m/n = 20$, the *default* variant exceeded the local hard disk's capacity leading to a halt in the algorithm's execution. We thus only report the elapsed wall time up until that point.

73% on average over a simple version based on *STXXL* vectors. Even for our Sibeyn implementation (where even a simple implementation incurs much less copying), introducing pipelining improved the running time by approximately 10% on average. The tunings to the PQ based version of Sibeyn suggested in Section 6.4, e.g. reducing the processed volume of edges in the PQ, turned out to be beneficial, lowering running time by an additional approximately 29% on average.

Approximate Counting

For the estimation of V using approximate counting algorithms, we tested the FM algorithm by Flajolet and Martin [68]. In essence, the FM algorithm computes for a given input stream an estimate of its number of distinct elements. For this, every input element is mapped by a hash function and incorporated into a later modified and returned proxy value. Due to the output variance being intolerably large, standard median-of-means techniques are employed which in turn, however, require more independent hash functions.

For several graph classes, we employed the FM algorithm in the sampling step of the Karger-Klein-Tarjan algorithm with an increasing number of hash functions. To accurately assess the returned estimates we separately ran the Karger-Klein-Tarjan algorithm with the same seed and explicitly counted the correct number of nodes in each sampling step.

We found that the number of hash functions needed in order to get a useful precision in the estimate was so high that it impacted the running time. Additionally, the errors in the estimation are two-sided, which fits badly with the fact that invoking a Union-Find based base case when not actually being semi-external will have disastrous effects on the running time. Combined, this made us decide not to include this method in the final experiments.

Karger-Klein-Tarjan

For the contraction steps in the Karger-Klein-Tarjan algorithm we tried both standard and randomized Borůvka steps, as well as the PQ based version of Sibeyn, and the latter proved to be the better option.

When varying the sampling parameter p in Karger-Klein-Tarjan, we observed a rather clear correlation (see Figure 6.2 and Figures 6.10 to 6.16): the best choice for both I/O volume and running time seems to be p equal to the inverse density V/E of the input graph, likely for the reasons conjectured in Section 6.4: a sampled subset of edges containing around V nodes will often by itself contract the node set considerably, while the left recursion will be cheap if this is achieved for small p, which may happen more often for high densities.

Also, when visualizing the recursion trees, a clear pattern was a balanced tree for this value of p, whereas quite strongly left-leaning and right-leaning trees appeared for larger and smaller values, respectively. Profiling of the distribution of time spent in the nodes of the recursion trees showed the root to be dominating, which is aligned with the analysis in [108]. Often, the contraction step was dominating (as can also be seen in Figure 6.2). There was also a small tendency for Karger-Klein-Tarjan to improve relative to the other algorithms when V grew compared to M (for fixed density and graph class). These observations (which are visible in the plots in Figures 6.10 to 6.17) inspired us to implement variants of the Karger-Klein-Tarjan which do not use contraction at the root, and adaptive variants which in all recursion tree nodes choose contractions only for low (estimated) density and also choose a sampling parameter close to V/E.

Choice of Contraction Target in Sibeyn

Perhaps our most interesting observation in phase one was the influence on Sibeyn of the choice of which neighbor to contract (see Section 6.4). We tried the choices of nearest, random, and farthest in node-ID order (i.e., the “timeline” in the time-forward processing by the PQ). Of these, the random choice intuitively can be expected to behave like the *MSF* variant of the algorithm (where each node must choose the neighbor of its lightest incident edge, and where the node IDs are randomly permuted). As exemplified by the first plot of Figure 6.6, where a message is a PQ entry (i.e., an edge in its original form or a later replaced form) each inducing $\mathcal{O}(1)$ PQ operations, the choice of nearest is by far the worst and was not considered again. On the other hand, farthest is always better than random (see rest of plots in Figure 6.6).

This effect was first studied in [167], where an expected bound of $\mathcal{O}(E \log\log(V))$ messages was claimed (but the proof omitted) for *Gilbert* graphs. As seen in Figure 6.3, we here verify that claim empirically, and also demonstrate that it does not hold for the random choice. Even more interesting, for random grid graphs and random hyperbolic graphs, the empirical evidence even suggests a better bound of expected $\mathcal{O}(E)$ (Figure 6.8 and Figure 6.9). These findings suggest that Sibeyn in practice is strictly faster for *CC* than for *MSF*, and that it for the former may often run in cost $\mathcal{O}(\mathrm{sort}(E))$. Additionally,

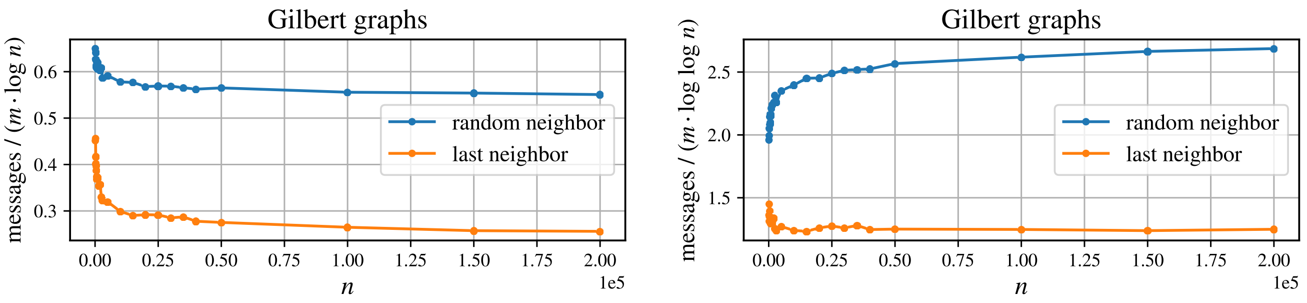


Figure 6.3: (Copy of Figure 6.7) Number of forwarded messages divided by $m \log n$ (left) or $m \log \log n$ (right) for Gilbert graphs for increasing values of n. The value p is chosen s.t. a density of five is fixed. In (left) we observe that the total number of produced messages is dominated by $m \log n$ whereas in (right) we see that the volume asymptotically matches with $m \log \log n$ if messages are forwarded to the last neighbor.

the bulk of the messages seem concentrated very late in the time-forward process, which in the external version is preempted by entering the semi-external Union-Find case, which in turn has lower overhead per edge/message than a PQ. Combined with the general simplicity of SIBEYN, these findings indicate that it may be very hard to surpass. For our final SIBEYN implementations, we naturally used the farthest neighbor choice.

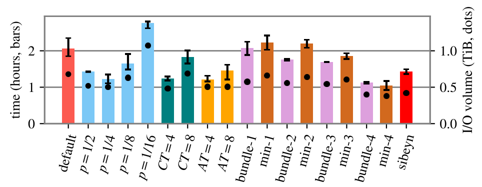
6.7.3 Phase 2 – Final Algorithms

For the second and final phase of experiments we selected the following algorithms (with implementation choices fixed as described above).

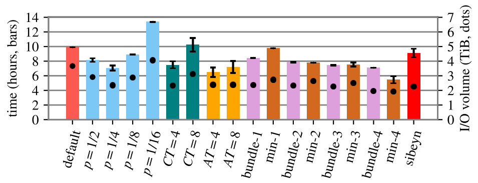
- KARGER-KLEIN-TARJAN in several versions: One with contractions in all recursion tree nodes and fixed sampling parameter $p = 1/2$. Four versions omitting contraction at the root of the recursion tree and having fixed sampling parameters of $1/2$, $1/4$, $1/8$, and $1/16$, respectively. Two adaptive versions which in each node of the recursion tree choose (among the values above) the sampling parameter closest to the estimated inverse density V/E of the input graph of the node, and also omit contraction if the estimated density E/V is below a fixed threshold of 4 or 8, respectively. Two similar adaptive versions where instead the threshold is 4 or 8, respectively, when the estimated V is close to the base case, but tends to 2 for growing V. These nine algorithms are denoted *default*, $p = 1/2$, $p = 1/4$, $p = 1/8$, $p = 1/16$, $CT = 4$, $CT = 8$, $AT = 4$, and $AT = 8$, respectively.

- SIBEYN's algorithm based on buckets, using Union-Find for solving *CC* in buckets, as described in [167] (where buckets are called bundles). We tried four increasing bucket sizes, all without and with relinking to minimize edges straddling buckets (Section 6.4 and [167]). These eight algorithms are denoted *bundle-x* and *min-x* for $x = 1, 2, 3, 4$.

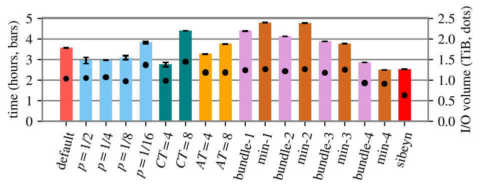
- The basic SIBEYN using a PQ. This algorithm is denoted *sibeyn*.



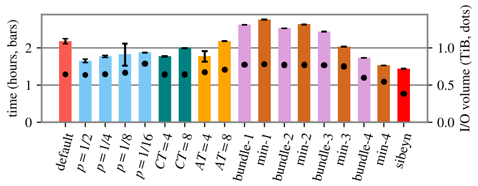
(a) Gilbert graph, smallest node set, $m/n = 5$



(b) Gilbert graph, largest node set, $m/n = 5$



(c) Random Geometric Graph



(d) Random Hyperbolic Graph

Figure 6.4: *(Copies of Figure 6.10b, Figure 6.13b, Figure 6.16d and Figure 6.17c) Running times and I/O volumes for two Gilbert graphs and the two largest generated RGG and RHG instances.*

Comparing Karger-Klein-Tarjan Variants

We find based on Figures 6.10 to 6.17 that among the Karger-Klein-Tarjan variants the adaptive ones are either winning or performing close to the best variant. This can be observed for all considered graph classes (see Figure 6.4 for an overview). In almost all cases, fixed contraction thresholds tend to perform better than adaptive ones. Further, setting the threshold to a small value seems preferable. This behaviour is consistent when increasing the number of nodes while keeping the density fixed (see for instance Figure 6.10b, Figure 6.11b, Figure 6.12b and Figure 6.13b) where it is clear that relative performances remain unchanged.

The good performance of these adaptive variants and the comparatively generally weak performance of the *default* variant support our claim that contractions can be intolerably costly.

Comparing Sibeyn Variants

We find that both versions of Sibeyn's algorithm are strong contenders. While the PQ based Sibeyn algorithm generally performs better on low density graphs (see Figure 6.4c and Figure 6.4d), its relative performance gets worse with increasing V (see Figure 6.4a, Figure 6.4b and Figures 6.10 to 6.13). Additionally, while the overall I/O volume may be near optimal (see Figure 6.4d), the achieved wall clock time does not always reflect this, indicating that the I/Os incurred by the PQ may be more costly than those for sorting.

In comparison, the bucket based Sibeyn algorithm performs consistently among the studied graph classes (see Figure 6.4 and Figures 6.10 to 6.17). We notice two clear trends, larger buckets generally perform better and adding relinking typically improves performance for graphs with higher densities.

6.8 Conclusion

The results of our experiments in phase two on the above set of algorithms can be seen in Figures 6.10 to 6.17. Sibeyn's algorithm is a strong contender. One reason is that it is very simple, using essentially only a priority queue (or repeated Union-Find in the bucket version). A tuned implementation of external priority queues can be highly efficient: our measurements on *STXXL* show that sorting by its priority queue is less than a factor of 2.5 slower than its sorting routine. Another reason is that for its *CC* variant, the choice of farthest neighbors seems to lower the number of messages generated to essentially linear (with the exact observed bound depending on the graph class) in E, which translates into a similar number of priority queue operations. Very few sorting and scanning steps on the input edge list can be performed by a competing algorithm before it will lose to Sibeyn.

Still, with the right tunings, the Karger-Klein-Tarjan algorithm can be implemented to be competitive in many cases. The best Karger-Klein-Tarjan variant often either wins over PQ based Sibeyn, but not bucket based Sibeyn, or vice versa. If nothing is known about the graph type and density, an adaptive variant such as $CT = 4$ may be a robust choice. In general, higher graph density seems to benefit Karger-Klein-Tarjan relative to Sibeyn. If choosing the bucket based Sibeyn variant, using the largest bucket size is clearly preferable (and often the min variant has a slight advantage). Borůvka's algorithm was not able to compete with neither Sibeyn nor Karger-Klein-Tarjan.

Natural future work suggested by this work include: 1) To investigate theoretically the observed positive effects on Sibeyn of the farthest neighbors choice. As demonstrated in Figures 6.7 to 6.9, different results seem plausible for different graph classes. 2) To compare empirically also the *MSF* versions of the algorithms.

Acknowledgements

Extensive calculations on the Goethe-HLR high-performance computer of the Goethe University Frankfurt were conducted for this research. The authors would like to acknowledge the CSC team for their support. We would also like to thank Peter Sanders for valuable feedback on an earlier draft of this paper.

Based upon work partially performed while attending AlgoPARC Workshop on Parallel Algorithms and Data Structures at the University of Hawaii at Manoa, organized by Nodari Sitchinava and in part supported by the National Science Foundation under Grant No. CCF-1930579. Gerth Stølting Brodal and Rolf Fagerberg have been supported by the Independent Research Fund Denmark under grants 9131-00113B and DFF-7014-00041, respectively. David Hammer, Ulrich Meyer, Manuel Penschuck, and Hung Tran have been supported by the Deutsche Forschungsgemeinschaft (DFG) under grants ME 2088/4-2 (SPP 1736 Algorithms for Big Data) and ME 2088/5-1.

Appendix 6.A Plots

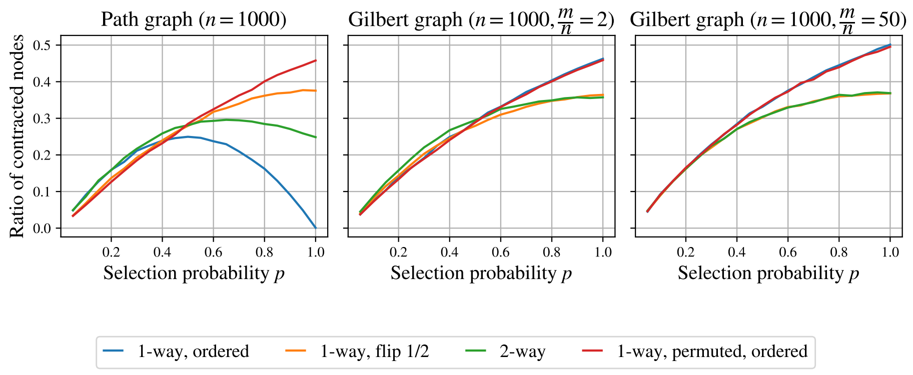


Figure 6.5: Contraction ratios achieved by variants of a randomized Borůvka step for varying p and varying edge representations. For Gilbert graphs, the contraction ratio increases with increasing selection probability p where the best candidates are the ordered variants. For path graphs, the variants without randomness peak and start to perform worse.

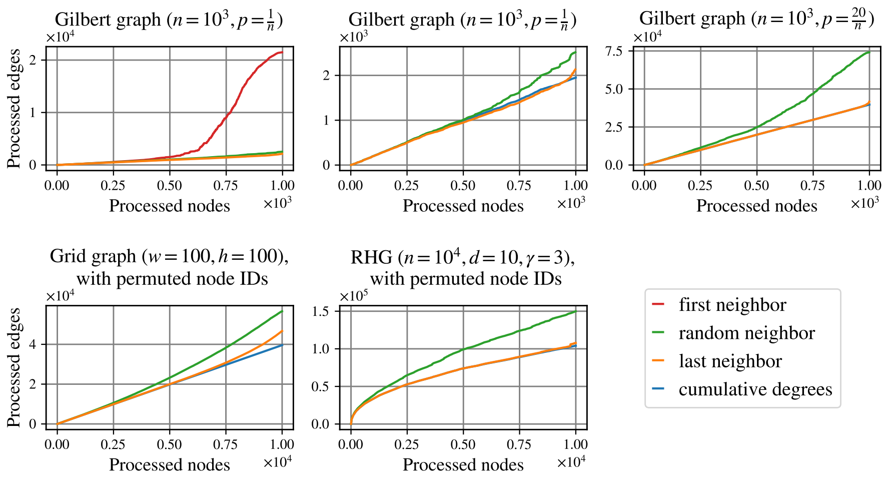


Figure 6.6: Message volume of forwarded messages for different graphs depending on the contraction strategy. Sibeyn's algorithm processes significantly more edges (priority queue messages) when messages are sent to the first neighbor (see first plot). In comparison, sending messages to the last neighbor produces volumes very close to the baseline (cumulative degrees) and always performs better than sending to a random neighbor.

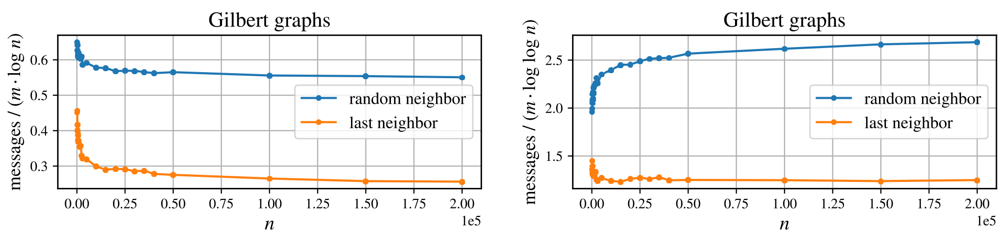


Figure 6.7: Number of forwarded messages divided by $m \log n$ (left) or $m \log \log n$ (right) for Gilbert graphs for increasing values of n. The value p is chosen s.t. a density of five is fixed. In (left) we observe that the total number of produced messages is dominated by $m \log n$ whereas in (right) we see that the volume asymptotically matches with $m \log \log n$ if messages are forwarded to the last neighbor.

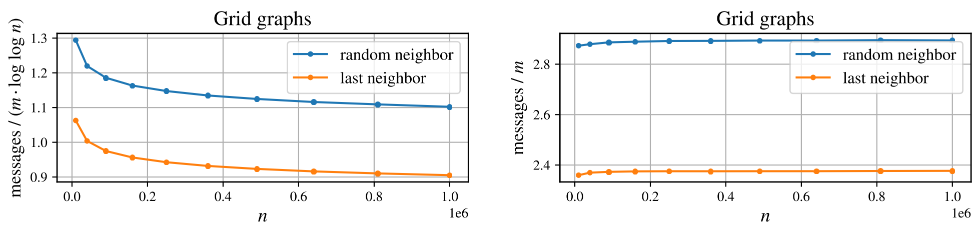


Figure 6.8: Number of forwarded messages divided by $m \log \log n$ (left) or m (right) for quadratic grid graphs for increasing values of n. By construction, these have density approximately two. In (left) we observe that the total number of produced messages is dominated by $m \log \log n$ whereas in (right) we see that the volume asymptotically matches with m.

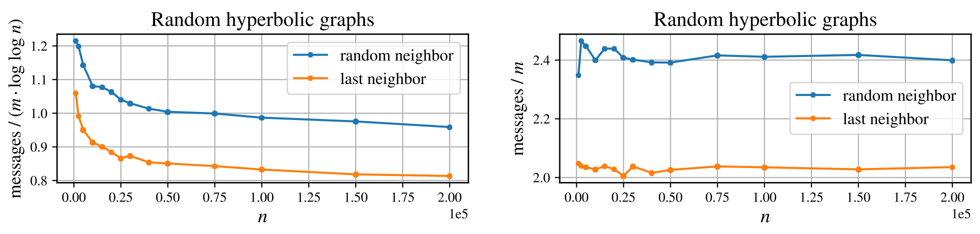
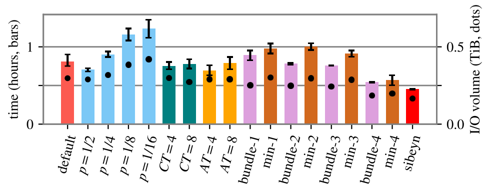
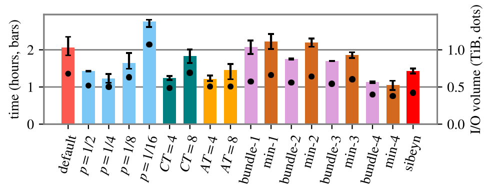


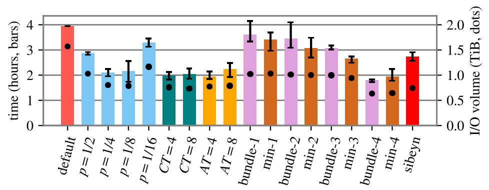
Figure 6.9: Number of forwarded messages divided by $m \log \log n$ (left) or m (right) for RHGs for increasing values of n. The degree parameter is set to 10 for all of these, yielding an approximate density of five. The degree exponent is set to 3. In (left) we observe that the total number of produced messages is dominated by $m \log \log n$ whereas in (right) we see that the volume asymptotically matches with m.



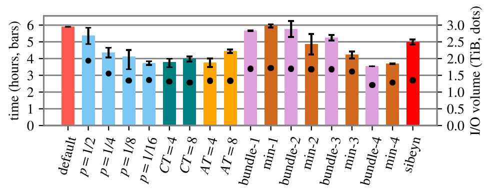
(a) $m = 1.34 \cdot 10^9$, $m/n = 2.04$



(b) $m = 3.36 \cdot 10^9$, $m/n = 5.00$



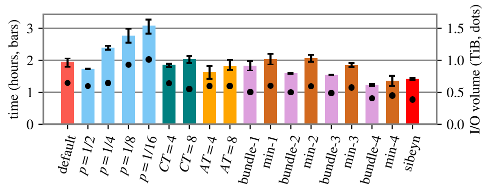
(c) $m = 6.71 \cdot 10^9$, $m/n = 10.00$



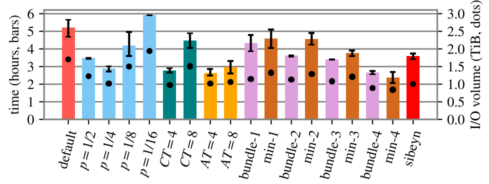
(d) $m = 1.34 \cdot 10^{10}$, $m/n = 20.00$

Figure 6.10: Running times and I/O volumes for $\mathcal{G}(n,p)$ graphs with a node set size of 5GiB and varying density. For $m/n = 20$, the *default* variant exceeded the local hard disk's capacity leading to a halt in the algorithm's execution. We thus only report the elapsed wall time up until that point.
The considered algorithms are in fixed order from left to right:

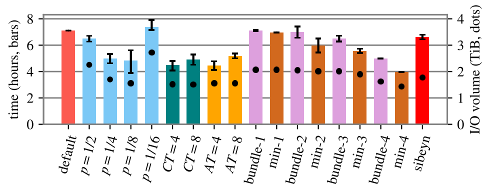
default:	fixed sampling $p = 1/2$, always contract
$p = 1/x$:	fixed sampling $p = 1/x$, always contract except in root
CT = *x*:	adaptive sampling, contract if estimated density below fixed threshold x
AT = *x*:	adaptive sampling, contract if estimated density below adaptive threshold x
bundle-x:	Sibeyn's algorithm based on buckets, without linking
min-x:	Sibeyn's algorithm based on buckets, with linking
sibeyn:	Sibeyn's algorithm based on priority-queues



(a) $m = 2.68 \cdot 10^9$, $m/n = 2.04$

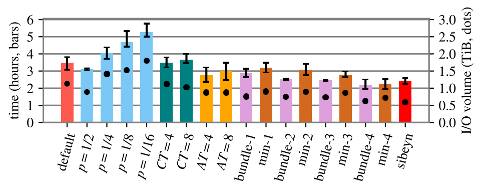


(b) $m = 6.71 \cdot 10^9$, $m/n = 5.00$

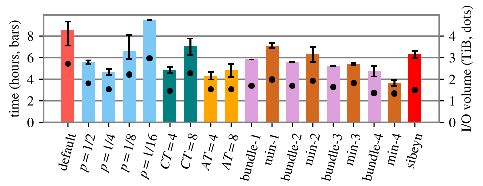


(c) $m = 1.34 \cdot 10^{10}$, $m/n = 10.00$

Figure 6.11: Running times and I/O volumes for $\mathcal{G}(n,p)$ graphs with a node set size of 10GiB and varying density. For $m/n = 10$, the *default* variant exceeded the local hard disk's capacity leading to a halt in the algorithm's execution. We thus only report the elapsed wall time up until that point.

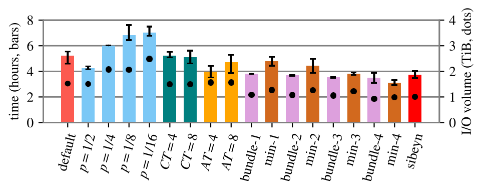


(a) $m = 4.03 \cdot 10^9$, $m/n = 2.04$

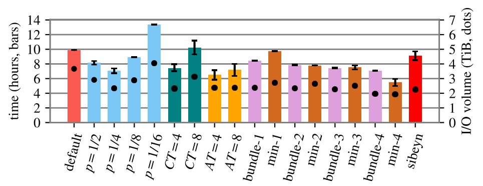


(b) $m = 1.01 \cdot 10^{10}$, $m/n = 5.00$

Figure 6.12: Running times and I/O volumes for $\mathcal{G}(n, p)$ graphs with a node set size of 15GiB and varying density.

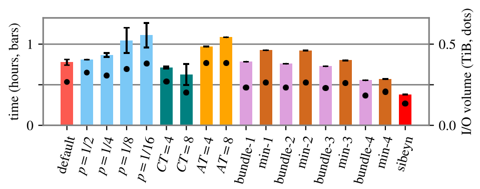


(a) $m = 5.37 \cdot 10^9$, $m/n = 2.04$

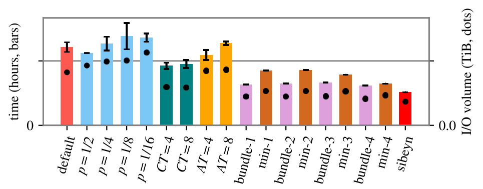


(b) $m = 1.34 \cdot 10^{10}$, $m/n = 5.00$

Figure 6.13: Running times and I/O volumes for $\mathcal{G}(n, p)$ graphs with a node set size of 20GiB and varying density.

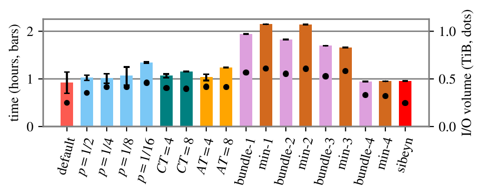


(a) $m = 1.34 \cdot 10^9$, $m/n = 2.00$

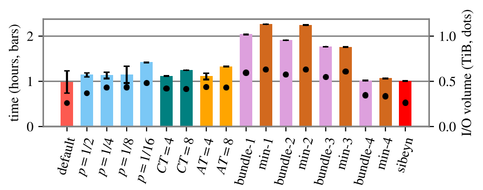


(b) $m = 1.34 \cdot 10^9$, $m/n = 1.00$

Figure 6.14: Running times and I/O volumes for (a) a grid graph with $(w, h) = (25{,}905, 25{,}905)$ and (b) a path graph. For both instances the parameters were chosen to generate a 20GiB graph. Node IDs are permuted.

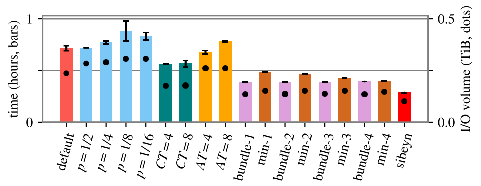


(a) $m = 3.89 \cdot 10^9$, $m/n = 11.96$

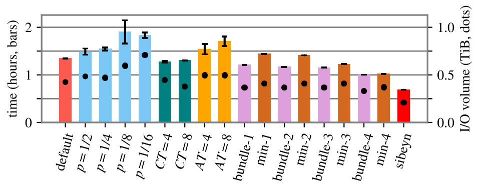


(b) $m = 4.05 \cdot 10^9$, $m/n = 12.00$

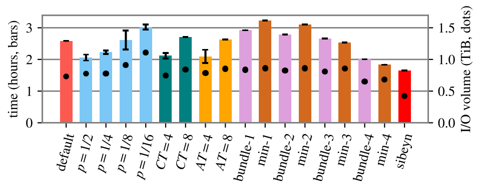
Figure 6.15: Running times and I/O volumes for cubes with the parameters (a) one layer and $(w, h, d) = (18{,}000, 18{,}000, 2)$ and (b) 100 layers and $(w, h, d) = (2600, 1300, 2)$.



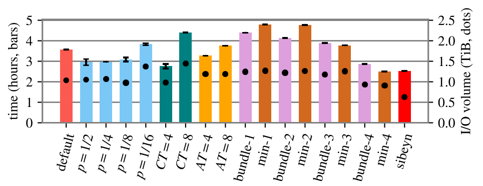
(a) $n = 9.18 \cdot 10^8$, $m = 1.04 \cdot 10^9$, $m/n = 1.14$



(b) $n = 1.06 \cdot 10^9$, $m = 2.62 \cdot 10^9$, $m/n = 2.46$

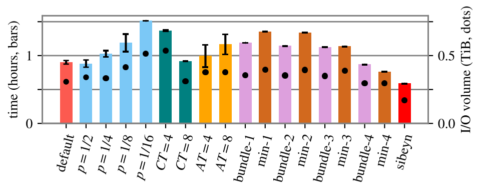


(c) $n = 1.07 \cdot 10^9$, $m = 5.48 \cdot 10^9$, $m/n = 5.11$

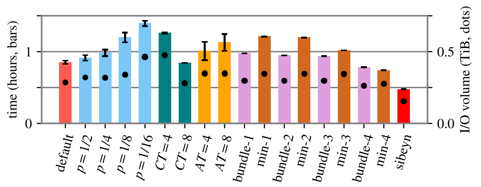


(d) $n = 1.07 \cdot 10^9$, $m = 8.14 \cdot 10^9$, $m/n = 7.59$

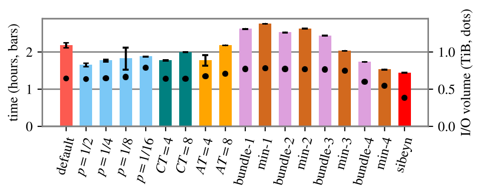
Figure 6.16: Running times and I/O volumes for RGGs with roughly $n = 2^{30}$ and varying density. Node IDs are permuted.



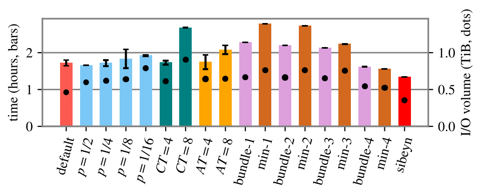
(a) $n = 6.54 \cdot 10^8$, $m = 2.61 \cdot 10^9$, $m/n = 3.99$, $\gamma = 3$



(b) $n = 6.46 \cdot 10^8$, $m = 2.36 \cdot 10^9$, $m/n = 3.65$, $\gamma = 4$



(c) $n = 6.70 \cdot 10^8$, $m = 5.57 \cdot 10^9$, $m/n = 8.30$, $\gamma = 3$



(d) $n = 6.70 \cdot 10^8$, $m = 5.22 \cdot 10^9$, $m/n = 7.80$, $\gamma = 4$

Figure 6.17: Running times and I/O volumes for RHGs with roughly $n = 2^{30}$, degree exponent $\gamma \in \{3, 4\}$ and varying density. Node IDs are permuted.

Certifying Induced Subgraphs in Large Graphs

joint work with U. Meyer, and K. Tsakalidis

We introduce I/O-optimal certifying algorithms for bipartite graphs, as well as for the classes of split, threshold, bipartite chain, and trivially perfect graphs. When the input graph is a class member, the certifying algorithm returns a certificate that characterizes this class. Otherwise, it returns a forbidden induced subgraph as a certificate for non-membership. On a graph with n vertices and m edges, our algorithms take optimal $\mathcal{O}(\mathrm{sort}(n + m))$ I/Os in the worst case or with high probability for bipartite chain graphs, and the certificates are returned in optimal I/Os. We give implementations for split and threshold graphs and provide an experimental evaluation.

This chapter is based on the peer-reviewed conference article [134]:

[134] U. Meyer, H. Tran, and K. Tsakalidis. Certifying induced subgraphs in large graphs. In C. Lin, B. M. T. Lin, and G. Liotta, editors, *WALCOM: Algorithms and Computation - 17th Int. Conference and Workshops, WALCOM 2023, Hsinchu, Taiwan, March 22-24, 2023, Proceedings*, volume 13973 of *Lecture Notes in Computer Science*, pages 229–241. Springer, 2023. doi:10.1007/978-3-031-27051-2_20 .

My contribution

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

7.1 Introduction

Certifying algorithms [127] ensure the correctness of an algorithm's output without having to trust the algorithm itself. The user of a certifying algorithm inputs x and receives the output y with a *certificate* or *witness* w that proves that y is a correct output for input x. Certifying the bipartiteness of a graph is a textbook example where the returned witness w is a bipartition of the vertices (YES-certificate) or an induced *odd-length cycle* subgraph, i.e. a cycle of vertices with an odd number of edges (NO-certificate).

Emerging big data applications need to process large graphs efficiently. Standard models of computation in internal memory (RAM, pointer machine) do not capture the algorithmic complexity of processing graphs with size that exceed the main memory. The *I/O model* by Aggarwal and Vitter [1] is suitable for studying large graphs stored in an external memory hierarchies, e.g. comprised of cache, RAM and hard disk memories. The input data elements are stored in *external memory* (EM) packed in *blocks* of at most B elements and computation is free in *main memory* for at most M elements. The *I/O-complexity* is measured in *I/O-operations* (*I/Os*) that transfer a block from external to main memory and vice versa. *I/O-optimal* external memory algorithms for sorting and scanning n elements take $\text{sort}(n) = \mathcal{O}\Big((n/B)\log_{M/B}(n/B)\Big)$ I/Os and $\text{scan}(n) = \mathcal{O}(n/B)$ I/Os, respectively.

7.1.1 Previous Work

Certifying bipartiteness in internal memory takes time linear in the number of edges by any traversal of the graph. However, all known external memory breadth-first search [2] and depth-first search [42] traversal algorithms take suboptimal $\omega(\text{sort}(n+m))$ I/Os for an input graph with n vertices and m edges. Heggernes and Kratsch [96] present optimal internal memory algorithms for certifying whether a graph belongs to the classes of split, threshold, bipartite chain, and trivially perfect graphs. They return in linear time a YES-certificate characterizing the corresponding class or a forbidden induced subgraph of the class (NO-certificate). The YES- and NO-certificates are authenticated in linear and constant time, respectively. A straightforward application to the I/O model leads to suboptimal certifying algorithms since graph traversal algorithms in external memory are much more involved and no worst-case efficient algorithms are known.

7.1.2 Our Results

We present I/O-optimal certifying algorithms for *split*, *threshold*, *bipartite chain*, and *trivially perfect* graphs. All algorithms return in the membership case, a YES-certificate w characterizing the graph class, or a $\mathcal{O}(1)$-size NO-certificate in the non-membership case. As a byproduct, we show how to efficiently certify graph *bipartiteness* in external memory using standard I/O-efficient techniques. Additionally, we perform experiments for split and threshold graphs showing scaling beyond the size of main memory.

7.1.3 Preliminaries and Notation

For a graph $G = (V, E)$, let $n = |V|$ and $m = |E|$ denote the number of vertices V and edges E, respectively. For a vertex $v \in V$ we denote by $N(v)$ the *neighborhood* of v and by $N[v] = N(v) \cup \{v\}$ the *closed neighborhood* of v. The *degree* $\deg(v)$ of a vertex v is given by $\deg(v) = |N(v)|$. A vertex is called *simplicial* if $N(v)$ is a clique and *universal* if $N[v] = V$.

Graph Substructures and Orderings The subgraph of G that is induced by a subset $A \subseteq V$ of vertices is denoted by $G[A]$. The *substructure* (subgraph) of a cycle on k vertices is denoted by C_k and of a path on k vertices is denoted by P_k. The substructure $2K_2$ is a graph that is isomorphic to the following constant size graph: $(\{a, b, c, d\}, \{ab, cd\})$.

Henceforth we refer to different types of orderings of vertices: an ordering $(v_1, \ldots, v_n)$ is a (i) *perfect elimination ordering* (*peo*) if v_i is simplicial in $G[\{v_i, v_{i+1}, \ldots, v_n\}]$ for $i \in \{1, \ldots, n\}$, and a (ii) *universal-in-a-component-ordering* (*uco*) if v_i is universal in its connected component in $G[\{v_i, v_{i+1}, \ldots, v_n\}]$ for $i \in \{1, \ldots, n\}$. For a subset $X = \{v_1, \ldots, v_k\}$, we call $(v_1, \ldots, v_k)$ a *nested neighborhood ordering* (*nno*) if $(N(v_1) \setminus X) \subseteq (N(v_2) \setminus X)) \subseteq \ldots \subseteq (N(v_k) \setminus X)$. Finally for any ordering, we partition $N(v_i)$ into lower and higher ranked neighbors, respectively, $L(v_i) = \{x \in N(v_i) :$ v_i is ranked lower than $x\}$ and $H(v_i) = \{x \in N(v_i) : v_i$ is ranked higher than $x\}$.

Graph Representation We assume an *adjacency row representation* where the graph $G = (V, E)$ is represented by two arrays $P = [\ P_i\ ]_{i=1}^{n}$ and $E = [\ u_i\ ]_{i=1}^{m}$. The neighbors of a vertex v_i are then given by the vertices at position $P[v_i]$ to $P[v_{i+1}] - 1$ in E. We use the adjacency row representation to allow for efficient scanning of G: (i) computing k consecutive adjacency lists consisting of m edges requires $\mathcal{O}(\mathrm{scan}(m))$ I/Os and (ii) computing the degrees of k consecutive vertices requires $\mathcal{O}(\mathrm{scan}(k))$ I/Os.

Time Forward Processing *Time Forward Processing* (*TFP*) is a generic technique to manage data dependencies of external memory algorithms [123]. These dependencies are typically modeled by a directed acyclic graph $G = (V, E)$ where every vertex $v_i \in V$ models the computation of z_i and an edge $(v_i, v_j) \in E$ indicates that z_i is required for the computation of z_j.

Computing a solution then requires the algorithm to traverse G according to some topological order $\prec_T$ of the vertices V. The *TFP* technique achieves this in the following way: after z_i has been calculated, the algorithm inserts a message $\langle v_j, z_i \rangle$ into a minimum priority-queue data structure for every successor $(v_i, v_j) \in E$ where the items are sorted by the recipients according to $\prec_T$. By construction, v_j receives all required values z_i of its predecessors $v_i \prec_T v_j$ as messages in the data structure. Since these predecessors already removed their messages from the data structure, items addressed to v_j are currently the smallest elements in the data structures and thus can be dequeued with a delete-minimum operation. By using suitable external memory priority-queues [14], *TFP* incurs $\mathcal{O}(\mathrm{sort}(k))$ I/Os, where k is the number of messages sent.

7.2 Certifying Graph Classes in External Memory

7.2.1 Certifying Split Graphs in External Memory

A split graph is a graph that can be partitioned into two sets of vertices (K, I) where K and I induce a clique and an independent set, respectively. The partition (K, I) is called the *split partition*. They are additionally characterized by the forbidden induced substructures $2K_2, C_4$ and C_5, meaning that any vertex subset of a split graph cannot induce these structures [93]. Since split graphs are a subclass of chordal graphs, there exists a peo of the vertices for every split graph. In fact, any non-decreasing degree ordering of a split graph is a peo [96].

Our algorithm adapts the internal memory certifying algorithm of Heggernes and Kratsch [96] to external memory by adopting TFP. As output it either returns the split partition (K, I) as a YES-certificate or one of the forbidden substructures C_4, C_5 or $2K_2$ as a NO-certificate.

First, we compute a non-decreasing degree ordering $\alpha = (v_1, \ldots, v_n)$ and relabel[1] the graph according to α. Thereafter it checks whether α is a peo in $\mathcal{O}(\text{sort}(n + m))$ I/Os by Proposition 7.1. In the non-membership case, the algorithm returns three vertices v_j, v_k, v_i where $\{v_i, v_j\}, \{v_i, v_k\} \in E$ but $\{v_j, v_k\} \notin E$ and $i < j < k$, violating that v_i is simplicial in $G[\{v_i, \ldots, v_n\}]$. In order to return any of the forbidden substructures we find additional vertices that complete the induced subgraphs. Note that (v_k, v_i, v_j) already forms a P_3 and may extend to a C_4 if $N(v_k) \cap N(v_j)$ contains a vertex $z \neq v_i$ that is not adjacent to v_i. Computing $(N(v_k) \cap N(v_j)) \setminus N(v_i)$ requires scanning the adjacencies of $\mathcal{O}(1)$ many vertices totaling to $\mathcal{O}(\text{scan}(n))$ I/Os. If $(N(v_k) \cap N(v_j)) \setminus N(v_i)$ is empty we try to extend the P_3 to a C_5 or output a $2K_2$ otherwise. To do so, we find vertices $x \neq v_i$ and $y \neq v_i$ for which $\{x, v_j\}, \{y, v_k\} \in E$ but $\{x, v_k\}, \{y, v_j\} \notin E$ that are also not adjacent to v_i, i.e. $\{x, v_i\}, \{y, v_i\} \notin E$. Both x and y exist due to the ordering α [96] and are found using $\mathcal{O}(1)$ scanning steps requiring $\mathcal{O}(\text{scan}(n)$ I/Os. If $\{x, y\} \in E$ then (v_j, v_i, v_k, y, x) is a C_5, otherwise $G[\{v_j, x, v_k, y\}]$ constitutes a $2K_2$. Determining whether $\{x, y\} \in E$ requires scanning $N(x)$ and $N(y)$ using $\mathcal{O}(\text{scan}(n))$ I/Os.

In the membership case, α is a peo and the algorithm proceeds to verify first the clique K and then the independent set I of the split partition (K, I). Note that for a split graph the maximum clique of size k must consist of the k-highest ranked vertices in α [96] where k can be computed using $\mathcal{O}(\text{sort}(m))$ I/Os by Proposition 7.2. Therefore, it suffices to verify for each of the k candidates v_i whether it is connected to $\{v_{i+1}, \ldots, v_n\}$ since the graph is undirected. For a sorted sequence of edges relabeled by α, we check this property using $\mathcal{O}(\text{scan}(m))$ I/Os. If we find a vertex $v_i \in \{v_{n-k+1}, \ldots, v_n\}$ where $\{v_i, v_j\} \notin E$ with $i < j$ then $G[\{v_i, \ldots, v_n\}]$ already does not constitute a clique and we have to return a NO-certificate. Since the maximum clique has size k, there are k vertices with degree at least $k - 1$. By these degree constraints there must exist an edge $\{v_i, x\} \in E$ where $x \in \{v_1, \ldots, v_{i-1}\}$ [96]. Additionally, it holds that $\{x, v_j\} \notin E$ and

[1]If a vertex v_i has rank k in α it will be relabeled to v_k.

Algorithm 8: Recognizing Perfect Elimination in EM

```
Data: edges $E$ of graph $G$, non-decreasing degree ordering $\alpha = (v_1, \dots, v_n)$
Output: bool whether $\alpha$ is a peo, three invalidating vertices $\{v_i, v_j, v_k\}$ if not
1  Sort $E$ and relabel according to $\alpha$
2  for $i = 1, \dots, n$ do
3      Retrieve $H(v_i)$ from $E$
4      if $H(v_i) \neq \emptyset$ then
5          Let $u$ be the smallest successor of $v_i$ in $H(v_i)$
6          for $x \in H(v_i) \setminus \{u\}$ do
7              PQ.PUSH($\langle u, x, v_i \rangle$)            // inform $u$ of $x$ coming from $v_i$
8
9      while $\langle v, v_k, v_j \rangle \leftarrow$ PQ.TOP() where $v = v_i$ do     // for each message to $v_i$
10         if $v_k \notin H(v_i)$ then                  // $v_i$ does not fulfill peo property
11             return FALSE, $\{v_i, v_j, v_k\}$
12         PQ.POP()
13 return TRUE
```

there exists an edge $\{z, v_j\} \in E$ where $z \in \{v_1, \dots, v_{i-1}\}$ that cannot be connected to x, i.e. $\{x, z\} \notin E$ [96]. Thus, we first scan the adjacency lists of v_i and v_j to find x and z in $\mathcal{O}(\text{scan}(n))$ I/Os and return $G[\{v_i, v_j, x, z\}]$ as the $2K_2$ NO-certificate. Otherwise let $K = \{v_{n-k+1}, \dots, v_n\}$.

Lastly, the algorithm verifies whether the remaining vertices form an independent set. We verify that each candidate v_i is not connected to $\{v_{i+1}, \dots, v_{n-k}\}$, since the graph is undirected. For this, it suffices to scan over $n-k$ consecutive adjacency lists in $\mathcal{O}(\text{scan}(m))$ I/Os. More precisely, we scan the adjacency lists from v_{n-k} to v_1 and in case an edge $\{v_i, v_j\}$ where $i < j \leq n-k$ is found we find two more vertices to again complete a $2K_2$. For the first occurrence of such a vertex v_i, we remark that $\{v_{i+1}, \dots, v_{n-k}\}$ and $\{v_{n-k+1}, \dots, v_n\}$ form an independent set and a clique, respectively. Therefore there exists a vertex $y \in K$ that is adjacent to x but not to v_i [96]. We find y by scanning $N(x)$ and $N(v_i)$ in $\mathcal{O}(\text{scan}(n))$ I/Os. To complete the $2K_2$ we similarly find $z \in N(y) \setminus (N(x) \cup N(y_i))$ in $\mathcal{O}(\text{scan}(n))$ I/Os which is guaranteed to exist [96].

Proposition 7.1. Verifying that a non-decreasing degree ordering $\alpha = (v_1, \dots, v_n)$ of a graph G with n vertices and m edges is a perfect elimination ordering requires $\mathcal{O}(\text{sort}(n+m))$ I/Os. ◀

Proof. We follow the approach of [82, Theorem 4.5] and adapt it to the external memory using *TFP*, see Algorithm 8.

After relabeling and sorting the edges by α we iterate over the vertices in the order given by α. For a vertex v_i the set of neighbors $N(v_i)$ needs to be a clique. In order to verify this for all vertices, for a vertex v_i we first retrieve $H(v_i)$. Then let $u \in H(v_i)$ be the smallest ranked neighbor according to α. In order for v_i to be simplicial, u needs to be adjacent to all vertices of $H(v_i) \setminus \{u\}$. In *TFP*-fashion we insert a message $\langle u, w \rangle$

Algorithm 9: Maximum Clique Size for Chordal Graphs in EM

```
   Data: edges $E$ of input graph $G$, peo $\alpha = (v_1, \ldots, v_n)$
   Output: maximum clique size $\chi$
1  Sort $E$ and relabel according to $\alpha$
2  $\chi \leftarrow 0$
3  for $i = 1, \ldots, n$ do
4      Retrieve $H(v_i)$ from $E$                                  // scan E
5      if $H(v_i) \neq \emptyset$ then
6          Let $u$ be the smallest successor of $v_i$ in $H(v_i)$
7          PQ.push($\langle u, |H(v_i)| - 1\rangle$)        // $v_i$ simplicial $\Rightarrow G[N(v_i)]$ is clique
8      $S(v_i) \leftarrow -\infty$
9      while $\langle v, S\rangle \leftarrow$ PQ.top() where $v = v_i$ do
10         $S(v_i) \leftarrow \max\{S(v_i), S\}$                // compute maximum over all
11         PQ.pop()
12     $\chi \leftarrow \max\{\chi, S(v_i)\}$
13 return $\chi$
```

into a priority-queue where $w \in H(v_i) \setminus \{u\}$ to inform u of every vertex it should be adjacent to. Conversely, after sending all adjacency information, we retrieve for v_i all messages $\langle v_i, -\rangle$ directed to v_i and check that all received vertices are indeed neighbors of v_i.

Relabeling and sorting the edges takes $\mathcal{O}(\text{sort}(m))$ I/Os. Every vertex v_i inserts at most all its neighbors into the priority-queue totaling up to $\mathcal{O}(m)$ messages which requires $\mathcal{O}(\text{sort}(m))$ I/Os. Checking that all received vertices are neighbors only requires a scan over all edges since vertices are handled in non-descending order by α. □

Proposition 7.2. Computing the size of a maximum clique in a split graph requires $\mathcal{O}(\text{sort}(m))$ I/Os. ◀

Proof. Note that split graphs are both chordal and co-chordal [93]. For chordal graphs, computing the size of a maximum clique in internal memory takes linear time [82, Theorem 4.17] and is easily convertible to an external memory algorithm using $\mathcal{O}(\text{sort}(m))$ I/Os. To do so, we simulate the data accesses of the internal memory variant using priority-queues to employ *TFP*, see Algorithm 9. Instead of updating each $S(v_i)$ value immediately, we delay its consecutive computation by sending a message $\langle v_i, S\rangle$ to v_i to inform v_i, that v_i is part of a clique of size S. After collecting all messages, the overall maximum is computed and the global value of the currently maximum clique is updated if necessary. □

By the above description it follows that split graphs can be certified using $\mathcal{O}(\text{sort}(n+m))$ I/Os which we summarize in Lemma 7.3.

Lemma 7.3. A graph with n vertices and m edges stored in external memory is certified whether it is a split graph or not in $\mathcal{O}(\text{sort}(n+m))$ I/Os. In the membership case the algorithm returns in $\mathcal{O}(\text{scan}(K+I))$ I/Os the split partition (K, I) as the YES-certificate, and otherwise it returns a $\mathcal{O}(1)$-size NO-certificate. ◀

7.2.2 Certifying Threshold Graphs in External Memory

Threshold graphs [53, 82, 121] are split graphs with the additional property that the independent set I of the split partition (K, I) has an nno. Its corresponding forbidden substructures are $2K_2$, P_4 and C_4. Alternatively, threshold graphs can be characterized by a graph generation process: repeatedly add universal or isolated vertices to an initially empty graph. Conversely, by repeatedly removing universal and isolated vertices from a threshold graph the resulting graph must be the empty graph. In comparison to certifying split graphs, threshold graphs thus require additional steps.

First, the algorithm certifies whether the input is a split graph. In the non-membership case, if the returned NO-certificate is a C_5 we extract a P_4 otherwise we return the substructure immediately. For the membership case, we recognize whether the input is a threshold graph by repeatedly removing universal and isolated vertices using the previously computed peo α in $\mathcal{O}(\text{sort}(m))$ I/Os by Proposition 7.4. If the remaining graph is empty, we return the independent set I with its non-decreasing degree ordering. Note that after removing a universal vertex v_i, vertices with degree one become isolated. Since low-degree vertices are at the front of α, an I/O-efficient algorithm cannot determine them on-the-fly after removing a high-degree vertex. Therefore pre-processing is required. For every vertex v_i we compute the number of vertices $S(v_i)$ that become isolated after the removal of $\{v_i, \dots, v_n\}$. To do so, we iterate over α in non-descending order and check for v_i with $L(v_i) = \emptyset$. Since v_i has no lower ranked neighbors, it would become isolated after removing all vertices in $H(v_i)$, in particular when the successor with smallest index $v_j \in H(v_i)$ is removed. We save v_j in a vector S and sort S in non-ascending order. The values $S(v_n), \dots, S(v_1)$ are now accessible by a scan over S to count the occurrences of each v_j in $\mathcal{O}(\text{scan}(m))$ I/Os.

In the non-membership case, there must exist a P_4 since the input is split and cannot contain a C_4 or a $2K_2$. We can delete further vertices from the remaining graph that cannot be part of a P_4. For this, let $K' \subset K$ and $I' \subset I$ be the remaining vertices of the split partition. Any $v \in K'$ where $N(v) \cap I' = \emptyset$ and any $v \in I'$ where $N(v) \cap K' = K'$ cannot be part of a P_4 [96] and can therefore be deleted. We proceed by considering and removing vertices of K by non-descending degree and vertices of I by non-ascending degree. After this process, we retrieve the highest-degree vertex v in I where there exists $\{v, y\} \notin E$ and $\{y, z\} \in E$ where $y \in K$ and $z \in I$ [96]. Additionally, there is a neighbor $w \in K$ of v for which $\{w, z\} \notin E$ [96] and we return the P_4 given by $G[\{v, w, y, z\}]$. Finding the P_4 therefore only requires $\mathcal{O}(\text{scan}(n+m))$ I/Os.

Proposition 7.4. Verifying that a non-decreasing degree ordering $\alpha = (v_1, \dots, v_n)$ of a graph G with n vertices and m edges emits an empty graph after repeatedly removing universal and isolated vertices requires $\mathcal{O}(\text{sort}(n) + \text{scan}(m))$ I/Os. ◀

Proof. Generating the values $S(v_n), \dots, S(v_1)$ requires a scan over all adjacency lists in non-descending order and sorting S which takes $\mathcal{O}(\text{sort}(n) + \text{scan}(m))$ I/Os. Afte

Algorithm 10: Recognizing Threshold Graphs for Split Graphs in EM

```
   Data: edges $E$ of split graph $G$, max. clique size $k$, peo $\alpha = (v_1, \dots, v_n)$
   Output: bool whether $G$ is threshold
1  Sort $E$ and relabel according to $\alpha$
2  Vector S
3  for $i = 1, \dots, n$ do
4      if $L(v_i) = \emptyset$ then
5          Let $v_j$ be the smallest successor of $v_i$ in $H(v_i)$
6          S.PUSH($v_j$)            // $v_i$ would be isolated after deleting $\{v_j, \dots, v_n\}$
7  Sort S in non-ascending order
8  $n_{\text{del}} \leftarrow 0$                // number of deleted universal/isolated vertices
9  for $i = n, \dots, 1$ do
10     if $L(v_i) \neq \emptyset$ then       // $v_i$ not isolated in $G[\{v_1, \dots, v_n\}]$
11         if $|L(v_i)| < (n-1) - n_{del}$ then          // $v_i$ not universal
12             return FALSE
13         $n_{\text{del}} \leftarrow n_{\text{del}} + 1 +$ occurrences of $v_i$      // $v_i$ removed, scan S
14 return TRUE
```

pre-processing, the algorithm only requires a reverse scan over the degrees $d_n, \dots, d_1$, see Algorithm 10. We iterate over α in reverse order, where for each v_i we check whether $L(v_i) = \emptyset$. If v_i is not isolated it must be universal. Therefore we compare its current degree $\deg(v_i)$ with the value $(n-1) - n_{\text{del}}$ where $n_{\text{del}} = \sum_{j=j+1}^{n} S(v_j)$. All operations take $\mathcal{O}(\text{scan}(m))$ I/Os in total. □

We summarize our findings for threshold graphs in Lemma 7.5.

Lemma 7.5. A graph with n vertices and m edges stored in external memory is certified whether it is a threshold graph or not in $\mathcal{O}(\text{sort}(n+m))$ I/Os. In the membership case the algorithm returns in $\mathcal{O}(\text{scan}(\beta))$ I/Os a nested neighborhood ordering β as the YES-certificate, and otherwise it returns a $\mathcal{O}(1)$-size NO-certificate. ◀

Proof. Certifying that the input graph is a split graph requires $\mathcal{O}(\text{sort}(n+m))$ I/Os by Lemma 7.3. If it is, we check if the input is a threshold graph directly by checking whether the graph is empty after repeatedly removing universal and isolated vertices in $\mathcal{O}(\text{sort}(m))$ I/Os by Proposition 7.4. Otherwise we have to find a P_4, since the input is a split but not a threshold graph. Hence, this step requires $\mathcal{O}(\text{scan}(n+m))$ I/Os and the total I/Os are $\mathcal{O}(\text{sort}(n+m))$. □

7.2.3 Certifying Trivially Perfect Graphs in External Memory

Trivially perfect graphs have no vertex subset that induces a P_4 or a C_4 [82]. In contrast to split graphs, any non-increasing degree ordering of a trivially perfect graph is a uco [96]. In fact, this is a one-to-one correspondence: a non-increasing sorted degree sequence of a graph is a uco iff the graph is trivially perfect [96].

Algorithm 11: Recognizing Universal-in-a-Component Orderings in EM

```
Data: edges $E$ of graph $G$, non-increasing degree ordering $\gamma = (v_1, \dots, v_n)$
Output: bool whether $\gamma$ is a uco
1  Sort $E$ and relabel according to $\gamma$
2  for $i = 1, \dots, n$ do
3      Vector L = [0]                                          // initialize with 0
4      while $\langle v, v_j, \ell \rangle \leftarrow$ PQ.TOP() where $v = v_i$ do     // $v_i$'s received labels
5          L.PUSH($\ell$)
6          PQ.POP()
7      for $i = 1, \dots,$L.size$/2$ do                          // L.SIZE() is even
8          if L[$2i$] $\neq$ L[$2i$+1] and L.size $> 1$ then      // mismatch / anomaly
9              return FALSE
10     $\ell(v_i) \leftarrow$ L[L.SIZE()]                            // assign label of $v_i$
11     Retrieve $H(v_i)$ from $E$                                 // scan E
12     for $u \in H(v_i)$ do
13         PQ.PUSH($\langle u, v_i, \ell(v_i) \rangle$)
14         PQ.PUSH($\langle u, v_i, i \rangle$)
15 return TRUE
```

In external memory this can be verified using *TFP* by adapting the algorithm in [96], see Algorithm 11. After computing a non-increasing degree ordering γ the algorithm relabels the edges of the graph according to γ and sorts them. Now we iterate over the vertices in non-descending order of γ, process for each vertex v_i its received messages and relay further messages forward in time.

Initially all vertices are labeled with 0. Then, at step i vertex v_i checks that all adjacent vertices $N(v_i)$ have the same label as v_i. After this, v_i relabels each vertex $u \in N(v_i)$ with its own index i and is then removed from the graph. In the external memory setting we cannot access labels of vertices and relabel them on-the-fly but rather postpone the comparison of the labels to the adjacent vertices instead. To do so, v_i forwards its own label $\ell(v_i)$ to $u \in H(v_i)$ by sending two messages $\langle u, v_i, \ell(v_i) \rangle$ and $\langle u, v_i, i \rangle$ to u, signaling that u should compare its own label to v_i's label $\ell(v_i)$ and then update it to i. Since the label of any adjacent vertex is changed after processing a vertex, when arriving at vertex v_j an odd number of messages will be targeted to v_j, where the last one corresponds to its actual label at step j. Then, after collecting all received labels, we compare disjoint consecutive pairs of labels and check whether they match. In the membership case, we do not find any mismatch and return γ as the YES-certificate. Otherwise, we have to return a P_4 or C_4.

In the description of [96] the authors stop at the first anomaly where v_i detects a mismatch in its own label and one of its neighbors. We simulate the same behavior by writing out every anomaly we find, e.g. that v_j does not have the expected label of v_i via an entry $\langle v_i, v_j, k \rangle$ where k denotes the found label of v_j. After sorting the entries, we find the earliest anomaly $\langle v_i, v_j, k \rangle$ with the largest label k of v_i's neighbors. Since v_j received the label k from v_k, but v_i did not, it is clear that v_k is not universal in its connected component in $G[\{v_k, v_{k+1}, \dots, v_n\}]$ and we thus return a P_4 or C_4. Note that (v_k, v_j, v_i) already constitutes a P_3 where $\deg(v_k) \geq \deg(v_j)$, because v_j received the label k. Since v_j is adjacent to both v_k and v_i and $\deg(v_k) \geq \deg(v_j)$, there must exist a vertex $x \in N(v_k)$ where $\{v_j, x\} \notin E$. Thus, $G[\{v_k, v_j, v_i, x\}]$ is a P_4 if $\{v_i, x\} \notin E$ and a C_4 otherwise. Finding x and determining whether the forbidden substructure is a P_4 or a C_4 requires scanning $\mathcal{O}(1)$ adjacency lists in $\mathcal{O}(\text{scan}(n))$ I/Os.

Proposition 7.6. Verifying that a non-increasing degree ordering $\gamma = (v_1, \dots, v_n)$ of a graph G with n vertices and m edges is a universal-in-a-component-ordering requires $\mathcal{O}(\text{sort}(m))$ I/Os. ◀

Proof. Every vertex v_i receives exactly two messages per neighbor in $L(v_i)$ and verifies that all consecutive pairs of labels match. Then, either the label i is sent to each higher ranked neighbor of $H(v_i)$ via *TFP* or it is verified that γ is not a uco. Since at most $\mathcal{O}(m)$ messages are inserted, the resulting overall complexity is $\mathcal{O}(\text{sort}(m))$ I/Os. Correctness follows from [96] since Algorithm 11 performs the same operations but only delays the label comparisons. □

We again summarize our results in Lemma 7.7.

Lemma 7.7. A graph with n vertices and m edges stored in external memory is certified whether it is a trivially perfect graph or not in $\mathcal{O}(\text{sort}(n + m))$ I/Os. In the membership case the algorithm returns in $\mathcal{O}(\text{scan}(\gamma))$ I/Os the universal-in-a-component ordering γ as the YES-certificate, and otherwise it returns a $\mathcal{O}(1)$-size NO-certificate. ◀

7.2.4 Certifying Bipartite Chain Graphs in External Memory

Bipartite chain graphs are bipartite graphs where one part of the bipartition has an nno [184] similar to threshold graphs. Interestingly, for chain graphs one side of the bipartition exhibits this property if and only if both partitions do [184]. Its forbidden induced substructures are $2K_2, C_3$ and C_5. By definition, bipartite chain graphs are bipartite graphs which therefore requires I/O-efficient bipartiteness testing.

We follow the linear time internal memory approach of [96] with slight adjustments to accommodate the external memory setting. First, we check whether the input is indeed a bipartite graph. Instead of using breadth-first search which is very costly in external memory, even for constrained settings [2], we can use a more efficient approach with spanning trees which is presented in Lemma 7.8. In case the input is not connected, we simply return two edges of two different components as the $2K_2$. If the graph is connected, we proceed to verify that the graph is bipartite and return a NO-certificate in the form of a C_3, C_5 or $2K_2$ in case it is not. In order to find a C_3, C_5 or $2K_2$ some modifications to Lemma 7.8 are necessary. Essentially, the algorithm instead returns a minimum odd cycle that is built from T and a single non-tree edge. Due to minimality we can then find a $2K_2$. The result is summarized in Corollary 7.9.

Then, it remains to show that each side of the bipartition has an nno. Let U be the larger side of the partition. By [121] it suffices to show that the input is a chain graph iff the graph obtained by adding all possible edges with both endpoints in U is a threshold graph. Instead of materializing the mentioned threshold graph, we implicitly represent the adjacencies of vertices in U to retain the same I/O-complexity and apply Lemma 7.5 using $\mathcal{O}(\text{sort}(n+m))$ I/Os. If the input is bipartite but not chain, we repeatedly delete vertices that are connected to all other vertices of the other side and the resulting isolated vertices, similar to Section 7.2.3 and [96]. After this, the vertex v with highest degree has a non-neighbor y in the other partition. By similar arguments y is adjacent to another vertex z that is adjacent to a vertex x where $\{v,x\} \notin E$ [96]. As such $G[\{v,y,z,x\}]$ is a $2K_2$ and can be found in $\mathcal{O}(\text{scan}(n))$ I/Os and returned as the NO-certificate.

Lemma 7.8. A graph with n vertices and m edges stored in external memory is certified whether it is a bipartite graph or not in $\mathcal{O}(\text{sort}(n+m))$ I/Os, given a spanning forest of the input graph. In the membership case the algorithm returns in $\mathcal{O}(\text{scan}(n))$ I/Os a bipartition $(U, V\setminus U)$ as the YES-certificate, and otherwise it returns an odd cycle as the NO-certificate. ◀

Proof. In case there are multiple connected components, we operate on each individually and thus assume that the input is connected. Let T be the edges of the spanning tree and $E\setminus T$ the non-tree edges. Any edge $e \in E\setminus T$ may produce an odd cycle by its addition to T. In fact, the input is bipartite iff $T \cup \{e\}$ is bipartite for all $e \in E \setminus T$[2]. We check whether an edge $e=\{u,v\}$ closes an odd cycle in T by computing the distance $d_T(u,v)$ of its endpoints in T. Since this is required for every non-tree edge $E\setminus T$, we resort to batch-processing. Note that T is a tree and hence after choosing a designated root $r \in V$ it holds that $d_T(u,v) = d_T(u,\text{LCA}_T(u,v)) + d_T(v,\text{LCA}_T(u,v))$ where $\text{LCA}_T(u,v)$ is the lowest common ancestor of u and v in T. Therefore for every edge $E\setminus T$ we compute its lowest common ancestor in T using $\mathcal{O}((m/n)\cdot\text{sort}(n)) = \mathcal{O}(\text{sort}(m))$ I/Os [51].

Additionally, for each vertex $v \in V$ we compute its depth in T in $\mathcal{O}(\text{sort}(m))$ I/Os using Euler Tours [51] and inform each incident edge of this value by a few scanning and sorting steps. Similarly, each edge $e=\{u,v\}$ is provided of the depth of $\text{LCA}_T(u,v)$. Then, after a single scan over $E\setminus T$ we compute $d_T(u,v)$ and check if it is even. If any value is even, we return the odd cycle as a NO-certificate or a bipartition in T as the YES-certificate. Both can be computed using Euler Tours in $\mathcal{O}(\text{sort}(m))$ I/Os. □

Corollary 7.9. If a connected graph G contains a C_3, C_5 or $2K_2$ then any of these subgraphs can be found in $\mathcal{O}(\text{sort}(n+m))$ I/Os given a spanning tree of G. ◀

Proof. We extend the algorithm presented in Lemma 7.8 since it does not return an induced cycle. While iterating over the edges to find an odd cycle we save the smallest seen odd cycle by keeping a copy of the edge $e \in E\setminus T$ and the length of the minimum odd cycle. In case we find a C_3 or a C_5 we are done and return the NO-certificate immediately otherwise for a C_k with $k = 2\ell+1 > 5$ we return a $2K_2$ by finding a matching edge to the non-tree edge $e \in E\setminus T$ in the cycle.

Let $C = (u_1,\dots,u_k,u_1)$ be the returned cycle where $\{u_k,u_1\}$ is the non-tree edge. In this case we return for the $2K_2$ the graph $(\{u_\ell,u_{\ell+1},u_1,u_k\}, \{\{u_1,u_k\},\{u_\ell,u_{\ell+1}\}\})$.

[2]Since T is bipartite, one can think of T as a representation of a 2-coloring on T.

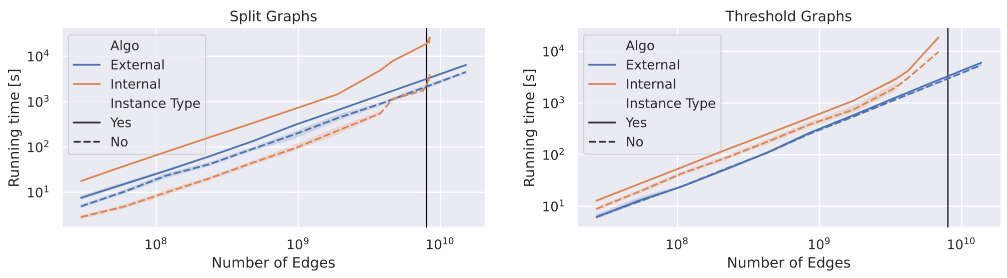


Figure 7.1: Running times of the external memory algorithms for certifying split (left) and threshold graphs (right) for different random graph instances. The black vertical lines depicts the number of elements that can concurrently be held in internal memory.

If ℓ is odd, the non-edges of the $2K_2$ cannot exist since otherwise any of the following smaller odd cycles $(u_1, u_2, \ldots, u_{\ell+1}, u_k, u_1)$, $(u_1, u_2, \ldots, u_\ell, u_1)$, $(u_\ell, u_{\ell+1}, \ldots, u_k, u_\ell)$ and $(u_1, u_{\ell+1}, u_{\ell+2}, \ldots, u_k, u_1)$ would be present, contradicting the minimality of C. For the other case where ℓ is even, a similar argument can be found. The I/O-complexity therefore remains the same. □

We summarize our findings for bipartite chain graphs in Lemma 7.10.

Lemma 7.10. A graph with n vertices and m edges stored in external memory is certified whether it is a bipartite chain graph or not in $\mathcal{O}(\text{sort}(n+m))$ I/Os with high probability. In the membership case the algorithm returns in $\mathcal{O}(\text{scan}(n))$ I/Os the bipartition $(U, V \setminus U)$ and nested neighborhood orderings of both partitions as the YES-certificate, and otherwise it returns a $\mathcal{O}(1)$-size NO-certificate. ◀

Proof. Computing a spanning tree T requires $\mathcal{O}(\text{sort}(n+m))$ I/Os with high probability by an external memory variant of the Karger, Klein and Tarjan minimum spanning tree algorithm [51]. By Corollary 7.9 we find a C_3, C_5 or $2K_2$ if the input is not bipartite or not connected. We proceed by checking the nno's of both partitions in $\mathcal{O}(\text{sort}(n+m))$ I/Os using Lemma 7.5. □

7.3 Experimental Evaluation

We have implemented our external memory certifying algorithms for split and threshold graphs in C++ using the *STXXL* library [60]. To provide a comparison of our algorithms, we also implemented the internal memory state-of-the-art algorithms by Heggernes and Kratsch [96]. *STXXL* offers external memory versions of fundamental algorithmic building blocks like scanning, sorting and several data structures. Our benchmarks are built with GNU g++-10.3 and executed on a machine equipped with an AMD EPYC 7302P processor and 64 GB RAM running Ubuntu 20.04 using six 500 GB solid-state disks.

In order to validate the predicted scaling behaviour we generate our instances parameterized by n. For YES-instances of split graphs we generate a split partition (K, I) with $|K| = n/10$ and add each possible edge $\{u, v\}$ with probability $1/4$ for $u \in I$ and

$v \in K$. Analogously, YES-instances of threshold graphs are generated by repeatedly adding either isolated or universal vertices with probability $9/10$ and $1/10$, respectively. We additionally attempt to generate NO-instances by adding $\mathcal{O}(1)$ many random edges to the YES-instances. In a last step we randomize the vertex indices to extend the effect of random accesses on the running time of the algorithms.

In Figure 7.1 we present the running times of all algorithms on multiple YES- and NO-instances. It is clear that the performance of both external memory algorithms is not impacted by the main memory barrier while the running time of their internal memory counterparts already increases when at least half the main memory is used. This effect is amplified immensely after exceeding the size of main memory by only a small fraction for split graphs, Figure 7.1 (left) and we expect the same for threshold graphs.

Certifying the produced NO-instances of split graphs seems to require less time than their corresponding unmodified YES-instances as the algorithm typically stops prematurely. Furthermore, due to the low data locality of the internal memory variant it is apparent that the external memory algorithm is superior for the YES-instances. The performance on both YES- and NO-instances is very similar in external memory. This is in part due to the fact that the algorithm first performs a relabeling which increases the ratio of common computation significantly.

For threshold graphs, the external memory variant outperforms the internal memory variant due to improved data locality. Analogously to split graphs, the difference in performance between YES- and NO-instances is more profound for the internal memory variants.

7.4 Conclusions

We have presented the first I/O-efficient certifying recognition algorithms for split, threshold, trivially perfect, bipartite and bipartite chain graphs. Our algorithms require $\mathcal{O}(\mathrm{sort}(n+m))$ I/Os matching common lower bounds for many algorithms in external memory. It would be interesting to further extend the scope of certifying algorithms to more graph classes for the external memory regime.

Acknowledgements

This work is partially supported by the International Exchanges Grant IES\R3\203041 of the Royal Society and by the Deutsche Forschungsgemeinschaft (DFG) under grant ME 2088/5-1 (FOR 2975 | Algorithms, Dynamics, and Information Flow in Networks).

Appendix 7.A Further Discussion on the Returned Certificates

We note that reverting any relabeling again requires only $\mathcal{O}(\text{sort}(n+m))$ I/Os by a constant number of scanning and sorting steps. Authenticating the YES-certificates of all our algorithms requires $\mathcal{O}(\text{sort}(n+m))$ I/Os anyway which is why we assume that the graph is given in its relabeled form.

Proposition 7.11. Authenticating (K, I) for a given split graph with n vertices and m edges requires $\mathcal{O}(\text{sort}(n+m))$ I/Os. ◀

Proof. Since we can assume relabelled vertex indices, let $I = \{v_1, \dots, v_k\}$ and $K = V \setminus I$. After sorting the edges in $\mathcal{O}(\text{sort}(m))$ I/Os, we check that no edge between I and K exists by comparing the indices. Verifying that K is a clique only requires looking at the $\binom{|K|}{2}$ last edges where both are done in $\mathcal{O}(\text{scan}(m))$ I/Os. □

Proposition 7.12. Authenticating $\beta = (u_1, \dots, u_k)$ for a given threshold graph with n vertices and m edges requires $\mathcal{O}(\text{sort}(n+m))$ I/Os. ◀

Proof. We can again assume that the vertices in the ordering of β are given by $I = \{v_1, \dots, v_k\}$. Verifying (K, I) is done as described in Proposition 7.11 using $\mathcal{O}(\text{sort}(m))$ I/Os. It remains to verify that β is a nno. For increasing i we verify $N(v_i) \subseteq N(v_{i+1})$ by a concurrent scan over both neighborhoods requiring in total $\mathcal{O}(\text{scan}(m))$ I/Os for all i. □

Proposition 7.13. Authenticating $\gamma = (v_1, \dots, v_n)$ for a given trivially perfect graph with n vertices and m edges requires $\mathcal{O}(\text{sort}(n+m))$ I/Os. ◀

Proof. We rerun the Algorithm 10 using $\mathcal{O}(\text{sort}(n+m))$ I/Os as the certificate is the ordering itself. □

Proposition 7.14. Authenticating $(U, V \setminus U)$ with two nested neighborhood orderings for a given bipartite chain graph with n vertices and m edges requires $\mathcal{O}(\text{sort}(n+m))$ I/Os. ◀

Proof. Similar to Proposition 7.11 we check that U and $V \setminus U$ are both independent sets using $\mathcal{O}(\text{sort}(n+m))$ I/Os. Thereafter similar to Proposition 7.12 we verify that both orderings are indeed nested neighborhood orderings using again $\mathcal{O}(\text{sort}(n+m))$ I/Os. □

Summary

8

The present thesis considers algorithmic aspects for the generation and analysis of graphs. First, we develop efficient sequential, shared-memory parallel and I/O-efficient algorithms that obtain uniform samples of random graphs with prescribed degrees based on the following stochastic processes.

- *Edge Switching* ☞ *Chapters 2 and 3*
- *Global Edge Switching* ☞ *Chapter 3*
- *Curveball* ☞ *Chapter 4*
- *Global Curveball* ☞ *Chapter 4*

Additionally, specifically for power-law graphs we engineer an implementation of the exact uniform sampling algorithm Inc-Powerlaw and provide crucial details that were previously not discussed including previously overlooked gaps leading to a compromised running time.

- Inc-Powerlaw ☞ *Chapter 5*

Second, we present algorithms for the analysis and processing of large networks. In particular, we provide I/O-efficient algorithms and implementations for the following two problems.

- *Connected Components* ☞ *Chapter 6*
- *Certifying Graph Recognition* ☞ *Chapter 7*

In this chapter, we summarize our results for both parts of the thesis and give an outlook for further future research opportunities.

8.1 Uniform Sampling of Simple Graphs with Prescribed Degrees

In this section, we summarize our results for the problem of uniformly sampling simple graphs from a prescribed degree sequence. It is a common task in the analysis and synthesis of networks; e.g., to randomize existing graphs, as a subroutine in more complex generators or to construct null-models [55, 102, 163]. Our results are two-fold; first we give a multitude of algorithms implementing Markov-Chain-Monte-Carlo (MCMC) processes for the efficient randomization of graphs while maintaining the degrees, see Figure 8.1 for a comprehensive overview. Finally, we present an exact uniform generation algorithm for power-law degree sequences in Section 8.1.3.

8.1.1 Edge Switching and Global Edge Switching

LFR: ☞ *Section 2.3*

Chapter 2 is based on our article [90] and presents EM-LFR, a complex assembly of several I/O-efficient subroutines to generate large *LFR* graph instances exceeding the size of main memory. In EM-LFR the most challenging component is to uniformly sample a simple graph from a prescribed degree sequence. To realize this, EM-LFR splits the graph sampling into two steps following the *Fixed-Degree-Sequence-Model* (*FDSM*). First, a biased simple graph is deterministically generated that matches the prescribed degrees. Then, the obtained graph is perturbed using *Edge Switching* (*ES*) while maintaining the prescribed degrees. In its original formulation, EM-LFR employs the algorithms EM-HH and EM-ES to implement the aforementioned steps where EM-HH is an I/O-efficient realization of a generator due to Havel and Hakimi [95, 89] and EM-ES provides an I/O-efficient algorithm to perturb a given graph using *Edge Switching*. As the graphs generated by EM-HH are highly biased and therefore may require many perturbation steps of EM-ES, we consider the *Configuration Model* (*CM*) as a substitute for the first step and adjust the pipeline of EM-LFR accordingly.

EM-HH und EM-ES: ☞ *Sections 2.4 and 2.5*

EM-CM/ES: ☞ *Section 2.6*

- *EM-CM/ES:* We provide EM-CM/ES as an alternative to the originally proposed combination of EM-HH and EM-ES. For this, we implement an I/O-efficient generator sampling from the *Configuration Model* substituting EM-HH in the first step. As the initial graph is unlikely to be a simple graph, we adjust the perturbation phase to adhere to the potentially non-simple input.

  To do so, EM-ES is modified to accept non-simple inputs without an increase in I/O-complexity. In order to arrive at a simple graph, the algorithm accepts all switches that neither produce further multi-edges nor introduce self-loops. To additionally accelerate this process, non-simple edges are switched more often than legal edges.

  As the samples obtained by this rewiring process are still biased [5, 17], further perturbation using *ES* is necessary requiring additional applications of EM-ES. While this approach is much more involved, we provide empirical evidence that EM-CM/ES can converge faster to a uniform sample than the previously proposed combination.

Chapter 3 is based on our article [6] and considers shared-memory parallel approaches to randomize simple graphs. We provide a simple parallelization of the *Edge Switching Markov Chain* (*ES-MC*) which, due to the dependencies arising in *ES-MC*, is unlikely to scale well. As a means to alleviate this issue, we further propose *Global Edge Switching Markov Chain* (*G-ES-MC*) a variant of *ES-MC* with easier dependencies to enable better parallelism. Similar to *ES-MC*, we demonstrate that *G-ES-MC* also converges to a uniform distribution on the set of graphs with matching degrees. In our experiments, we provide empirical evidence that *G-ES-MC* uses at most the same number of switches as the standard *ES-MC* and show the efficiency and scalability of our implemented algorithms.

- *RobinES and GlobalES:* We provide RobinES and GlobalES as sequential baseline solutions of *ES-MC* and *G-ES-MC*, respectively. Our implementations use hash-sets to support edge insertion, deletion and existence queries in expected constant time, further auxiliary data structures for the sampling of edges and prefetching to accelerate the random I/Os to main memory.

  ES-MC and G-ES-MC: ☞ *Section 3.2*

  In a comparison with *NetworKit* [171] and *Gengraph* [181], we show that our solutions run 15-50 times faster than *NetworKit* and 5-10 times faster than *Gengraph*. For large graphs, GlobalES outperforms RobinES as shuffling the edges becomes more efficient than sampling, whereas RobinES is faster for smaller graphs.

- *EagerES and SteadyGlobalES:* In order to establish a performance baseline for our parallel algorithms, we present EagerES, a simplistic parallelization of *ES-MC*. The algorithm uses a concurrent hash-set and only deploys an implicit synchronization scheme where each processing unit performs its switches independently. This however results in an execution order that is dependent on the scheduler and therefore does not faithfully represent the *ES-MC*.

  EagerES: ☞ *Section 3.4.1*

  For the *G-ES-MC*, we present SteadyGlobalES, a parallel faithful implementation that processes uniform random global switches instead. By design, global switches exhibit easier dependencies which we categorize into two types: *erase* and *insert* dependencies. A global switch is then processed in parallel using multiple rounds where in each round switches with no prohibiting dependencies are resolved. Despite the increased algorithmic complexity, our experiments suggest that SteadyGlobalES only incurs a slowdown of at most 2 compared to EagerES.

  SteadyGlobalES: ☞ *Section 3.4.2*

8.1.2 Curveball and Global Curveball

Chapter 4 is based on our article [48] and presents a variety of algorithms implementing a different type of randomization process, namely *Curveball* (*CB*) [174, 46] and *Global Curveball* (*G-CB*) [47, 48]. *CB* proceeds similar to *ES* but instead randomly selects two nodes $u \neq v$ and performs a *trade*, shuffling their neighborhoods. To do so, *CB* first collects all non-common neighbors of u and v, excludes the trading nodes, and randomly redistributes them while maintaining the degrees of the traded nodes. Since the entire neighborhoods of both nodes may be considered, a single trade possibly yields a larger change in the graph compared to a single *ES* switch.

While *CB* draws its trade constituents uniformly at random, *G-CB* proceeds differently by grouping multiple single trades into a superstep. These so-called *global trades* target each node exactly once[1] and the execution of all its single trades is interpreted as a single step of *G-CB*. Though *G-CB* initially was proposed for directed graphs, we generalize it for the undirected case and show that it converges to a uniform distribution. Additionally, our experiments give empirical evidence that *G-CB* performs better than *CB*. With further experiments we show the efficiency and scalability of our algorithms beyond the size of the main memory.

Undirected G-CB: ☞ *Section 4.3.3*

- *EM-CB and IM-CB:* We provide EM-CB, an I/O-efficient sequential algorithm for *CB*. As neighborhood changes due to trade have to be reflected for the participating nodes as well, EM-CB abandons a classical static graph data structure where unstructured accesses would need to be performed. By employing *Time Forward Processing* (*TFP*), we circumvent these unstructured access patterns and dynamically manage the graph by interpreting each trade as a point in time. Then, for each trade, only the neighborhoods of the constituents are required, which are provided in *TFP* fashion. To realize this, EM-CB performs trades in batches where for each batch all trade pairs are first randomly sampled and then organized in auxiliary data structures to properly forward messages to trades accordingly.

  EM-CB: ☞ *Section 4.4.1*

  TFP: ☞ *Section 4.2.2*

  In the case that memory accesses are not the bottleneck, we propose IM-CB as a faster alternative to EM-CB, especially for small graph instances. By discarding the necessary data structures for *TFP* and using a classical adjacency vector data structure, IM-CB accepts the unstructured accesses and therefore excels at small and medium-sized graphs.

  IM-CB: ☞ *Section 4.4.2*

- *EM-GCB and EM-PGCB:* EM-GCB is our I/O-efficient algorithm for undirected *G-CB*. By exploiting the additional structure of global trades we can omit the auxiliary data structures of EM-CB and IM-CB. More precisely, we interpret a global trade as a random permutation of nodes and represent it implicitly using a suitable type of hash-function enabling further optimizations.

  EM-GCB: ☞ *Section 4.4.3*

  Further engineering leads to EM-PGCB, a parallel extension of EM-GCB. To enable parallelism, we subdivide the global trades into even smaller so-called *macrochunks* that are individually held in main memory. These macrochunks are then equally divided into smaller *microchunks* where their size is carefully chosen such that almost all trades can be independently performed in parallel. In the rare cases where dependencies arise, we resort to a variant of work stealing to avoid unnecessary waiting times. We give experimental evidence that, in some cases, EM-PGCB is nearly one order of magnitude faster in comparison to EM-ES while achieving the same randomization quality.

  EM-PGCB: ☞ *Section 4.4.4*

[1]For the sake of simplicity we assume the number of nodes to be even; for the general case see Section 4.4.3.

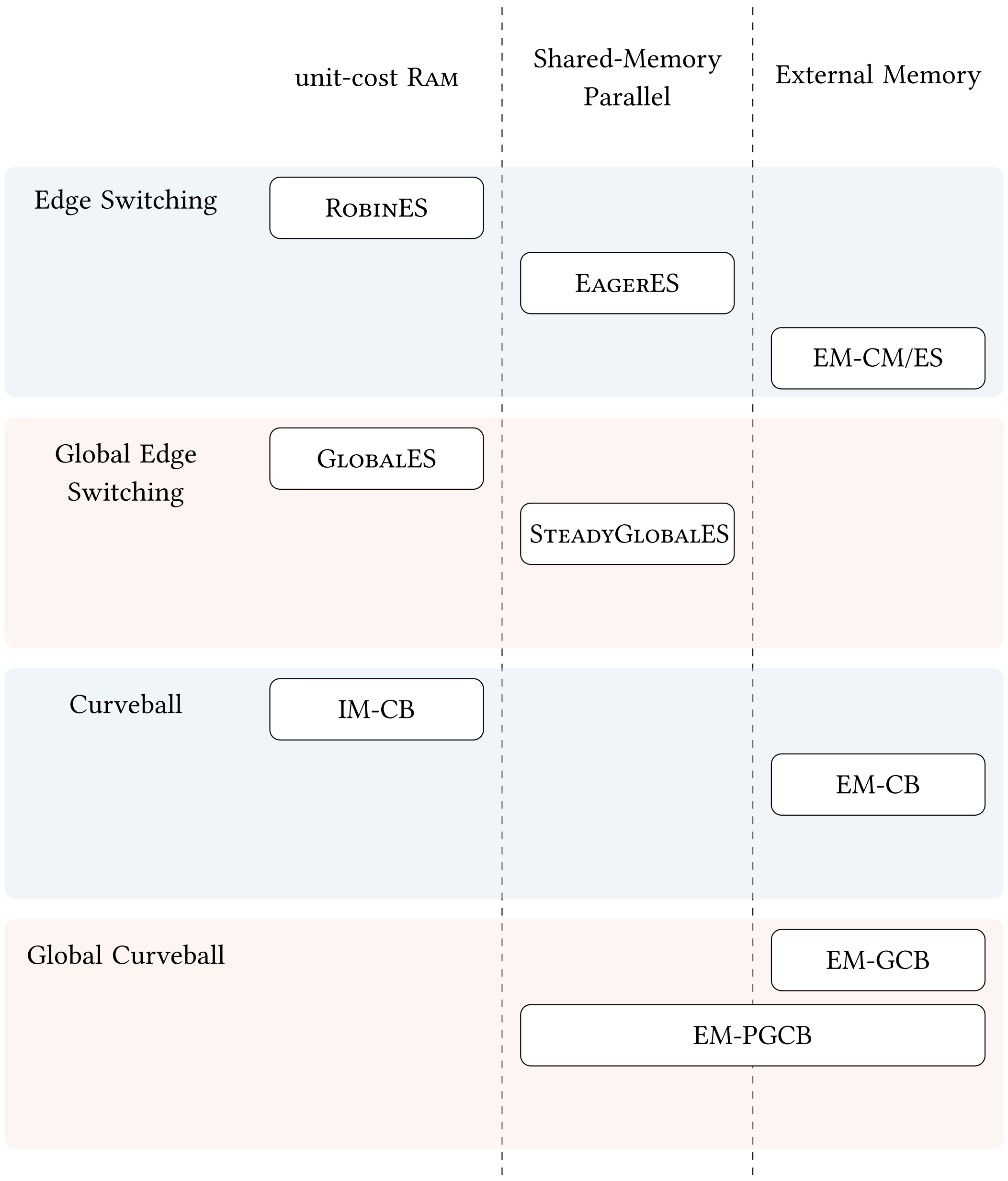


Figure 8.1: An overview of the described algorithms categorized into the machine models.

8.1.3 Exact Uniform Sampling of Power-law Graphs

Chapter 5 is based on our article [9] and presents the Inc-Powerlaw algorithm in its full description. Inc-Powerlaw improves on the Pld algorithm [78] by applying a recently developed technique called incremental relaxation due to Arman et al. [17]. For the parts where incremental relaxation are used, we determine the order in which the relevant graph substructures should be relaxed, how to count them and provide appropriate lower bounds. In our investigation we find that incremental relaxation compromised Inc-Powerlaw's linear runtime in its original formulation due to too many rejections. To solve this issue we add new switchings to Phases 4 and 5 of the algorithm and provide proofs that the rejection probability is sufficiently small.

To verify our findings, we implement and engineer an Inc-Powerlaw implementation and consider parallelization options. In our empirical study, we observe that Inc-Powerlaw is very efficient for small average degrees and notice significantly larger constants for larger average degrees. Furthermore, we empirically confirm the linear expected runtime of Inc-Powerlaw.

Inc-Powerlaw: ☞ *Section 5.2*

- *Inc-Powerlaw:* Roughly speaking, Inc-Powerlaw first generates a random graph according to the *Configuration Model* and rewires illegal structures into legal counterparts using a choreography of more than 20 different switching types. To ensure uniformity, rejection sampling is used such that all intermediate graphs remain uniform samples in their respective class. More precisely, whenever a random switching is sampled it may be rejected (f-rejection or b-rejection) and the algorithm be forced to restart.

  We present all necessary requirements to recover the linear runtime of Inc-Powerlaw. For Phase 4 the addition of incremental relaxation eases the computational cost but compromises the b-rejection probability by causing too many rejections. We address this issue by adding three *booster* switchings (t_a, t_b and t_c switchings) to the originally used t-switching. All four switchings create the *triplet* structure and potentially additional edges depending on the switching. By computing the appropriate constants and lower bounds we prove that the probability of a b-rejection in Phase 4 is then $o(1)$.

  Similarly, for Phase 5 of Inc-Powerlaw the b-rejection probability is increased by adding incremental relaxation. We again, add booster switchings (type-III, type-IV, type-V, type-VI and type-VII switchings) to create the *doublet* structure and reduce the probability of rejection. Analogously, we then provide a proof that the probability of a b-rejection in Phase 5 is $o(1)$.

- *Intra-Run and Inter-Run Parallelism:* In our experimental evaluation we show that the sampling of the initial multi-graph and constructing the appropriate data structures are the dominating factors in the algorithm. Our implementation therefore incorporates parallelization strategies to alleviate the impact of the aforementioned bottleneck. We consider two orthogonal strategies: Intra-Run and Inter-Run.

Parallelization: ☞ *Section 5.4.2*

While Intra-Run directly parallelizes the construction of the multi-graph and its representative data structure, Inter-Run starts multiple runs and accepts the first accepted one. We use synchronization to avoid a selection bias towards quicker runs: all processors assign globally unique indices to their runs and we return the run with smallest index that is accepting.

8.2 Experimental Algorithms for External Memory

In this section, we present our results that particularly focus on the large-scale analysis of networks. Due to the ever-increasing size of data, large-scale algorithmic solutions are necessary to facilitate a means to study large networks. As the classical assumption of a *unit-cost Random-Access Machine* (*unit-cost Ram*) does not fully capture the practical performance of algorithms in the presence of memory hierarchies, we turn our attention to the *External Memory Model* (Emm).

8.2.1 Connected Components

Chapter 6 is based on our article [41] and presents an empirical study of the *Connected Components* (*CC*) problem in *External Memory Model*. We consider several candidate algorithms that are either theoretically efficient or seem practically promising. Our experiments are then executed on a variety of different graph classes including popular random models like *Gilbert* graphs, *Random Geometric Graphs* (*RGGs*) and *Random Hyperbolic Graphs* (*RHGs*).

- *Borůvka and Randomized-Borůvka:* We provide an implementation of the externalized version of Borůvka's algorithm [51] as a baseline for our external memory algorithms. It computes the *Connected Components* of a graph by repeatedly performing Borůvka steps that reduces the number of nodes. Its translation to external memory however, significantly increases its algorithmic complexity requiring the use of many auxiliary data structures leading to subpar performances in practice.

  Borůvka: ☞ *Section 6.3*

  To mitigate the use of these additional data structures, we propose a similar but more light-weight algorithm called Randomized-Borůvka. Randomized-Borůvka works essentially like Borůvka but produces smaller easier to handle subgraphs that can be contracted more efficiently. Both algorithms, however, could not compete with the other candidates.

  Randomized-Borůvka: ☞ *Section 6.3*

- *Sibeyn:* Sibeyn is a *Minimum Spanning Forest* (*MSF*) algorithm that exhibits further optimization possibilities when considering the easier problem of computing *Connected Components*. We consider Sibeyn for its simplicity which resulted in our highly efficient implementation. In this setting, Sibeyn repeatedly lets some node select an arbitrary incident edge which is contracted subsequently. This process is realized lazily using *Time Forward Processing* and therefore produces comparably very small I/O-volumes in practice.

  Sibeyn: ☞ *Section 6.3*

In our experiments we consider several edge selection strategies and a variety of ways to implement the lazy contractions using either priority-queues or buckets.

Karger-Klein-Tarjan: ☞ *Section 6.3*

- *Karger-Klein-Tarjan:* We consider the well-known algorithm of Karger, Klein and Tarjan [108] for the *MSF* problem in a more general framework. For the external memory setting, an externalized version exists and can again be translated to the *Connected Components* problem. Our general framework proceeds recursively and requires a few subroutines: reducing the set of nodes using contractions, random sampling of the edges and merging recursively computed solutions of the *CC* problem. Due to the increased variability, we are able to consider a plethora of parameter combinations for Karger-Klein-Tarjan. Among these are Borůvka, Randomized-Borůvka or Sibeyn for the node contractions and different sampling parameters that could even be adaptively set.

  Even though Karger-Klein-Tarjan is the theoretically most efficient algorithm, a lot of tuning was necessary to provide a practical implementation. This is in part due to its recursive nature and the additionally required auxiliary data structures.

In our experiments we find that Sibeyn is a strong contender due to its simplicity. Its implementation essentially only requires the use of a single external memory priority-queue. As such, any competing algorithm can only perform very few operations before losing to Sibeyn. However, Karger-Klein-Tarjan is still a competitive choice; using the right subroutines and tunings the algorithm provides a robust solution that performs comparably well.

8.2.2 Certifying Graph Recognition

Chapter 7 is based on our article [134] and presents I/O-efficient certifying recognition algorithms for bipartite, split, threshold, bipartite chain and trivially perfect graphs. On a graph with n vertices and m edges, our algorithms incur $\mathcal{O}(\text{sort}(n+m))$ I/Os in the worst-case or with high probability for bipartite and bipartite chain graphs. In the positive membership case, a YES-certificate that characterizes the graph class is returned. Contrary, in the non-membership case a $\mathcal{O}(1)$-size NO-certificate is returned for all graph classes except for the bipartite case.

We adapt the internal memory algorithms of Heggernes and Kratsch [96] to the external memory setting using standard techniques including *Time Forward Processing* and Euler tour computations. For all graph classes we exploit their key structural properties to develop I/O-efficient algorithms.

Certifying Split Graphs: ☞ *Section 7.2.1*

- *Split Graphs:* Split graphs are graphs where the vertices form a split partition (K, I) meaning that K and I are a clique and an independent set, respectively. We additionally exploit the following key insights: (i) the maximum clique consists of the highest degree vertices; and (ii) any non-decreasing degree ordering of a split graph is a perfect elimination ordering. Computing this ordering and relabeling the graph accordingly eases necessary further computational tasks, i.e., verifying that K is indeed a clique and I is indeed an independent set.

- *Threshold Graphs:* Threshold graphs are split graphs with the additional property that the independent set I has a nested neighborhood ordering. They are additionally characterized by the following graph generation process: repeatedly add universal or isolated nodes to an initially empty graph. Due to the relabeling, we can I/O-efficiently verify this property in reverse by repeatedly removing universal and isolated nodes from the input graph.

Certifying Threshold Graphs: ☞ Section 7.2.2

- *Trivially Perfect Graphs:* Trivially perfect graphs are graphs where in each of its induced subgraphs the size of the maximum independent set equals the number of maximal cliques. Similar to split graphs, we exploit that trivially perfect graphs exhibit a distinguished feature in their degrees: any non-increasing degree ordering is a universal-in-a-component ordering. Using *Time Forward Processing* we translate the iterative labeling scheme of Heggernes and Kratsch [96] and verify this property I/O-efficiently.

Certifying Trivially Perfect Graphs: ☞ Section 7.2.3

- *Bipartite and Bipartite Chain Graphs:* Bipartite chain graphs are bipartite graphs where the partitions emit a nested neighborhood ordering. As such, we first develop an I/O-efficient certifying algorithm for the recognition of bipartite graphs. Instead of using graph traversal algorithms, we rely on spanning forests and batch-processing to realize an I/O-efficient certifying recognition algorithm for bipartite graphs. By combining this and the developed algorithm for threshold graphs; we also provide an algorithm for the case of bipartite chain graphs requiring $\mathcal{O}(\mathrm{sort}(n+m))$ I/Os with high probability.

Certifying Bipartite and Bipartite Chain Graphs: ☞ Section 7.2.4

8.3 Future Research Opportunities

In this section, we highlight a few additional open questions related to Chapters 2 to 7 that may lead to further future research opportunities.

8.3.1 Uniformly Sampling Simple Graphs with Prescribed Degrees

While there exists a plethora of MCMC algorithms and many practical efficient implementations thereof, a lack of rigorous mixing times for large families of degree sequences still persists. Even comparisons among them are only scarcely available both in theory and practice. Our empirical evaluations in Chapters 2 to 4 shed some light towards this issue but further work is necessary for the reliable sampling of uniform simple graphs with prescribed degrees. Especially for practical implications, new techniques inspired by recent developments in machine learning may be of particular interest. In this context, practitioners may use machine learning tools to systemically find experimental lower bounds for any given MCMC algorithm.

On the other hand, efficient exactly uniform samplers are less prevalent and further theoretical work is required to facilitate more practical implementations. To this end, dedicated algorithms similar to Inc-Powerlaw can provide a consistent means to uniformly sample simple graphs with prescribed degrees. Further results in this direction may be achievable by adding even more switchings analogously to Inc-Powerlaw, potentially increasing the range of permissible degree sequences.

8.3.2 Connected Components and Related Problems

In the external memory regime, extensive experimental studies are scarcely conducted due to their heavy computational requirements. They, however, provide an important view on the designed algorithms that are typically rooted in theoretical considerations. As such, it would be interesting to execute a large-scale experimental study not only for the *Connected Components* problem but also for related problems like *Minimum Spanning Forest* or finding *any* spanning forest in a given graph. The then gained insights may inspire new algorithmic techniques to break current theoretical barriers; e.g., whether computing *Connected Components* or *Minimum Spanning Forest* are equivalently hard in external memory, finding improved upper bounds and many more.

8.3.3 Certifying Graph Recognition

Naturally, it would be interesting to extend the scope of certifying algorithms in the external memory regime; broadening the range of permissible graph classes that can be certified I/O-efficiently. In internal memory, a plethora of graph classes are efficiently certifiable, while they currently have no efficient external memory pendant, e.g. circular-arc graphs [72], HHD-free graphs [147], interval graphs [111], normal helly circular-arc graphs [44], permutation graphs [111], proper interval graphs [97], proper interval bigraphs [97], unit interval graphs [57] and many more. Due to limited data locality, straight-forward applications of these algorithms are highly inefficient for use in external memory. In turn, new algorithmic techniques are necessary to bridge the gap to larger processing scales.

Furthermore, moving away from the static setting and rather considering dynamic certifying algorithms is also a natural choice. Again, in internal memory some dynamic recognition algorithms exist (e.g. [168, 58]) but have not been extended to larger scales.

Bibliography

[1] 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.

[2] D. Ajwani and U. Meyer. Design and engineering of external memory traversal algorithms for general graphs. In J. Lerner, D. Wagner, and K. A. Zweig, editors, *Algorithmics of Large and Complex Networks - Design, Analysis, and Simulation [DFG priority program 1126]*, volume 5515 of *Lecture Notes in Computer Science*, pages 1–33. Springer, 2009. doi:10.1007/978-3-642-02094-0_1.

[3] 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.

[4] S. Albers, A. Crauser, and K. Mehlhorn. Lecture notes on algorithms for very large data sets. https://web.archive.org/web/19970816002522/http://www.mpi-sb.mpg.de/~crauser/Plan.ps.gz, 1997.

[5] 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.

[6] D. Allendorf, U. Meyer, M. Penschuck, and H. Tran. Parallel global edge switching for the uniform sampling of simple graphs with prescribed degrees. In *IEEE Int. Parallel and Distributed Processing Symp. IPDPS*, pages 269–279. IEEE, 2022. doi:10.1109/IPDPS53621.2022.00034.

[7] D. Allendorf, U. Meyer, M. Penschuck, and H. Tran. Parallel and I/O-efficient algorithms for non-linear preferential attachment. In G. Navarro and J. Shun, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2023, Florence, Italy, January 22-23, 2023*, pages 65–76. SIAM, 2023. doi:10.1137/1.9781611977561.ch6.

[8] D. Allendorf, U. Meyer, M. Penschuck, and H. Tran. Parallel global edge switching for the uniform sampling of simple graphs with prescribed degrees. *J. Parallel Distributed Comput.*, 174:118–129, 2023. doi:10.1016/j.jpdc.2022.12.010.

[9] D. Allendorf, U. Meyer, M. Penschuck, H. Tran, and N. Wormald. Engineering uniform sampling of graphs with a prescribed power-law degree sequence. In C. A. Phillips and B. Speckmann, editors, *Proceedings of the Symp. on Algorithm Engineering and Experiments ALENEX*, pages 27–40. Society for Industrial and App. Math. SIAM, 2022. doi:10.1137/1.9781611977042.3.

[10] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999. doi:10.1006/jcss.1997.1545.

[11] G. Amanatidis and P. Kleer. Rapid mixing of the switch markov chain for strongly stable degree sequences. *Random Struct. Algorithms*, 57(3):637–657, 2020. doi:10.1002/rsa.20949.

[12] 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.

[13] 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.

[14] 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.

[15] L. Arge, G. S. Brodal, and L. Toma. On external-memory mst, SSSP and multi-way planar graph separation. *J. Algorithms*, 53(2):186–206, 2004. doi:10.1016/j.jalgor.2004.04.001.

[16] L. Arge, M. Rav, S. C. Svendsen, and J. Truelsen. External memory pipelining made easy with TPIE. In J. 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, *2017 IEEE International Conference on Big Data (IEEE BigData 2017), Boston, MA, USA, December 11-14, 2017*, pages 319–324. IEEE Computer Society, 2017. doi:10.1109/BigData.2017.8257940.

[17] 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.

[18] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders. Engineering in-place (shared-memory) sorting algorithms. *ACM Trans. Parallel Comput.*, 9(1):2:1–2:62, 2022. doi:10.1145/3505286.

[19] 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.

[20] 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.

[21] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In J. D. P. Rolim and S. P. Vadhan, editors, *Proceedings of Randomization and Approximation Techniques RANDOM*, volume 2483 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2002. doi:10.1007/3-540-45726-7_1.

[22] A. L. Barabási. *Network science.* Cambridge university press, 2016.

[23] M. Bayati, J. H. Kim, and A. Saberi. A sequential algorithm for generating random graphs. *Algorithmica*, 58(4):860–910, 2010. doi:10.1007/s00453-009-9340-1.

[24] 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.

[25] A. Békéssy, P. Békéssy, and J. Komlós. Asymptotic enumeration of regular matrices. *Stud. Sci. Math. Hungar.*, 7, 1972.

[26] 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.

[27] 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.

[28] 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.

[29] A. Berger and C. J. Carstens. Smaller universes for uniform sampling of 0,1-matrices with fixed row and column sums. *CoRR*, abs/1803.02624, 2018.

[30] 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.

[31] M. H. Bhuiyan, M. Khan, J. Chen, and M. V. Marathe. Parallel algorithms for switching edges in heterogeneous graphs. *J. Parallel Distributed Comput.*, 104:19–35, 2017. doi:10.1016/j.jpdc.2016.12.005.

[32] M. H. Bhuiyan, M. Khan, and M. V. Marathe. A parallel algorithm for generating a random graph with a prescribed degree sequence. In J. 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 BigData*, pages 3312–3321. IEEE Computer Society, 2017. doi:10.1109/BigData.2017.8258316.

[33] A. Bhushan and S. Gopalan. An I/O efficient algorithm for minimum spanning trees. In Z. Lu, D. Kim, W. Wu, W. Li, and D. Du, editors, *Int. Conf. on Combinatorial Optimization and Applications COCOA*, volume 9486 of *Lecture Notes in Computer Science*, pages 499–509. Springer, 2015. doi:10.1007/978-3-319-26626-8_36.

[34] Y. M. Bishop, S. E. Fienberg, and P. W. Holland. *Discrete multivariate analysis: theory and practice.* Springer Science & Business Media, 2007.

[35] J. K. Blitzstein and P. Diaconis. A sequential importance sampling algorithm for generating random graphs with prescribed degrees. *Internet Math.*, 6(4):489–522, 2011. doi:10.1080/15427951.2010.557277.

[36] 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.

[37] P. Boldi and S. Vigna. Axioms for centrality. *Internet Math.*, 10(3-4):222–262, 2014. doi:10.1080/15427951.2013.865686.

[38] B. Bollobás. *Random Graphs, Second Edition*, volume 73 of *Cambridge Studies in Advanced Math.* Cambridge University Press, 2011. doi:10.1017/CBO9780511814068.

[39] B. Bollobás. A probabilistic proof of an asymptotic formula for the number of labelled regular graphs. *Eur. J. Comb.*, 1(4):311–316, 1980. doi:10.1016/S0195-6698(80)80030-8.

[40] U. Brandes, G. Robins, A. McCranie, and S. Wasserman. What is network science? *Network Science*, 1(1):1–15, 2013. doi:10.1017/nws.2013.2.

[41] G. S. Brodal, R. Fagerberg, D. Hammer, U. Meyer, M. Penschuck, and H. Tran. An experimental study of external memory algorithms for connected components. In D. Coudert and E. Natale, editors, *Int. Symp. on Experimental Algorithms SEA*, volume 190 of *LIPIcs*, pages 23:1–23:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.SEA.2021.23.

[42] A. L. Buchsbaum, M. H. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In D. B. Shmoys, editor, *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, January 9-11, 2000, San Francisco, CA, USA*, pages 859–860. ACM/SIAM, 2000. URL: http://dl.acm.org/citation.cfm?id=338219.338650.

[43] 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.

[44] Y. Cao. Direct and certifying recognition of normal helly circular-arc graphs in linear time. In *Proc. Frontiers in Algorithmics (FAW)*, volume 8497 of *LNCS*, pages 13–24. Springer, 2014.

[45] 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.

[46] C. J. Carstens. *Topology of Complex Networks: Models and Analysis.* PhD thesis, RMIT University, 2016.

[47] 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. arXiv:1609.05137.

[48] 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.

[49] 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.

[50] B. Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *J. ACM*, 47(6):1028–1047, 2000. doi:10.1145/355541.355562.

[51] 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.

[52] 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.

[53] V. Chvátal. Set-packing and threshold graphs. *Res. Rep., Comput. Sci. Dept., Univ. Waterloo, 1973*, 1973.

[54] 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 ComplexNetworks CompleNet 2014*, volume 549 of *Studies in Computational Intelligence*, pages 199–208. Springer, 2014. doi:10.1007/978-3-319-05401-8_19.

[55] 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.

[56] C. Cooper, M. E. Dyer, and C. S. Greenhill. Sampling regular graphs and a peer-to-peer network. *Comb. Probab. Comput.*, 16(4):557–593, 2007. doi:10.1017/S0963548306007978.

[57] D. G. Corneil. A simple 3-sweep LBFS algorithm for the recognition of unit interval graphs. *Disc. Appl. Math.*, 138(3):371–379, 2004.

[58] C. Crespelle and C. Paul. Fully dynamic recognition algorithm and certificate for directed cographs. *Disc. Appl. Math.*, 154(12):1722–1741, 2006.

[59] A. Czumaj and A. Lingas. On truly parallel time in population protocols. *CoRR*, abs/2108.11613, 2021. arXiv:2108.11613.

[60] 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.

[61] R. Dementiev, P. Sanders, D. Schultes, and J. F. Sibeyn. Engineering an external memory minimum spanning tree algorithm. In J. Lévy, E. W. Mayr, and J. C. Mitchell, editors, *IFIP Int. Conf. on Theor. Comp. Science TCS*, volume 155 of *IFIP*, pages 195–208. Kluwer/Springer, 2004. doi:10.1007/1-4020-8141-3_17.

[62] 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.

[63] 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. doi:10.1371/journal.pone.0159161.

[64] P. Erdős and A. Rényi. On random graphs I. *Publicationes Mathematicae Debrecen*, 1959.

[65] P. L. Erdös, C. S. Greenhill, T. R. Mezei, I. Miklós, D. Soltész, and L. Soukup. The mixing time of switch markov chains: A unified approach. *Eur. J. Comb.*, 99:103421, 2022. doi:10.1016/j.ejc.2021.103421.

[66] P. L. Erdős, S. Z. Kiss, I. Miklós, and L. Soukup. Approximate counting of graphical realizations. *PLOS ONE*, 10(7):1–20, 07 2015. doi:10.1371/journal.pone.0131300.

[67] A. V. Esquivel and M. Rosvall. Comparing network covers using mutual information, 2012. arXiv:1202.0425.

[68] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985. doi:10.1016/0022-0000(85)90041-8.

[69] S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75–174, Feb 2010. doi:10.1016/j.physrep.2009.11.002.

[70] S. Fortunato and M. Barthélemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences*, 104(1):36–41, 2007. doi:10.1073/pnas.0605965104.

[71] 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.

[72] M. C. Francis, P. Hell, and J. Stacho. Forbidden structure characterization of circular-arc graphs and a certifying recognition algorithm. In *ACM-SIAM Symp. on Discrete Algorithms SODA*, pages 1708–1727, 2015.

[73] 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.

[74] 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.

[75] P. Gao and N. C. Wormald. Uniform generation of random regular graphs. *SIAM J. Comput.*, 46(4):1395–1427, 2017. doi:10.1137/15M1052779.

[76] P. Gao and C. S. Greenhill. Mixing time of the switch markov chain and stable degree sequences. *Discret. Appl. Math.*, 291:143–162, 2021. doi:10.1016/j.dam.2020.12.004.

[77] P. Gao and N. C. Wormald. Uniform generation of random regular graphs. In V. Guruswami, editor, *IEEE Symp. on Foundations of Comp. Science FOCS*, pages 1218–1230. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.78.

[78] P. Gao and N. C. Wormald. Uniform generation of random graphs with power-law degree sequences. In A. Czumaj, editor, *ACM-SIAM Symp. on Discrete Algorithms SODA*, pages 1741–1758. SIAM, 2018. doi:10.1137/1.9781611975031.114.

[79] E. N. Gilbert. Random graphs. *The Annals of Mathematical Statistics*, 30(4):1141–1144, Dec 1959. doi:10.1214/aoms/1177706098.

[80] 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.

[81] 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 ALENEX*, pages 16–25. Society for Industrial and App. Math. SIAM, 2003.

[82] M. C. Golumbic. *Algorithmic graph theory and perfect graphs.* Elsevier, 2004.

[83] N. J. Gotelli and G. R. Graves. *Null models in ecology.* Smithsonian Institution, 1996.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] 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.

[89] 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.

[90] 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.

[91] 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.

[92] M. Hamann, B. Strasser, D. Wagner, and T. Zeitz. Simple distributed graph clustering using modularity and Map Equation. *CoRR*, abs/1710.09605, 2017. arXiv:1710.09605.

[93] P. L. Hammer and S. Földes. Split graphs. *Congressus Numerantium*, 19:311–315, 1977.

[94] 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.

[95] V. Havel. Poznámka o existenci konečných grafů. *Časopis pro pěstování matematiky*, 080(4), 1955.

[96] P. Heggernes and D. Kratsch. Linear-time certifying recognition algorithms and forbidden induced subgraphs. *Nord. J. Comput.*, 14(1-2):87–108, 2007.

[97] P. Hell and J. Huang. Certifying LexBFS recognition algorithms for proper interval graphs and proper interval bigraphs. *SIAM J. Disc. Math.*, 18(3):554–570, 2004.

[98] L. Hubert and P. Arabie. Comparing partitions. *J. of Classification*, 2(1):193–218, 1985.

[99] J. Iacono, R. Jacob, and K. Tsakalidis. External memory priority queues with decrease-key and applications to graph algorithms. In M. A. Bender, O. Svensson, and G. Herman, editors, *European Symp. on Algorithms ESA*, volume 144 of *LIPIcs*, pages 60:1–60:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ESA.2019.60.

[100] Intel Corporation. *Intel®64 and IA-32 Architectures — Software Developer's Manual — Volume 2*, 2019.

[101] 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.

[102] 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.

[103] J. JáJá. *An Introduction to Parallel Algorithms.* Addison Wesley, 1992.

[104] S. Janson. The probability that a random multigraph is simple. *Comb. Probab. Comput.*, 18(1-2):205–225, 2009. doi:10.1017/S0963548308009644.

[105] M. Jerrum and A. Sinclair. Fast uniform generation of regular graphs. *Theor. Comput. Sci.*, 73(1):91–100, 1990. doi:10.1016/0304-3975(90)90164-D.

[106] 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.

[107] R. Kannan, P. Tetali, and S. S. Vempala. Simple markov-chain algorithms for generating bipartite graphs and tournaments. *Random Struct. Algorithms*, 14(4):293–308, 1999.

[108] D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, 1995. doi:10.1145/201019.201022.

[109] T. Kawamoto and M. Rosvall. Estimating the resolution limit of the map equation in community detection. *Physical Review E*, 91:012809, 2015.

[110] J. H. Kim and V. H. Vu. Generating random regular graphs. *Comb.*, 26(6):683–708, 2006. doi:10.1007/s00493-006-0037-7.

[111] D. Kratsch, R. M. McConnell, K. Mehlhorn, and J. P. Spinrad. Certifying algorithms for recognizing interval graphs and permutation graphs. *SIAM J. Comput.*, 36(2):326–353, 2006.

[112] 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.

[113] 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.

[114] 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.

[115] A. Lancichinetti and S. Fortunato. Community detection algorithms: A comparative analysis. *Physical Review E*, 80(5), Nov 2009.

[116] 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.

[117] A. Lancichinetti, F. Radicchi, J. J. Ramasco, and S. Fortunato. Finding statistically significant communities in networks. *PLOS ONE*, 6(4):1–18, 04 2011. doi:10.1371/journal.pone.0018961.

[118] D. Lemire. Fast random integer generation in an interval. *ACM Trans. Model. Comput. Simul.*, 29(1):3:1–3:12, 2019. doi:10.1145/3230636.

[119] 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.

[120] D. A. Levin, Y. Peres, and E. L. Wilmer. *Markov chains and mixing times.* American Mathematical Society AMS, Providence, Rhode Island, 2009.

[121] N. V. Mahadev and U. N. Peled. *Threshold graphs and related topics.* Elsevier, 1995.

[122] 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.

[123] 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*, volume 2625 of *LNCS*, pages 36–61. Springer, 2002. doi:10.1007/3-540-36574-5_3.

[124] T. Maier, P. Sanders, and R. Dementiev. Concurrent hash tables: Fast and general(?)! *ACM Trans. Parallel Comput.*, 5(4):16:1–16:32, 2019. doi:10.1145/3309206.

[125] P. Massart. *Concentration inequalities and model selection*, volume 6. Springer, 2007.

[126] 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. doi:10.1145/272991.272995.

[127] R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. Certifying algorithms. *Comput. Sci. Rev.*, 5(2):119–161, 2011. doi:10.1016/j.cosrev.2010.09.009.

[128] 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.

[129] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In R. H. Möhring and R. Raman, editors, *European Symp. on Algorithms ESA*, volume 2461 of *Lecture Notes in Computer Science*, pages 723–735. Springer, 2002. doi:10.1007/3-540-45749-6_63.

[130] K. Mehlhorn and S. Näher. LEDA: A library of efficient data types and algorithms. In A. Kreczmar and G. Mirkowska, editors, *Mathematical Foundations of Computer Science MFCS*, volume 379 of *Lecture Notes in Computer Science*, pages 88–106. Springer, 1989. doi:10.1007/3-540-51486-4_58.

[131] K. Mehlhorn and S. Näher. LEDA: A platform for combinatorial and geometric computing. *Commun. ACM*, 38(1):96–102, 1995. doi:10.1145/204865.204889.

[132] 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.

[133] 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.

[134] U. Meyer, H. Tran, and K. Tsakalidis. Certifying induced subgraphs in large graphs. In C. Lin, B. M. T. Lin, and G. Liotta, editors, *WALCOM: Algorithms and Computation - 17th Int. Conference and Workshops, WALCOM 2023, Hsinchu, Taiwan, March 22-24, 2023, Proceedings*, volume 13973 of *Lecture Notes in Computer Science*, pages 229–241. Springer, 2023. doi:10.1007/978-3-031-27051-2_20.

[135] 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.

[136] R. Milo. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594):824–827, Oct 2002. doi:10.1126/science.298.5594.824.

[137] 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.

[138] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press, 2005. doi:10.1017/CBO9780511813603.

[139] 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.

[140] 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.

[141] R. Motwani and P. Raghavan. *Randomized algorithms.* Chapman & Hall/CRC, 2010.

[142] K. Munagala and A. G. Ranade. I/O-complexity of graph algorithms. In R. E. Tarjan and T. J. Warnow, editors, *ACM-SIAM Symp. on Discrete Algorithms SODA*, pages 687–694. ACM/SIAM, 1999.

[143] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003. doi:10.1137/S003614450342480.

[144] M. E. J. Newman. *Networks: An Introduction.* Oxford University Press, 2010. doi:10.1093/ACPROF:OSO/9780199206650.001.0001.

[145] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(026113):1–16, 2004.

[146] 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.

[147] S. D. Nikolopoulos and L. Palios. An o(nm)-time certifying algorithm for recognizing hhd-free graphs. *Theoret. Comput. Sci.*, 452:117–131, 2012.

[148] M. D. Penrose. *Random Geometric Graphs.* Oxford University Press, 2003. doi:10.1093/acprof:oso/9780198506263.001.0001.

[149] 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.

[150] M. Penschuck, U. Brandes, M. Hamann, S. Lamm, U. Meyer, I. Safro, P. Sanders, and C. Schulz. Recent advances in scalable network generation. In D. A. Bader, editor, *Massive Graph Analytics*, pages 333–376. CRC Press, 2022.

[151] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. *J. ACM*, 49(1):16–34, 2002. doi:10.1145/505241.505243.

[152] 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.

[153] 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.

[154] 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.

[155] 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.

[156] D. P. Rodgers. Improvements in multiprocessor system design. In T. F. Gannon, T. Agerwala, and C. V. Freiman, editors, *Proceedings of the Symp. on Computer Architecture ISCA*, pages 225–231. IEEE Computer Society, 1985. doi:10.1145/327070.327215.

[157] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In B. Bonet and S. Koenig, editors, *Proc. on Artificial Intelligence AAAI*, pages 4292–4293. AAAI Press, 2015.

[158] M. Rosvall, D. Axelsson, and C. T. Bergstrom. The map equation. *The European Physical J. Special Topics*, 178(1):13–23, 2009.

[159] 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.

[160] P. Sanders. Fast priority queues for cached memory. *ACM J. of Experimental Algorithmics*, 5:7, 2000. doi:10.1145/351827.384249.

[161] P. Sanders. Algorithm engineering - an attempt at a definition. In S. Albers, H. Alt, and S. Näher, editors, *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, volume 5760 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2009. doi:10.1007/978-3-642-03456-5_22.

[162] P. Sanders. Algorithm engineering - an attempt at a definition using sorting as an example. In G. E. Blelloch and D. Halperin, editors, *Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments, ALENEX 2010, Austin, Texas, USA, January 16, 2010*, pages 55–61. SIAM, 2010. doi:10.1137/1.9781611972900.6.

[163] 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 Advancesin Social Networks Analysis and Mining ASONAM*, pages 514–519. Assoc. for Computing Machinery ACM, 2015. doi:10.1145/2808797.2809400.

[164] W. E. Schlauch, E. Horvát, and K. A. Zweig. Different flavors of randomness: comparing random graph models with fixed degree sequences. *Soc. Netw. Anal. Min.*, 5(1):36:1–36:14, 2015. doi:10.1007/s13278-015-0267-z.

[165] D. Schultes. External memory minimum spanning trees. Bachelor's thesis, Saarland University, Germany, 2003.

[166] D. Schultes. External memory spanning forests and connected components. http://algo2.iti.kit.edu/dementiev/files/cc.pdf, 2003.

[167] J. F. Sibeyn. External connected components. In T. Hagerup and J. Katajainen, editors, *Algorithm Theory - SWAT 2004, Workshop on Algorithm Theory, Proceedings*, volume 3111 of *Lecture Notes in Computer Science*, pages 468–479. Springer, 2004. doi:10.1007/978-3-540-27810-8_40.

[168] F. J. Soulignac. A certifying and dynamic algorithm for the recognition of proper circular-arc graphs. *Theoret. Comput. Sci.*, 889:105–134, 2021.

[169] I. Stanton and A. Pinar. Constructing and sampling graphs with a prescribed joint degree distribution. *ACM J. Exp. Algorithmics*, 17(1), 2011. doi:10.1145/2133803.2330086.

[170] I. Stanton and A. Pinar. Sampling graphs with a prescribed joint degree distribution using markov chains. In M. Müller-Hannemann and R. F. F. Werneck, editors, *Workshop on Algorithm Engineering and Experiments ALENEX*, pages 151–163. SIAM, 2011. doi:10.1137/1.9781611972917.15.

[171] 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.

[172] A. Steger and N. C. Wormald. Generating random regular graphs quickly. *Comb. Probab. Comput.*, 8(4):377–396, 1999.

[173] S. H. Strogatz. Exploring complex networks. *Nature*, 410(6825):268, 2001.

[174] 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.

[175] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975. doi:10.1145/321879.321884.

[176] The Lemur Project. ClueWeb12 Web Graph, Nov 2013. http://www.lemurproject.org/clueweb12/webgraph.php.

[177] G. Tinhofer. On the generation of random graphs with given properties and known distribution. *Appl. Comput. Sci., Ber. Prakt. Inf*, 13:265–297, 1979.

[178] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the Facebook social graph. *CoRR*, abs/1111.4503, 2011. arXiv:1111.4503.

[179] 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.

[180] N. D. Verhelst. An efficient MCMC algorithm to sample binary matrices with fixed marginals. *Psychometrika*, 73(4):705–728, 2008.

[181] 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.

[182] F. Viger and M. Latapy. Fast generation of random connected graphs with prescribed degrees. *CoRR*, abs/cs/0502085, 2005. arXiv:cs/0502085.

[183] J. S. Vitter. An efficient algorithm for sequential random sampling. *ACM Trans. Math. Softw.*, 13(1):58–67, 1987. doi:10.1145/23002.23003.

[184] M. Yannakakis. Node-deletion problems on bipartite graphs. *SIAM J. Comput.*, 10(2):310–327, 1981. doi:10.1137/0210022.

[185] 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.

[186] J. Y. Zhao. Expand and contract: Sampling graphs with given degrees and other combinatorial families. *CoRR*, abs/1308.6627, 2013. arXiv:1308.6627.