Some Results on the Theory and Practice of Algorithms

Dissertation
zur Erlangung des Doktorgrades
der Naturwissenschaften

vorgelegt beim Fachbereich Informatik und Mathematik
der Goethe-Universität
in Frankfurt am Main

von
David Hammer

Frankfurt 2020

Datum der Disputation: 27.11.2020

Abstract

This thesis presents research which spans three conference papers and one manuscript which has not yet been submitted for peer review.

The topic of 1 is the inherent complexity of maintaining perfect height in B-trees. We consider the setting in which a B-tree of optimal height contains $n = (1 - \epsilon)N$ elements where N is the number of elements in full B-tree of the same height (the capacity of the tree). We show that the rebalancing cost when updating the tree—while maintaining optimal height—depends on ϵ. Specifically, our analysis gives a lower bound for the rebalancing cost of $\Omega(1/(\epsilon B))$. We then describe a rebalancing algorithm which has an amortized rebalancing cost with an almost matching upper bound of $\mathcal{O}(1/(\epsilon B) \cdot \log^2(\min\{1/\epsilon, B\}))$. We additionally describe a scheme utilizing this algorithm which, given a rebalancing budget $f(n)$, maintains optimal height for decreasing ϵ until the cost exceeds the budget at which time it maintains optimal height plus one. Given a rebalancing budget of $\Theta(\log n)$, this scheme maintains optimal height for all but a vanishing fraction of sizes in the intervals between tree capacities.

Manuscript 2 presents empirical analysis of practical randomized external-memory algorithms for computing the connected components of graphs. The best known theoretical results for this problem are essentially all derived from results for minimum spanning tree algorithms. In the realm of randomized external-memory MST algorithms, the best asymptotic result has I/O-complexity $\mathcal{O}(\text{sort}(|E|))$ in expectation while an empirically studied practical algorithm has a bound of $\mathcal{O}(\text{sort}(|E|) \cdot \log(|V|/M))$. We implement and evaluate an algorithm for connected components with expected I/O-complexity $\mathcal{O}(\text{sort}(|E|))$—a simplification of the MST algorithm with this asymptotic cost, we show that this approach may also yield good results in practice.

In paper 3, we present a novel approach to simulating large-scale population protocol models. Naive simulation of N interactions of a population protocol with n agents and m states requires $\Theta(n \log m)$ bits of memory and $\Theta(N)$ time. For very large n, this is prohibitive both in memory consumption and time, as interesting protocols will typically require $N > n$ interactions for convergence. We describe a histogram-based simulation framework which requires $\Theta(m \log n)$ bits of memory instead—an improvement as it is typically the case that $n \gg m$. We analyze, implement, and compare a number of different data structures to perform correct agent sampling in this regime. For this purpose, we develop *dynamic alias tables* which allow sampling an interaction in expected amortized constant time. We then show how to use sampling techniques to process agent interactions in batches, giving a simulation approach which uses subconstant time per interaction under reasonable assumptions.

With paper 4, we introduce the new model of *fragile complexity* for comparison-based algorithms. Within this model, we analyze classical comparison-based problems such as finding the minimum value of a set, selection (or finding the median), and sorting. We prove a number of lower and upper bounds and in particular, we give a number of randomized results which describe trade-offs not achievable by deterministic algorithms.

Zusammenfassung

Diese Dissertation befasst sich mit Forschungsergebnissen dreier Publikationen im Rahmen wissenschaftlicher Konferenzen und eines unveröffentlichten Manuskripts. Hierbei widmen sich Publikation 1 und Manuskript 2 beide dem übergefassten Thema der Algorithmen und Datenstrukturen für Externspeicher (External Memory). Spezifisch widmen wir uns in Publikation 1 der Problemstellung der Ausbalancierung bzw. des Rebalancings von B-Bäumen. Wir analysieren die Komplexität des Rebalancings nach durchgeführten Aktualisierungen während die optimale Höhe des Baums aufrecht erhalten wird. In Manuskript 2 implementieren, testen und optimieren wir effiziente, randomisierte External Memory Algorithmen zur Bestimmung der Zusammenhangskomponenten eines vollständig externen Graphen. Publikation 3 befasst sich mit einem Ansatz zur effizienteren Simulation von Population Protocols beim Vorliegen von großen Populationsgrößen. Zuletzt stellen wir in Publikation 4 algorithmische Ergebnisse für ein neues Komplexitätsmodell vor, welches wir als Fragile Complexity bezeichnen.

Publikation 1: *"On Optimal Balance in B-Trees: What Does It Cost to Stay in Perfect Shape?"* Hier widmen wir uns der inhärenten Komplexität des Erhalts einer idealen Höhe von B-Bäumen. Bei B-Bäumen [6] handelt es sich um balancierte Suchbäume, die zur Indizierung von Daten auf Datenträgern angewandt werden und somit vor allem für Datenbanken und Dateisystemen von Bedeutung sind. Eines der Hauptmerkmale von B-Bäumen ist hierbei deren hoher Verzweigungsfaktor, der zu einer geringen Höhe dieser Suchbäume führt.

Das standardmäßig angewandte Rebalancing Schema legt hierbei einen unteren Grenzwert für den Verzweigungsfaktor fest, welcher für gewöhnlich die Hälfte des oberen Grenzwertes des Verzweigungsfaktors beträgt. Erfolgt eine Überschreitung des oberen Grenzwerts wird der Baumknoten infolgedessen geteilt. Bei einer Unterschreitung des unteren Grenzwerts hingegen erfolgt eine Verschmelzung zwischen zwei Geschwisterknoten. Beide Operationen können sich hierbei nach oben hin ausweiten. Sollte dabei die Wurzel erreicht werden, kommt es im Falle einer Teilungsoperation zur Vergrößerung der Baumhöhe, wohingegen es im Falle einer Verschmelzungsoperation zu einer Verkleinerung kommt. Die Anwendung dieses Standardschemas ist sehr effizient und die Kosten eines Rebalancings sind höchstens logarithmisch. Erfolgt eine geringe Senkung des unteren Grenzwerts sind die amortisierten Rebalancingkosten konstant. Jedoch kann es im schlechtesten Fall dazu führen, dass die Höhe des Baums negativ beeinflusst wird. Somit liegt der Unterschied zwischen der idealen und der schlechtesten Höhe eines B-Baums unter Anwendung dieses Schemas in einem konstanten Faktor.

Da die Suchkosten innerhalb eines B-Baums proportional zur Höhe sind und die Höhe dieser Suchbäume üblicherweise relativ gering ist, hat eine suboptimale Höhe einen signifikanten Einfluss auf die Suchkosten. Der Fokus unserer Arbeit lag in der Analyse des Einflusses auf die Kosten des Rebalancings unter der Annahme, dass die Höhe eines B-Baums im idealen Zustand gehalten wird.

Wir betrachten den Zustand, in welchem ein B-Baum mit optimaler Höhe $n = (1 - \epsilon)N$ Elemente enthält, wobei N die Anzahl der Elemente in vollständigen B-Bäumen identischer Höhe darstellt. Wir zeigen, dass die Kosten eines Rebalancings bei einer Aktualisierung des Baums mit gleichzeitigem Erhalt der optimalen Höhe abhängig ist von ϵ. Konkret stellt unsere Analyse die Untergrenze der Kosten eines Rebalancings von $\mathcal{O}(1/(\epsilon B))$ dar. Wir beschreiben

hierzu einen Rebalancing Algorithmus, welcher amortisierte Kosten eines Rebalancings mit einer nahezu übereinstimmenden oberen Schranke von

$$\mathcal{O}\left(\frac{1}{\epsilon B}\cdot\log^2(\min\{1/\epsilon, B\})\right)$$

aufweist. Ferner wird ein Schema beschrieben, das unter Anwendung dieses Algorithmus und unter Voraussetzung eines Rebalancing Budgets $f(n)$ die ideale Höhe für ein sinkendes ϵ so lange erhält, bis die Kosten das Budget übersteigen, in welchem Fall die ideale Höhe plus eins beibehalten wird. Eine offensichtliche Wahl für das Rebalancing Budget stellt $\mathcal{O}(\log n)$ dar, da dies den Suchkosten in diesem Baum entspricht. Mit diesem Rebalancing Budget erhält dieses Schema die ideale Höhe für den Großteil des Intervalls zwischen den Baumkapazitäten aufrecht. Soweit es uns bekannt ist, handelt es sich hierbei um das erste Rebalancing Schema, welches garantiert, dass die Höhe des B-Baumes maximal bei der idealen Höhe plus einer Konstante gehalten wird.

Manuskript 2: *"Engineering an Expected $O(\mathbf{Sort}(m))$ External Connected Components Algorithm."* Im Rahmen unseres Manuskripts präsentieren wir eine empirische Analyse angewandter, randomisierter External Memory Algorithmen zur Berechnung von Zusammenhangskomponenten von Graphen. Die Partitionierung eines Graphen in dessen Zusammenhangskomponenten stellt eine der einfachsten Berechnungen dar, welche auf einem Graphen durchgeführt werden kann. Bezogen auf Internspeicher (Internal Memory) ist die Berechnung der Zusammenhangskomponenten trivial in linearer Zeit unter Verwendung von jeglichen Graphtraversierungen durchfürbar, wie zum Beispiel durch Breitensuche (Breadth First Search, BFS) oder Tiefensuche (Depth First Search, DFS).

Im External Memory hingegen ist die Berechnung nicht-trivial und die bekannten theoretischen Ergebnisse für diese Problemstellung beziehen sich hier meist auf das kompliziertere minimale Spannbaum (MST) Problem. Die derzeit beste asymptotische Schranke für deterministische Algorithmen [5] liegt bei einer I/O-Komplexität von

$$\mathcal{O}\left(\text{sort}(|E|)\log\log\left(\frac{|V|B}{|E|}\right)\right).$$

Mit $\text{sort}(N)$ beschreiben wir die I/O-Komplexität des Sortierungsvorgangs von N Elementen im External Memory Modell [2]:

$$\text{sort}(N) = \Theta\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$$

Im Bereich der randomisierten External Memory MST Algorithmen weist das beste asymptotische Ergebnis eine I/O-Komplexität von $\mathcal{O}(\text{sort}(|E|))$ auf [7, 1]. Dies wird erreicht, indem der Algorithmus mit erwarteter linearer Zeit von Karger, Klein und Tarjan (KKT) [8] für External Memory adaptiert wird. Die MST Version des KKT Algorithmus ist hierbei auf einen nicht-trivialen Verifikationsalgorithmus angewiesen. Weiterhin nutzt er Borůvka Schritte zur Knotenkontraktion.

Es gibt jedoch auch einen deutlich simpleren Algorithmus nach Sibeyn und Meyer (SM) [9], der sich empirisch als effizient erwiesen hat und eine erwartete I/O-Komplexität von

$$\mathcal{O}\left(\text{sort}(|E|) \cdot \log\left(\frac{|V|}{M}\right)\right)$$

aufweist. Dieser SM Algorithmus verwendet weiterhin eine effiziente Knotenkontraktionsstrategie basierend auf Time Forward Processing. Im Falle der Berechnung der Zusammenhangskomponenten entfällt jedoch für den KKT Algorithmus ebenfalls die Notwendigkeit des Verifikationsalgorithmus; dieser kann hier durch simples Relabelling ersetzt werden.

In der in Manuskript 2 präsentierten Arbeit implementieren, evaluieren und optimieren wir verschiedene Varianten des KKT Algorithmus für das Bestimmen der Zusammenhangskomponenten eines Graphen. Wir erwägen verschiedene Knotenkontraktionsalgorithmen zur Nutzung innerhalb des KKT Algorithmus. Die übliche Wahl zur Knotenkontraktion liegt in der Verwendung von Borůvka Schritten. Wir beschreiben und analysieren eine simple, randomisierte Modifikation eines Borůvka Schrittes. Der Vorteil dieser randomisierten Modifikation liegt darin, dass sie nur ein einfaches Relabelling erfordert, welches mittels zweier Sortierungsschritte erreicht wird. Der Nachteil jedoch liegt darin, dass der Vorgang eine Anzahl an Kanten bereits vor der Kontraktion verwirft, was zu weniger Kontraktion pro Schritt verglichen mit gewöhnlicher Borůvka Knotenkontraktion führt. Eine dritte Option liegt in der Nutzung von SM zur Knotenkontraktion. Für die Analyse des KKT Algorithmus muss der Knotenkontraktionsalgorithmus die Anzahl der Knoten um einer konstanten Bruchteil reduzieren, in welchem Fall SM erwartete I/O Kosten von $\mathcal{O}(\text{sort}(|E|))$ aufweist [9]. In unserer Arbeit stellen wir fest, dass SM sich als effizientester Weg zur Knotenkontraktion erweist.

Für eine Menge an Graphen vergleichen wir die Performance von Borůvkas Algorithmus, SM und den von uns erwogenen, verschiedenen Versionen des KKT Algorithmus. Im Allgemeinen lässt sich feststellen, dass SM eine hohe Effizienz aufweist und den schnellsten Algorithmus zur Verwendung bei sehr dünnen Graphen darstellt. Jedoch weisen wir nach, dass Varianten des KKT Algorithmus diese Effizienz schlagen, wenn die Graphendichte einen bestimmten Grenzwert überschreitet. Dieser Grenzwert variiert zwischen verschiedenen Graphenklassen und liegt bei etwa $|E|/|V| = 5$; er scheint niedriger für Gilbert-Graphen als für zufällige, hyperbolische Graphen zu sein.

Publikation 3: *"Simulating Population Protocols in Sub-Constant Time per Interaktion."* Wir erläutern im Rahmen dieser Arbeit einen neuen Ansatz für die Simulation von umfassenden Population Protocol Modellen [3, 4]. Innerhalb eines Population Protocols betrachten wir eine Gruppe von n Agenten, wobei jeder eine Instanz eines endlichen Automaten mit einer Menge von m Zuständen darstellt. Die Berechnung eines Protokolls erfolgt hierbei über die Interaktion zwischen Agenten. Eine Übergangsfunktion spezifiziert die resultierenden Zustände für jede mögliche Interaktion zweier Agenten. Im Standardmodell werden je Interaktion zwei Agenten uniform zufällig ausgewählt.

Für unterschiedliche Fragestellungen auf technischen Systemen wurde Forschung zur Erstellung von Protokollen durchgeführt, die nach einer möglichst geringen Anzahl an Interaktionen konvergieren und gleichzeitig möglichst wenige Zustände nutzen. Die Simulation

solcher Protokolle stellt hierbei ein sinnvolles Tool dar, um die Analyse zu steuern. Jedoch erfordert die empirische Unterscheidung zwischen polylogarithmischen Termen die Simulation von immensen Populationsgrößen.

Die Durchführung einer einfachen Simulation von N Interaktionen eines Population Protocols mit n Agenten und m Zuständen erfordert $\mathcal{O}(n \log m)$ Speicher und $\mathcal{O}(N)$ Zeit. Für sehr große Werte von n ist dieser Vorgang äußerst kostspielig bezüglich der Nutzung von Speicher und Zeit, da interessante Protokolle üblicherweise $N > n$ Interaktionen für eine Konvergenz benötigen.

Wir beschreiben einen Weg um die Konfiguration als Histogramm der Zustände zu repräsentieren. Diese alternative Darstellung benötigt hierbei lediglich $\mathcal{O}(m \log n)$ Speicher. Dies stellt eine Verbesserung dar, da üblicherweise von $n \gg m$ ausgegangen werden kann.

Die Auswahl von zufälligen Agenten basierend auf dem Histogramm stellt jedoch eine größere Herausforderung dar. Wir beschreiben eine Anzahl von Datenstrukturen, die ein solches Sampling zweier uniform zufälliger Agenten ermöglichen. Ein lineares Array von Zuständen lediglich zu erhalten, führt zu zufälligem Sampling in $\mathcal{O}(m)$ Zeit. Eine weitere Option liegt darin, das Histogramm in einem balancierten, binären Baum zu speichern, was einen $\mathcal{O}(\log m)$ Preis für das Sampling und das Aktualisieren der Frequenzen erfordert. Als dritte Option präsentieren wir eine neue Datenstruktur, die wir als Dynamic Alias Table bezeichnen. Hierbei handelt es sich um eine simple Erweiterung bereits bekannter Alias Tables, welche effizient implementiert werden können. Die amortisierten Sampling-Kosten zufälliger Agenten unter Verwendung unserer Dynamic Alias Tables liegen in Erwartung bei $\mathcal{O}(1)$.

Wir präsentieren anschließend einen Ansatz zur Verarbeitung von Batches von Agenteninteraktionen. Dies führt zu einem Simulationsverfahren für die Verarbeitung von Agenteninteraktionen, welches unter vernünftigen Rahmenbedingungen $o(N)$ Zeit benötigt. Die Kernobeobachtung, auf die wir uns für unseren Ansatz berufen liegt darin, dass wir eine Menge von unabhängigen Interaktionen effizient zeitgleich ausführen können. Hierzu wird ausgenutzt, dass die Agenten uniform zufällig gezogen werden; eine Sequenz ist unabhängig, wenn kein Agent doppelt in der Sequenz vorliegt. Wir analysieren die Verteilung der Längen solcher Sequenzen und zeigen, dass diese in Erwartung mindestens eine Länge von $\Omega(\sqrt{n})$ aufweisen. Liegt eine solche unabhängige Sequenz vor und sind die Frequenzen der Zustände der darin enthaltenen Agenten bekannt, so können wir in $\mathcal{O}(m^2)$ Zeit das Histogramm gemäß der Interaktionen innerhalb dieser Sequenz aktualisieren.

Unser Batch Processing Ansatz funktioniert hierbei wie folgt: wir ziehen zunächst die Länge der nächsten unabhängigen Sequenz. Hierzu nutzen wir die Inversionsmethode, um diesen Vorgang in $\mathcal{O}(\log n)$ Zeit abzuschließen. Daraufhin ziehen wir gemäß einer multivariaten hypergeometrischen Verteilung die Frequenzen jedes Zustandspaares. Das Histogramm wird nun entsprechend der Transitionsfunktion für alle Zustandspaare aktualisiert. Anschließend ziehen wir zufällig aus der Gruppe der Agenten innerhalb der Sequenz, um eine Kollision hervorzurufen. Auf diese Weise simulieren wir erwartet $\Omega(\sqrt{n})$ viele Interaktionen in $\mathcal{O}(\log n + m^2)$ Zeit.

Publikation 4: *"Fragile Complexity of Comparison-Based Algorithms."* Im Rahmen dieser Publikation führen wir die Fragile Complexity als neues Modell für vergleichsbasierte

Algorithmen ein. Für gewöhnlich liegt der Fokus der Algorithmusanalyse auf der Gesamtanzahl der Vergleiche, die ein Algorithmus durchführt. Hinter dem Fragile Complexity Modell steht der Gedanke, stattdessen die individuellen Elemente und die Anzahl an Vergleichen zu betrachten, in die jedes einzelne Element involviert ist. Somit definieren wir die Fragile Complexity eines Algorithmus als maximale Anzahl an Vergleichen, die für jedes individuelle Element durchgeführt werden. Dieses Modell weist immer dann eine hohe Relevanz auf, wenn die durchgeführten Vergleiche selbst einen Einfluss auf die Elemente haben, welche vergleichen werden.

Exemplarisch lässt sich dieser Zusammenhang gut anhand der Bestimmung einer Rangfolge für eine Reihe von Genussmitteln wie zum Beispiel Wein erläutern. Jeder Vergleich erfordert hierbei ein Testen beider Proben, die verglichen werden sollen. Da es nur eine limitierte Menge von jeder Probe gibt, muss hierbei sichergestellt werden, dass keine der Proben erschöpft wird, bevor der Test vollständig abgeschlossen werden konnte. Dieser Vergleich zeigt, dass ein solches Ranking nur durchgeführt werden kann, wenn die Fragile Complexity des Rankingalgorithmus die Menge einer jeden vorliegenden Probe nicht übersteigt.

Im Rahmen unserer Arbeit analysieren wir innerhalb des Modells der Fragile Complexity klassische, vergleichsbasierte Fragestellungen. Für die Bestimmung des Minimalwerts weisen wir eine scharfe Schranke von $\Theta(\log n)$ für deterministische Algorithmen nach. Interessanterweise können wir auch zeigen, dass randomisierte Algorithmen die Fragile Complexity für das minimale Element verringern können, wenn selbige gleichzeitig für alle anderen Elemente erhöht wird. Ein Beispiel für einen solchen Kompromiss liefert eine erwartete Fragile Complexity von $\mathcal{O}(1)$ für das minimale Element und $\mathcal{O}(n)$ für alle anderen Elemente. Dieser Kompromiss wird durch einen Sampling basierten Algorithmus erreicht. Wir beweisen auch eine nicht-triviale untere Schranke, die diesen Kompromiss charakterisiert. Diese untere Schranke besagt, dass sofern die Fragile Complexity aller Elemente $\mathcal{O}(\Delta)$ nicht überschreitet, so muss die Fragile Complexity des minimalen Elements mindestens $\Omega(\log_\Delta n)$ betragen.

Im Bezug auf die Problemstellung der Selektion und Sortierung wird logarithmische Fragile Complexity trivial erreicht durch Simulation von AKS Sortiernetzwerken. Die Größe solcher Netzwerke beträgt $\Theta(n \log n)$, was bedeutet, dass die Simulation eine Zeitkomplexität von $\Theta(n \log n)$ benötigt. Wir zeigen wie das Selektionsproblem deterministisch mit $\Theta(\log n)$ Fragile Complexity und $\Theta(n)$ Zeitkomplexität durchgeführt werden kann. Unter Zuhilfenahme eines bestehenden Ergebnisses zu Sortiernetzwerke, konnten wir nachweisen, dass jedes Sortiernetzwerk, welches Selektion durchführt, eine Größe $\Omega(n \log n)$ haben muss. Weiterhin stellen wir einen randomisierten Selektionsalgorithmus vor, der eine Anzahl an Kompromissen zwischen der Fragile Complexity des Medians und allen anderen Elementen ermöglicht.

Bezüglich der Sortierung ist eine logarithmische Fragile Complexity und $\Theta(n \log n)$ Zeitkomplexität asymptotisch optimal. Wir weisen nach, dass jeglicher deterministischer, Mergebasierter Algorithmus eine Fragile Complexity von mindestens $\Omega(\log^2 n)$ aufweisen muss. Ein gewöhnlicher Mergesort wird eine lineare Fragile Complexity haben, jedoch zeigen wir, wie man den Algorithmus mit Hilfe einer Modifikation im Merge-Vorgang so anpassen kann, dass die untere Schranke erreicht wird. Interessanterweise wird ein gewöhnlicher Mergesort mit hoher Wahrscheinlichkeit eine Fragile Complexity von $\mathcal{O}(\log n)$ aufweisen, wenn der Input vor dem Sortieren durchmischt wird, sodass er in einer rein zufälligen Permutation vorliegt.

Resumé

Denne afhandling præsenterer forskningsresultater, der spænder over tre konferenceartikler samt et manuskript, vi endnu ikke har indsendt til en konference.

Artikel 1 undersøger den iboende kompleksitet for rebalancering af B-trær med optimal højde. Betragt et B-træ af optimal højde, der indeholder $n = (1 - \epsilon N)$ elementer, hvor N er antal elementer et fuldt B-træ af samme højde ville have (kapaciteten for B-træet). Vi viser en sammenhæng mellem prisen for rebalancering (efter indsættelse af et nyt element) og parameteren ϵ. Mere konkret giver vi en nedre grænse på rebalanceringskompleksiteten i denne situation på $\Omega(1/(\epsilon B))$. Vi beskriver dernæst en rebalanceringsalgoritme, hvis amortiserede kompleksitet næsten matcher den nedre grænse: $\mathcal{O}(1/(\epsilon B) \cdot \log^2(\min\{1/\epsilon, B\}))$. Med denne algoritme beskriver vi et system, der—givet et rebalanceringsbudget $f(n)$—kan opretholde optimal træhøjde efterhånden som ϵ går mod 0 indtil omkostningen overstiger budgettet, hvorefter højden holdes på optimal plus én. Hvis rebalanceringsbudgettet er $\Theta(\log n)$, vil dette system holde optimal højde for størsteparten af størrelsesintervallerne mellem trækapaciteter for voksende n: brøkdelen af intervallerne, hvor højden bliver optimal plus én, vil være aftagende.

Manuskript 2 præsenterer empirisk analyse af praktiske algoritmer til at finde sammenhængskomponenter for grafer i ekstern hukommelse. De bedste teoretiske resultater for dette problem kommer alle fra algoritmer til at finde minimale udspændende træer. Blandt MST-algoritmer er den bedst kendte asymptotiske I/O-kompleksitet $\mathcal{O}(\text{sort}(|E|))$ i forventning, mens den mest lovende praktiske algoritme har en forventet asymptotisk I/O-kompleksitet på $\mathcal{O}(\text{sort}(|E|) \cdot \log(|V|/M))$. Vi implementerer og tester en algoritme til sammenhængskomponenter, der har forventet I/O-kompleksitet $\mathcal{O}(\text{sort}(|E))$ (en simplificeret version af MST-algoritmen med denne I/O-kompleksitet). Vi viser, at den er kompetitiv for adskillige grafer.

I artikel 3 præsenterer vi en ny algoritme til effektivt at simulere population protocols med ekstremt store populationer. Naiv simulering af N interaktioner i en protokol med n agenter med hver m tilstande kræver $\Theta(n \log m)$ bits hukommelse og har en køretid på $\Theta(N)$. Dette skaber udfordringer for simuleringer med store værdier for n; både mht. hukommelse (n kan overstige RAM) og mht. køretid, da interessante protokoller ofte bruger $\Omega(\text{poly}(n))$ interaktioner førend de konvergerer. Vi beskriver en alternativ måde at repræsentere konfigurationen for en populationsprotokol (grundlæggende repræsenteres konfigurationen som et histogram over tilstande), der kræver $\Theta(m \log n)$ bits hukommelse i stedet—en markant forbedring da det typisk gælder at $n \gg m$. Vi analyserer, implementerer, og evaluerer mulige datastrukturer til at opbevare og trække tilfældige elementer fra denne histogram-repræsentation. Blandt disse strukturer udvikler vi *dynamic alias tables*, som har forventet tidskompleksitet $\Theta(N)$ til brug i denne sammenhæng. Derefter beskriver vi en teknik til at simulere store partier af interaktioner på samme tid, hvilket lader os simulere N interaktioner med $o(N)$ operationer under passende betingelser.

Med artikel 4 præsenterer vi modellen *fragile complexity* for sammenligningsbaserede algoritmer. I denne nye model analyserer vi en række klassiske problemer såsom at finde mindste element, selection (eller at finde medianen) og sortering. Vi viser en række nedre og øvre grænser, heriblandt en række resultater for randomiserede algoritmer, der adskiller sig fra vores deterministiske resultater.

Acknowledgements

I would like to first thank my supervisors, Rolf Fagerberg and Ulrich Meyer. They made this project possible and have worked hard to give me excellent working conditions and opportunities during this project.

As part of the PhD program, I spent one and a half years at the Goethe-Universität in Frankfurt am Main. I would like to thank the members of the algorithm engineering group there for making me feel welcome during my stay and for providing interesting research opportunities. Manuel, Hung, Alex, and Claudia—thank you for the helpful discussions and many lunches.

Lastly, I would like to thank my family and my girlfriend Nina. They have been very supportive of me over the last years.

List of publications

The following papers and manuscripts are included in this cumulative thesis:

1. Rolf Fagerberg, **David Hammer**, and Ulrich Meyer. "On Optimal Balance in B-Trees: What Does It Cost to Stay in Perfect Shape?" In 30th International Symposium on Algorithms and Computation (ISAAC 2019), 149:35:1–35:16. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019. https://doi.org/10.4230/LIPIcs.ISAAC.2019.35.

   Introduced in section 1.2 and included in full in chapter 2.

2. Gerth Stølting Brodal, Rolf Fagerberg, **David Hammer**, Ulrich Meyer, Manuel Penschuck, and Hung Tran. "Engineering an Expected $O(\text{Sort}(m))$ External Connected Components Algorithm." 2020.

   Introduced in section 1.3 and included in chapter 3. This manuscript describes ongoing work and has not yet been submitted for peer review.

3. Petra Berenbrink, **David Hammer**, Dominik Kaaser, Ulrich Meyer, Manuel Penschuck, and Hung Tran. "Simulating Population Protocols in Sub-Constant Time per Interaction." In 28th Annual European Symposium on Algorithms (ESA 2020), 173:16:1–16:22. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. https://doi.org/10.4230/LIPIcs.ESA.2020.16.

   Introduced in section 1.4 and included in full in chapter 4.

4. Peyman Afshani, Rolf Fagerberg, **David Hammer**, Riko Jacob, Irina Kostitsyna, Ulrich Meyer, Manuel Penschuck, and Nodari Sitchinava. "Fragile Complexity of Comparison-Based Algorithms." In 27th Annual European Symposium on Algorithms (ESA 2019), 144:2:1–2:19. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. https://doi.org/10.4230/LIPIcs.ESA.2019.2.

   Received best paper award at ESA 2019 (track A).

   Introduced in section 1.5 and included in full in chapter 5.

Contents

1 Introduction

This thesis covers results spanning multiple distinct topics: external-memory algorithms, population protocols, and fragile complexity. At its outset, the focus of the PhD project was external memory algorithms. Among the results presented, the ones which fit this description include the results on B-trees which we introduce in section 1.2 and the work on connected components in external memory which we introduce in section 1.3. However, in working with colleagues at the algorithm engineering group at Goethe Universität Frankfurt am Main, other algorithmic problems came up which inspired additional lines of research, leading to paper 3 and 4 presented in this thesis.

The following sections provide a short introduction of the external memory model. The formal model we consider for external memory analysis is described along with some basic results in section 1.1. Two areas are touched upon in particular by the included papers: B-trees and graph algorithms. Section 1.2 presents a summary introduction to B-trees, introducing the work presented in paper 1. Section 1.3 gives a compact overview of some of the challenges in designing graph algorithms for external memory, describing the context for our ongoing work on computing connected components in external memory presented in manuscript 2.

The remaining two papers included in this thesis deal with simulation of population protocols and fragile complexity. To motivate the former, we describe briefly the population protocol model in section 1.4 before describing the main techniques used and interesting results presented in paper 3.

For the latter, we introduce the model of fragile complexity in section 1.5. As this is a novel new model, the introduction focuses on motivating and describing the model before summarizing some of the interesting results found in 4. Some of these results are contrasted with existing results for comparison networks to compare the properties of the two models.

Chapters 2 to 5 consist of the included papers mentioned in the publication list.

1.1 Algorithms for external memory

A fundamental challenge faced when processing sizeable datasets is the potential for data transfer to become a bottleneck. Even with modern solid state drives, there still remains an order-of-magnitude gap between latency for RAM and latency for disk accesses. Generally, memory systems offer trade-offs between size, speed, and price due to simple physical limitations, which has encouraged hardware manufacturers to add several intermediate cache levels. In modern consumer systems, CPUs typically have three levels of cache, L1 to L3, with increasing size and latency which cache data from RAM. To mitigate the effect of increased latency, data is typically transferred in blocks (or cache lines) between different levels of the memory hierarchy. For a thorough introduction to algorithms dealing with memory hierarchies, we refer to [45].

The rest of this section introduces the specific model we use in analyzing algorithms operating on external memory and some results relevant for the rest of this thesis.

For theoretical analysis of external-memory algorithms, we will use the external memory model of Aggarwal and Vitter [3], sometimes referred to as the I/O model or the disk access model. In this model, computation can only be done on (the subset of) data stored in internal memory which has a limited size M. Data is moved between internal memory and an unbounded external memory by input/output operations (I/Os), each of which moves blocks of B elements at a time. The model is illustrated in fig. 1.1. Motivating this model is the assump-

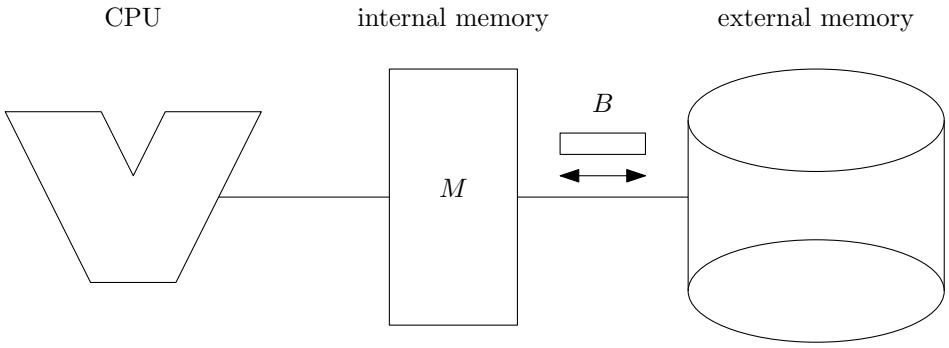


Figure 1.1: Illustration of the I/O model

tion that I/Os are expensive (CPU work is ignored) and that the block size B is rather large. Optimizing for I/O-efficiency involves arranging computation to exploit the B elements transferred per I/O. One basic operation which is trivially I/O-efficient is scanning N consecutive elements in order; this naturally requires $\text{scan}(N) = \lceil N/B \rceil$ I/Os. Simple queues and stacks implemented with consecutive arrays are efficient, too, requiring $\mathcal{O}(\text{scan}(N))$ I/Os to process N operations.

Another basic operation is sorting. Upper and lower bounds for comparison-based sorting in external memory were presented in [3]: sorting N elements requires

$$\text{sort}(N) = \Theta\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$$

I/Os. As we will see examples of in section 1.3, several more complex operations can be expressed by combining sorting and scanning steps. For reasonable values of M, B, and N, sorting is strongly preferable to random I/Os; $\text{scan}(N) < \text{sort}(N) \ll N$.

A number of data structures have been developed specifically for external memory. Paper 1 presented in this thesis deals with the rebalancing complexity of B-trees, which are search trees adapted for external memory (although they predate the particular formal I/O model we consider). B-trees and the particular work we have done on them is described in section 1.2.

Priority queues are a cornerstone of several graph algorithms. Significant work has gone into developing priority queues for external memory. For applications where the decrease-key operation is not needed, there are priority queues which are asymptotically optimal by the obvious bound imposed by sorting; N operations takes $\mathcal{O}(\text{sort}(N))$ I/Os [11, 40]. If decrease-key is required, the problem becomes more complicated. A recent lower bound [29] shows that

the same I/O bound (which is amortized $\mathcal{O}(1/B \log_{M/B} N/M)$) cannot be achieved for priority queues with the decrease-key operation. For a long time, the best upper bound was achieved with tournament-tree based priority queues requiring $\mathcal{O}(1/B \log_2 N/M)$ (amortized) I/Os per operation [40]. A recent more advanced structure [36] performs insert and decrease-key in $\mathcal{O}(1/B \log_{M/B} N/B)$ amortized I/Os and extract-min and delete key in

$$\mathcal{O}\left(\left\lceil \frac{M^\epsilon}{B} \log_{\frac{M}{B}} \frac{N}{B} \right\rceil \log_{\frac{M}{B}} \frac{N}{B}\right)$$

amortized I/Os.

One particular use for priority queues in external memory algorithms is time-forward processing. This refers to an algorithmic framework where data needed for later computation is sent forward in "time" (a key associated with data inserted into a priority queue) to the point at which it will be needed. Time-forward processing is applicable when the structure of the problem allows it to be cast as traversal of a DAG with a known topological ordering. Several graph algorithms employ time-forward processing—one example is the Sibeyn-Meyer algorithm [48] which we mention in section 1.3 and test empirically in manuscript 2.

In section 1.3, we introduce the challenges posed by graph traversal in external memory which is the context for our work in manuscript 2.

1.2 Keeping B-trees in shape

1.2.1 Preliminaries

B-trees were first introduced in 1970 by Bayer and McCreight [14, 15]. Suitable for indexing data stored on disk, they quickly gained traction in applications (described as ubiquitous in a 1979 survey [27]). They are found in different variants across numerous databases and virtually all modern file systems, including NTFS (Windows), HFS+ and APS (macOS), as well as Btrfs and ext4 (Linux). In this brief introduction, we will distinguish between the elements (also called records) stored in a tree and their search keys. The search keys are assumed to have a fixed size and to have a total order.

We will broadly consider B-trees to be multi-way search trees with fan-out at most B in which all leaf nodes lie on the same level. For simplicity, we in our work focus on leaf-oriented trees which means that all elements stored in the tree are stored in the leaf nodes. Each leaf node contains up to B elements. Internal nodes only serve to guide searches. An internal node contains up to $B - 1$ separator keys which partition the key space of the subtrees and up to B pointers to their children.

We will use *ordinary B-trees* to refer to B-trees where internal nodes have fan-out at least $\lceil B/2 \rceil$—except the root which only needs to have at least two children—and the tree balance (the upper and lower bound on node usage) is maintained as follows: If an insertion overfills a node, it is *split* in two. As splitting a node in two requires adding a separator key (and an additional child pointer) to the parent node, a split may overfill the parent and can propagate upwards towards the root. The height of the tree increases if the root node is split. The opposite of a split is a *merge*; due to the relationship between the upper and lower bounds, combining

an underfull node and a node exactly matching the lower bound results in one full node. As this decreases the degree of the parent, this can also propagate upwards. In case a deletion renders a node underfull while its neighbor has content to spare, a *share* operation can be used instead, equivalent to first merging and then splitting. A share operation does not propagate upwards.

Do note that the parameter B in this section only directly refers to the fan-out of B-trees rather than block size in the I/O model. For external-memory applications, the node size of a B-tree should be chosen appropriately for the block size of the storage system and the node size will typically be proportional to the block size. Therefore, although we do not explicitly discuss I/O complexity in this particular section, assuming that loading one node into memory generally costs $\Theta(1)$ I/Os, all results translate directly into I/O complexity. As such, the discussion can be focused on the tree structure. In the context of B-trees, the measure for (rebalancing) cost will be the number of nodes which need to be changed.

1.2.2 Motivation

Though B-trees have been studied intensely, we address a hitherto unanswered question about their inherent rebalancing complexity. Suppose one wants to keep a tree at optimal height in a dynamic setting—something not guaranteed by ordinary B-trees. One could ensure optimal height by rebuilding the tree after every update, though this would be prohibitively costly. Rearranging the contents of the tree locally instead will intuitively grow harder as the tree grows closer to being full (that is, containing the maximum number of records for its current, optimal height). The focus of our work is analyzing the relationship between how full the tree is and how costly rebalancing to maintain perfect height becomes.

For the model of B-trees we use in our analysis, a tree of height h can contain up to B^h elements. If a tree has optimal height h, then $B^{h-1} < n \leq B^h$. We consider B^h to be the capacity of a tree for its current height h, and denote it by $N = B^h$. Then, the amount of free space left in the tree is $N - n$. Our complexity results are expressed as a function of the ratio ϵ of free space relative to the capacity; $n = (1-\epsilon)N$. Intuitively, rebalancing (while maintaining optimal height) gets harder as ϵ approaches zero.

To justify this focus on optimal height, we recall that the I/O-cost of searching in a B-tree depends on the height of the tree. Typical applications have rather large block sizes and as such, practical B-trees will typically have rather high fan-out and very low height. Therefore, even a small improvement in height could have a non-trivial impact on cost of searching. From an I/O-perspective, this is further amplified by the fact that most of the internal layers of the tree fit into cache, meaning that disk access is only needed for very few bottom layers. In such cases, reducing the height by even a single layer could lower the amount of cache misses per lookup. For applications where the number of searches dominate the number of updates, it may be worthwhile to spend more time per update in order to achieve better height.

Note that the optimal height is $\lceil \log_B n \rceil$. Ordinary B-trees keep fan-out in a range from $\lceil B/2 \rceil$ to B (with the root being allowed to have fan-out of only two). In the worst case, an ordinary B-tree will therefore have worst-case height $\left\lfloor 1 + \log_{\lceil B/2 \rceil} n/2 \right\rfloor$. Compared to the optimal height of $\lceil \log_B n \rceil$, the change in the base of the logarithm increases the height by a

constant factor (not accounting for the potentially even worse fan-out for the root node and rounding).

As we will briefly review in section 1.2.3, the focus in much of the previous work on B-trees has been to increase storage utilization which is equivalent to increasing the fan-out. This typically entails designing schemes which guarantee that the storage utilization is kept at least some constant fraction, preferably higher than $1/2$ which ordinary B-trees achieve. Guaranteeing this, however, only improves the constant factor of the worst-case height.

Another motivation to investigate the trade-off is that this would provide a natural generalization for results known for binary trees. For binary trees, the upper and lower bounds are known to be tight (see [31] and [32]). Specifically, there is a lower bound stating that when the available space ratio is ϵ, then there exists an update such that rebalancing the tree to accommodate this update will require $\Omega(1/\epsilon)$ nodes to be changed [31]. A matching amortized upper bound is presented in [32]. The upper bound is given by a rebalancing scheme which requires $O(1/\epsilon)$ amortized node changes per update. Considering these results, an obvious question is whether similar bounds can be achieved for B-trees.

1.2.3 Previous work

A lot of work on B-trees has been done to create different variants with improved *storage utilization*: the average over all nodes allocation of the fraction fields. Given that the amount of disk space allocated for a node is typically fixed, low storage utilization directly implies wasting disk space. Additionally, low storage utilization for internal nodes implies lower average fan-out which potentially increases the height of the tree, slowing down searches. Variants found in literature typically present different trade-offs between storage utilization and rebalancing complexity. As discussed in the previous section, ordinary B-trees have a worst-case storage utilization of approximately roughly $1/2$ and their rebalancing cost by $\mathcal{O}(\log_B n)$.

If the goal is to reduce the cost of rebalancing, a small decrease in storage utilization has a significant effect. In (a, b) trees (which denotes B-trees with lower bound a and upper bound b on fan-out), decreasing a to $a < \lceil b/2 \rceil$ yields amortized constant rebalancing cost [35, 43]. If the gap between a and b is increased, the amortized rebalancing cost is even subconstant.

Another approach to improving update times is to add buffers. A noteworthy example of a structure doing this is B$^\varepsilon$-trees which give a scale of trade-offs between query and update cost [19]. The parameter ε in these trees determines the amount of space internal nodes use for buffering and how much they use for separator keys. With fan-out $\mathcal{O}(B^\varepsilon$, the height of the tree becomes $\mathcal{O}(1/\varepsilon \log_B n)$, but the amortized update cost drastically lowered: $\mathcal{O}(1/B^{1-\varepsilon} \log_B(N))$. Given this trade-off, these trees have been adopted in high-performance databases and seem promising for use in file systems [37]. In our work, we focus on pure B-trees and as such avoid considering the use of buffers, however.

An obvious idea for improving the worst-case storage utilization of B-trees beyond $1/2$ is to avoid splitting as often by first attempting to share node loads among siblings—Bayer and McCreight [15] themselves discuss such an *overflow* technique. B-trees with this adaptation are referred to as B*-trees by Knuth [39] and have worst-case storage utilization of approximately $2/3$ instead of $1/2$.

The overflow approach can be extended (also suggested in [15]) to consider larger groups for

load sharing—naturally, larger groups will lead to better storage utilization at the cost of more expensive rebalancing. Work has also been done to investigate average rather than worst-case storage utilization for these approaches. For instance, [41] shows that the storage utilization under random insertions approaches $m \ln((m+1)/m)$ with m being the group size used in the overflow method.

While the generalized overflow approach considers groups of siblings locally, one may consider extending the scope of local rebalancing vertically. Brown [22, 21] presents B-slack trees in which the *slack* of each node is defined to be the number of child pointers (or elements, in the case of leaf nodes) missing. The special constraint imposed on B-slack trees is that, for each internal node, the total slack of its children is at most $B-1$ (note that $B > 4$ for these trees). For $n > B^3$, it is shown (among several other interesting properties) that the space complexity of a B-slack tree is bounded by $2nB/(B-3)$ which goes to $2n$ (optimal space complexity) as B increases and that average degree exceeds $B-2$. Update times are amortized logarithmic and some variants are additionally presented which trade slightly worse space complexity for amortized constant update costs.

1.2.4 Our contribution

In paper 1, we give lower and upper bounds for the rebalancing complexity as a function of ϵ. As described in the preceding section 1.2.2, we analyse a model of simple leaf-oriented B-trees[1] in which all elements are stored in the leaves, each leaf node contains at most B elements, and internal nodes have at most B children. With $n = (1-\epsilon)N$ where n is the current number of records stored in the tree and $N = B^h$ is the capacity of the tree at its current (optimal) height, we show that there exists an insertion which will require updating $\Omega\left(1/(\epsilon B)\right)$ nodes in order to rebalance the tree.

We also describe a layout and rebalancing algorithm which requires amortized

$$\mathcal{O}\left(\frac{1}{\epsilon B} \cdot \log^2\left(\min\{1/\epsilon, B\}\right)\right)$$

node updates per insertion. The workings of this algorithm rely upon the ratio of free space being within a constant factor of the ϵ the layout was initialized with. Simple rebuilding once this no longer holds is sufficient to maintain the amortized complexity for the new ϵ'.

The core component of the algorithm is a strategy to distribute and manage the free space left within the tree, bounding the distance that keys need to be shifted around by grouping nodes. This approach is inspired by the optimal rebalancing algorithm for binary trees found in [32]. Quick analysis shows that the amortized rebalancing cost achieved by this algorithm is asymptotically $\mathcal{O}(1/(\epsilon B))$—matching the lower bound—for all levels except the leaf level. For the leaf level, however, the amortized rebalancing cost is $\mathcal{O}(1/\epsilon)$; the same as is achieved for binary search trees and far from our lower bound for B-trees. To improve this, we describe how to apply density keeping techniques such as described in section 3 of [20] to better manage the

[1] The results we prove also apply to other reasonably similar models—such as B-trees where leaf nodes may contain up to cB elements for some constant c or node-oriented trees where elements are stored in all nodes and not just the leaves.

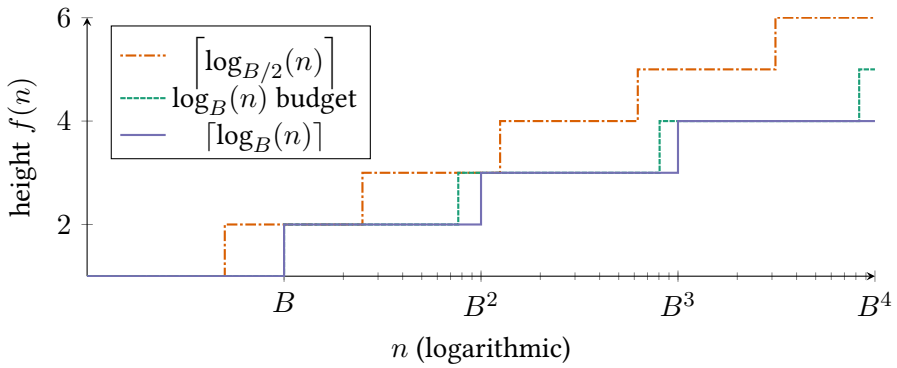


Figure 1.2: Plot (from fig. 2.2 in paper 1) illustrating different B-tree heights for growing n. The functions represent the worst-case height of standard B-trees, the height achieved by our new scheme using a rebalancing budget of $O(\log_B(n))$, and the optimal height bound for B-trees. For visualisation purposes, B is only 10 for this plot.

distribution of free space within the groups on the leaf level, bringing the amortized rebalancing cost down to the stated bound.

Using this algorithm, we additionally describe how to construct a scheme which, given a rebalancing time budget $f(n)$ dependent upon the tree size, will keep tree height optimal for n in the interval $B^{h-1} < n \leq N$ where the cost of doing so is within the budget and optimal plus one in the remaining interval $N' < n \leq N$.

Providing our scheme with a rebalancing budget of $\Theta(\log_B n)$—which matches the cost of searching in the B-tree—is sufficient for it to maintain optimal height for all but a vanishing fraction of the size intervals (between powers of B) for increasing n (illustrated in fig. 1.2). To the best of our knowledge, this is the first scheme achieving B-tree height within an additive constant of optimal.

1.2.5 Open problems

Given the results presented in our paper, an obvious open problem is to close the gap between the upper and lower bounds presented. It is not immediately obvious which of the two bounds (or indeed both bounds) could be improved.

Considering the upper bound, the $\log^2$ factor is a consequence of the density-keeping scheme applied to the groups of leaf nodes. Improving this would presumably require a significantly different approach: If we consider the isolated problem of shifting elements around within a leaf group as elements are inserted, this is essentially equivalent to the on-line list-labeling problem (also referred to as the file maintenance problem): each node in the group corresponds to a slot in an array of size $\Theta(1/\epsilon)$. For our scheme to work, the group needs to handle $\Theta(B)$ insertions before being rebuilt (see the paper for details). This, in the list labeling model, means that the array is filled to some constant fraction. A known lower bound [24] states that inserting n elements into an array of size Cn for $C > 1$ has a cost (counting number of times elements have to be moved) of $\Theta(\log^2 n)$.

Another open problem based on our results is deamortizing the upper bound. This would presumably require preemptive restructuring under our scheme and additional analysis. Fortunately, there already exists worst-case $O(\log^2 n)$ file maintenance algorithms [51, 16], suggesting that the density-keeping scheme for the leaves could be adapted more simply.

1.3 Graph algorithms for external memory

1.3.1 Overview

Graphs arise in modeling a large number of phenomena and a vast amount of research has gone into developing algorithms for analyzing graphs. Contemporary examples of large graphs include social graphs and communication networks. Handling large graphs in an external-memory setting, however, adds significant complexity to even simple graph problems.

In internal memory[2], several problems related to graph traversal are trivial in linear or almost linear time. Both breadth-first search and depth-first search take linear time and enable computation of interesting properties such as shortest (unweighted) paths or topological orderings in the case of directed acyclic graphs. Minimum spanning trees[3] can be computed in $\mathcal{O}(m\alpha(m, n))$ time deterministically [25], where α is the inverse Ackermann function and n and m denote the number of vertices and edges, respectively. Using the randomized algorithm proposed by Karger et al. [38], MSTs can be computed with expected time complexity $\mathcal{O}(m)$. It is a longstanding open problem whether there exists a deterministic $\mathcal{O}(m)$ time MST algorithm—though Pettie and Ramachandran have described an algorithm which they prove to be optimal [47], the exact upper bound for this algorithm is not known. Note however that simply finding *a* spanning tree is implicitly achieved in $\mathcal{O}(m)$ time using any graph traversal. Similarly, partitioning a graph into its connected components is trivial to do in time $O(m + n)$ while traversing it. Our work is focused on computing connected components of fully external graphs.

In external memory, simply traversing a graph efficiently is non-trivial. For the remainder of this section, V and E denote the vertices and edges of a graph. It is common in internal memory to represent graphs using adjacency lists. With this representation, the list of neighbors for any node can be accessed in $\mathcal{O}(1)$ time. If, in external memory, the graph is stored as an edge list plus an index enabling lookup of a neighbor list in $\mathcal{O}(1)$ I/Os, then following edges naively leads to random I/Os. Recalling that $|E| \gg \text{scan}(|E|)$, this is immensely inefficient.

For a closer look at the specific challenges in adapting algorithms for external memory, consider the ordinary internal-memory breadth-first search algorithm (shown in algorithm 1). For each edge (u, v) this algorithm processes, it needs to check whether v has already been visited. If the graph is fully external ($|V| > M$), keeping this information in memory for all $u \in V$ is not possible and this check may in and of itself cause an I/O, for a total of $|E|$ I/Os during the algorithm. As the order in which the adjacency lists of the nodes is not known beforehand, each visit to a new node starts with a random I/O, causing $|V|$ I/Os for the adjacency list

[2] We intentionally use the term "internal memory" somewhat loosely, serving only to discuss classical algorithms and how they fare in external memory. The term may be replaced with "the RAM model".

[3] For graphs with multiple components, the more general problem is that of computing minimum spanning forests. This does not, however, pose a problem for the algorithms we consider.

accesses (in addition to then scanning the adjacency lists which sums to $\mathcal{O}(\text{scan}(|V|))$).

Algorithm 1 Breadth-first traversal in internal memory

```
procedure BFS(G = (V, E), r)
    discovered[u] ← false for u ∈ V \ {r} and true for r
    let Q be a queue initially containing r
    while Q ≠ ∅
        u ← dequeue(Q)
        report u
        for v ∈ V adjacent to u
            if discovered[v] is false
                enqueue(Q, v)
                discovered[v] ← True
```

Specialised algorithms have been developed to reduce the I/O complexity below $\mathcal{O}(|V| + |E|)$. An algorithm by Munagala and Ranade [46] uses properties of BFS levels in undirected graphs to reduce the $|E|$ term, achieving an I/O complexity of $\mathcal{O}(|V| + \text{sort}(|E|))$. Adding preprocessing to first cluster the adjacency lists, Mehlhorn and Meyer [44] give an algorithm which additionally improves upon the $|V|$ term to achieve the first sublinear I/O complexity of

$$\mathcal{O}\left(\sqrt{\frac{|V|(|V|+|E|)}{B}} + \text{sort}(|V|+|E|)\right)$$

for BFS traversal.

Depth-first search (algorithm 2) is even more problematic as, when loaded, adjacency lists are not scanned—instead, the search continues recursively. Thus, even without considering the data structure keeping track of discovered nodes, naive depth-first search may incur $\Theta(|V| + |E|)$ I/Os. A simple adaptation by Chiang et al. [26] is to essentially keep the marking structure

Algorithm 2 Depth-first traversal in internal memory

```
procedure DFS((G = (V, E), s))
    marked[u] ← false for u ∈ V
    procedure RECURSE(u)
        marked[u] ← true
        report u
        for v ∈ adj[u]
            if marked[v] is false
                dfs(v)
    RECURSE(s)
```

in memory and whenever memory is full, scanning all edges and discarding ones which will not be needed before emptying the marking structure and continuing. This yields an I/O cost of $\mathcal{O}\left(1 + V/M\text{scan}(E) + V\right)$ (an improvement over $\mathcal{O}(V + E)$ if $E \in o(VB)$).

Buchsbaum et al. [23] propose an algorithm centered around buffered repository trees (a dictionary structure they introduce optimized for fast insertions) and priority queues to keep track of already visited nodes. It performs directed DFS (or directed BFS) with $\mathcal{O}(|V|)$ calls to extract-min and $\mathcal{O}(|E|)$ calls to insert on a number of priority queues and $\mathcal{O}(|V|)$ calls to extract and $\mathcal{O}(|E|)$ calls to insert on a BRT. Using their buffered repository trees based on $(2, 4)$-trees and an optimal priority queue (which does not need decrease-key; they refer to the buffer trees by Arge [11]), this yields an I/O cost of:

$$\mathcal{O}\left(\left(|V| + \frac{|E|}{B}\right)\log_2 \frac{|V|}{B} + \text{sort}(|E|)\right).$$

Using the recent external-memory priority queues of [36] instead, this algorithm has an I/O cost of:

$$\mathcal{O}\left(|V|\frac{M^{\frac{\alpha}{1+\alpha}}}{B}\log^2_{\frac{M}{B}}\frac{|E|}{B} + |V|\log_{\frac{M}{B}}\frac{|E|}{B} + \frac{|E|}{B}\log_{\frac{M}{B}}\frac{|E|}{B}\right),$$

which matches sort$(|E|)$ for fully external dense graphs with $|E| \in \Omega(|V|^{1+\varepsilon})$ for $\varepsilon > 0$.

It is worth noting that several improved results are available for restricted graph classes. Notably, for planar graphs, the separator algorithm of Maheswari and Zeh [42] combined with algorithms by Arge et al. [13, 12] enables BFS and DFS on planar graph in $\mathcal{O}(\text{sort}(|V|))$ I/Os.

Minimum spanning trees can be efficiently computed for semiexternal graphs ($V < M < E$) using Kruskal's algorithm; as a union-find structure for all nodes in V can in this case fit into memory, the I/O complexity is simply $\mathcal{O}(\text{sort}(|E|))$ to sort the edges before scanning them. As mentioned in [48], sorting E can be omitted by using dynamic trees, reducing the I/O cost to $\mathcal{O}(\text{scan}(|E|))$ at the cost of increased computational effort.

For fully external graphs, the best known deterministic algorithms are based on Borůvka's MST algorithm. In each step, Borůvka's algorithm contracts (super)nodes along their lightest incident edges. With common external-memory techniques, a Borůvka step can be performed in $\mathcal{O}(\text{sort}(|E|))$ I/Os. Arge et. al [12] show how to achieve I/O complexity

$$\mathcal{O}\left(\text{sort}(|E|)\log\log\left(\frac{|V|B}{|E|}\right)\right)$$

by carefully scheduling Borůvka steps in a number of super-phases with edge filtering in between.

In terms of randomized algorithms, the expected linear time algorithm for internal memory [38] (which we will refer to as KKT) can be translated into an expected $\mathcal{O}(\text{sort}(|E''|))$ I/O complexity algorithm [26, 2]. A simpler algorithm, designed to be efficient in practice, is presented by Sibeyn et al. in [48]; we will refer to this as the Sibeyn-Meyer algorithm. The main component of this algorithm is a simple node contraction approach which can be seen as an application of time-forward processing over the edges ordered by source and weight. Contracting nodes until the semi-external base case of Kruskal's algorithm is reached, the Sibeyn-Meyer algorithm requires $\mathcal{O}(\text{sort}(|E|)\log(|V|/M))$ I/Os in expectation.

Interestingly, current results do not provide a clear separation between the complexity of computing an MST and simply computing *a* spanning tree of a graph—or even decomposing

a graph into its connected components (which we will refer to as the CC problem). The best known algorithms for connected components are essentially the best known MST algorithms or asymptotically equivalent simplifications of them.

1.3.2 Our contribution

Considering the current best known results, our goal is to implement and evaluate a CC algorithm which is efficient in practice *and* has an expected I/O complexity of $\mathcal{O}(\text{sort}(|E))$. The KKT MST algorithm relies on a verification subroutine which would seem to introduce quite some overhead; though we have not found external-memory evaluations of this, an empirical survey [30] with experiments on somewhat smaller graphs concludes that the expected linear-time algorithm of Karger et al. is not feasible in practice.

If the goal is simply to compute connected components, however, the full verification routine is not needed: the verification step can be replaced with simple relabeling. Another way to describe this algorithm is that follows the structure of the functional CC algorithm from [1] with the addition of node contraction in the beginning of each recursive call. We describe, analyze, and test empirically several variants of this algorithm. In the ordinary version of the KKT algorithm, Borůvka steps are used to perform node contraction. We propose a simple randomized version of a Borůvka step, sacrificing some contraction power to make the execution of a simple step faster. Additionally, we suggest using node contraction approach from the Sibeyn-Meyer in place of the Borůvka steps: when used to reduce the number of nodes by a constant factor, the Sibeyn-Meyer algorithm uses only $\mathcal{O}(\text{sort}(|E|))$ I/Os with rather good constants.

The most obvious candidates to compare our results to include the external-memory version of Borůvka's algorithm simplified for CC computation and Sibeyn-Meyer's MST algorithm modified for CC computation. We find that the Sibeyn-Meyer algorithm is consistently more efficient for node contraction than both the ordinary and the randomized Borůvka algorithms we consider. However, our implementation of the KKT-based algorithm using Sibeyn-Meyer style node contraction is competitive with appropriate tuning parameters. For graphs which are not too sparse—the threshold depending on the graph class—it outperforms the Sibeyn-Meyer algorithm. On random graphs (Gilbert graphs), we see the optimized KKT-based algorithm winning already for graphs with $|E| \geq 5 \cdot |V|$.

1.3.3 Open problems

In existing literature, there seems to be a lack of empirical evaluation of external-memory MST algorithms. Our work includes an implementation of Borůvka's algorithm modified to compute connected components. We are, however, not aware of any implementations of the super-phase Borůvka MST algorithm by Arge et. al. Additionally, we are not aware of any implementations of the full KKT MST algorithm; in choosing to analyze the KKT-based CC algorithm, we assumed that the MST version would be impractical due to the constants involved. Considering [30], this may very well be the case, but this assumption should nonetheless be tested empirically in the external-memory setting.

When modifying the Sibeyn-Meyer algorithm to compute CCs rather than MSTs, a number

of simplifications can be made. One particular difference is that the CC version of the algorithm, when contracting a node, can freely choose which outgoing edge to contract along. This choice has significant impact on the time-forward process; whereas the MST version always contracts along the lightest incident edge (essentially a random choice, given that node IDs are first randomized), the CC version can choose to send information to the farthest neighbor with respect to the time-forward process. Based on experimental results, we conjecture that this makes the total number of edges processed asymptotically lower than the upper bound given for the MST algorithm.

1.4 Simulating population protocols

1.4.1 Preliminaries

Several models exist to model the behavior of systems consisting of multiple autonomous agents. These models can be used to reason about and predict the behavior of complex systems or aid in developing protocols for the systems to operate under. One such model is the *population model* introduced by Angluin et al. [9, 8].

The object of consideration in this model is a set of n agents, each an instance of a finite automaton. As a system, they compute a *population protocol* by interacting with each other in pairs. In each interaction, one or both agents may update their state according to a transition function.

The agents all follow the same state machine model though they may have different initial states. We denote the set of states by Q with $|Q| = m$. Originally presented as a way to model simple mobile sensors (for example, sensors attached to birds), an important characteristic of the model is that the agents have very simple behavior. This is reflected in the number of states; $m \ll n$ and in most relevant population models, $m \in \mathcal{O}(\operatorname{polylog} n)$. For some of our results, we will assume that $m < \sqrt{n}$.

The state of the entire system—that is, the state of all the agents at a given time—is referred to as the *configuration*. Though a population protocol does not terminate, it is meaningful to analyze when the system converges to a desired configuration. For instance, in a leader election protocol, the goal might be for the system to reach a (stable) configuration in which exactly one agent is in the leader state. In the case of majority protocols, agents start in one of two states and the goal is for all agents to arrive at a (correct) consensus on which state was initially in the majority.

The order in which different agents are chosen to interact is determined by a *scheduler*. In order to give convergence guarantees, one may analyze how a protocol performs under an adversarial scheduler (potentially restricted by some fairness conditions). One may also consider randomized schedulers—a natural choice for the setting of modeling simple mobile sensors being moved around, communicating when randomly meeting each other. In our work, we consider the standard probabilistic scheduler: for each interaction, it selects two agents uniformly at random two.

1.4.2 Previous work

Work has been done to analyze the computational model of population protocols, including characterizing its computational power [10] and considering extensions of the model [18].

For particular problems, research has been done to produce optimize protocols in terms of convergence time and number of states needed. An intensively studied example is the problem of *leader election*: all agents start in the same state and the goal is to reach a configuration in which exactly one agent is in a leader state and all others are in follower states. For this problem, several different properties have been proven. Note that the term *time* sometimes refers to *parallel time* in the literature—for consistency, we state the summarized results in terms of number of interactions.

Literature indicates that there is a trade-off between the number of states used by a protocol and the expected convergence time it can achieve. If protocols are restricted to a constant number of states, Doty and Soloveichik [28] showed that any stable leader election protocol requires $\Omega(n^2)$ interactions in expectation. Recently, Sudo and Masuzawa [49] showed that *any* population protocol needs $\Omega(n \log n)$ interactions, regardless of the number of states used. In terms of upper bounds, Alistarh and Gelashvili [6] gave a protocol which uses $\mathcal{O}(\log^3)$ states and converges in $\mathcal{O}(n \log^3 n)$ expected interactions. Gąsieniec et al. [33] improved both parts of the trade-off by providing a protocol converging in expected $\mathcal{O}(n \log^2 n)$ interactions while using only $\mathcal{O}(\log \log n)$ states. This number of states matches a lower bound [7] for any protocol converging in expected $o(n^2/\operatorname{polylog} n)$ interactions. Gąsieniec et al. [34] later improved the expected number of interactions to $\mathcal{O}(n \log n \log \log n)$ before Berenbrink et al. [17] recently closed the gap by providing a protocol using $\Theta(\log \log n)$ states and converging in $\mathcal{O}(n \log n)$ expected interactions (which is optimal for both states and time).

In the field of population protocols, simulation results can be useful in guiding theoretical analysis of the convergence behavior of different protocols. However, if hampered by computational effort, simulation may be of limited use. For instance, subtle differences in upper bounds may be impossible to infer from observed behaviors unless the simulations can reach massive scales. If simulation times restrict the feasible population sizes to $n \leq 2^{32}$, for example, then factors on the order of $\log \log n$ may be impossible to discern.

1.4.3 Our contributions

Our main contribution is an approach to more efficiently simulate population protocols with massive populations. We exploit two assumptions about the particular type of protocols we are interested in simulating: agents are indistinguishable and we only need to observe the concrete configuration infrequently rather than for every interaction, allowing us to process updates in batches. With these, we develop a batch processing approach which, under reasonable assumptions, simulates population protocols using $o(1)$ time per interaction, enabling simulation of massive populations.

Compact configuration representation In a population protocol, agents are indistinguishable in the sense that any agent is only defined by its current state. As such, the configuration at any time may be represented more compactly as a histogram over the states rather than as an

explicit enumeration of agents. This reduces the amount of memory required from $\Theta(n \log m)$ bits to $\Theta(m \log n)$ bits. For protocols where m is bounded polylogarithmically in n, this immediately makes it feasible to perform simulations in which n exceeds the amount of available memory.

We consider a number of data structures to represent the configuration as a histogram while allowing for efficiently sampling agents at random. Such data structures essentially simulate urns models: they support drawing random balls (corresponding to agents) from a set with or without replacement (with the transition function in our case defining the replacement policy). Among these, we present what we call *dynamic alias tables* which are an extension of the *alias method* [50]. There are other approaches capable of modeling a dynamic urn, but as these are more complex, we believe that, considering their efficiency and simplicity, our dynamic alias tables may be of independent interest.

Batch processing When empirically examining for instance the convergence behavior of a protocol, one typically wants to simulate a large number of interactions and it may not be relevant to inspect the global configuration after each individual interaction. If, for instance, the convergence time for a protocol is polynomial in n, it may suffice to perform $N > n$ interactions at a time. Our main contribution—leading to simulating protocols in sublinear time—is an efficient batch processing scheme capable of simulating a number $N > n$ of interactions as a batch operation.

The main idea of this scheme is to identify and process a *run* of *independent* interactions: a maximal sequence of agents drawn and grouped into pairs without any agent appearing more than once. With the interactions within a run being independent, the order in which updates happen does not matter, which means that we can update the states for all agents in the run at once.

The latter is rather simple: suppose we have an independent run and we know the frequency with which ordered pair of states occurs. Then, considering these frequencies and evaluating the transition function for each state pair in Q^2, we can determine how many agents will enter or exit each state. Having computed this, we update the frequencies stored for all states.

The efficiency of this approach depends on the length of the independent runs. If ℓ is the expected length of an independent run, then we need to simulate N/ℓ runs (in expectation) for a batch, each of which takes time $\mathcal{O}(m^2)$ plus some additional work (described later) which amounts to $\mathcal{O}(\log n)$ per run. In the paper, we show that $\ell \in \Theta(\sqrt{n})$ in expectation which means that our batched approach provides a speedup when $\sqrt{n} \in \omega(\max\{\log n, m^2\})$.

This process of course requires us to be able to sample an independent run in the first place and efficiently determine the state-pair frequencies. In the paper, we analyze in more detail the distribution that the independent run lengths follow. This allows us to use an inverse sampling technique to sample a run length at random from this distribution—this sampling process as implemented takes time $\mathcal{O}(\log n)$. Knowing the length of the run (the number of agents in total), we then sample from hypergeometric distributions based on the current frequencies of each state to arrive at the needed set of state pair frequencies. For correctness of the simulation, we sample and handle a collision: a repeat agent which appears immediately after the end of the independent run, by definition.

As sampling the m^2 interaction frequency matrix is the most computationally demanding part of our approach, we further develop a method to combine multiple batches into an epoch. Essentially, we sample the length of several independent runs, take care of collisions happening between them and only sample the full interaction matrix once per epoch. This takes $\mathcal{O}(\rho \log n + m^2)$ time for an epoch of ρ runs. Since the notion of collisions within an epoch refers to all agents sampled, later independent runs will in expectation get shorter, limiting efficient choices for ρ. We analyze the distribution of run lengths within an epoch to provide a balanced trade-off between the costs associated with each run (handling collisions, sampling the length) and batch processing (sampling the interaction matrix).

1.4.4 Open problems

We have so far considered a model in which agents are chosen uniformly at random to interact. It may be interesting to consider agents which are nodes in a network where edges define possible interactions (in which case our setting would correspond to a complete graph). Simulating models efficiently requires a different approach, however, as our approach specifically exploited the indistinguishability of agents with identical states. For large-scale simulations, external-memory graph techniques may be necessary. Using network-based model simulation to predict virus spread, for instance, would be quite relevant given current events.

1.5 Fragile complexity

1.5.1 Motivation

In our paper "Fragile Complexity of Comparison-Based Algorithms", we introduce the concept of fragile complexity. Ordinary algorithm analysis is focused on time complexity. In the case of comparison-based algorithms, one may asymptotically bound the total number of comparisons performed which typically will be proportional to the total amount of computation done. We propose to instead analyze how many times *individual* elements in the input set are subjected to comparison.

Inspiration for this model comes from scenarios in which comparing two elements may not only take some quantum of time—evaluating the comparison may also impact the elements being compared. For instance, the algorithmic problem at hand could be a boxing tournament and the elements to compare could be boxers. Each boxing match would result in an outcome proving one boxer to be superior to his or her opponent; a "comparison" between two elements. Boxing, however, is exhausting and each match brings a risk of debilitating injury. As such, a good boxing tournament structure should take care to avoid subjecting any individual boxer to an unreasonable number of fights. This example can be generalized to settings where consumables are sampled and compared directly. Perhaps one wishes to select the best wine out of a selection of wines by comparing them (avoiding assigning fixed ratings) two at a time. If the goal is instead to rank the wines, the problem of sorting naturally arises.

We define the *fragile complexity* of an algorithm to be the maximum over all input elements of the number of comparisons performed involving this particular element. One may additionally wish to protect a particular element. Perhaps the boxing tournament is a national

qualification event from which the winner will go on to the Olympics. In this case, it may make sense to lower the number of fights the winner will have to endure to improve their chances at the Olympics—even if this is at the expense of other unfortunate contestants. For some problems, we therefore also consider the fragile complexity of specific elements (such as the minimum or the median) rather than the maximum over all elements.

We compare results achieved in the fragile model to existing results for comparison or sorting networks. A comparison network gets its input on a number of "wires" and it performs computation by letting elements move along these wires, crossing through comparators: nodes at which elements on two wires are compared and potentially swapped. Such a network represents a data-oblivious algorithm in the sense that all the comparisons are given ahead of time. The *depth* of a network is the largest number of comparators any given element can pass through. When used as a model for parallel algorithms, the depth determines the parallel running time. In the fragile model, simulating a comparison network with depth d will directly result in an algorithm with fragile complexity d. The *size* of a network is the total number of comparators; when simulating the network sequentially, this translates into the running time.

1.5.2 Our contribution

To the best of our knowledge, the concept of fragile complexity has not been formally studied before. We formally define this new model of algorithmic complexity and provide bounds for a number of basic algorithmic problems for comparison-based algorithms, most notably minimum finding, selection (or median finding), and sorting.

In the case of finding the minimum element of a set (which is equivalent to boxing tournament example), we show that no deterministic algorithm can do better than regular tournament trees which have logarithmic height and therefore logarithmic fragile complexity. Interestingly, we prove that randomized approaches can provide different trade-offs between the expected fragile complexity for the minimum element and all other elements. For example, using surprisingly simple sampling, one can protect the winning element such that it in expectation only endures $\mathcal{O}(1)$ comparisons—at the expense of the runner-ups which go through $\mathcal{O}(n^\epsilon)$ comparisons. This is all the more significant as our lower bound for deterministic algorithms specifically targets the winning element, ruling out the possibility to "protect" it in a deterministic way. Our sampling approach allows for a number of different trade-offs between the expected number of comparisons involving the minimum element and all others. Additionally, we give an almost matching non-trivial lower bound on the fragile complexity of this trade-off.

For the problem of sorting, the ordinary lower bound on comparison-based sorting of $\Omega(n \log n)$ implies that the best fragile complexity one could hope for is $\Omega(\log n)$. This is achieved deterministically by simulating AKS sorting networks [4] which have logarithmic—although with (in)famously large constants—depth. The time complexity of such a simulation is $\mathcal{O}(n \log n)$. It remains an open problem to sort deterministically with $\mathcal{O}(\log(n))$ depth without using this construction.

The popular Quicksort algorithm is entirely infeasible in the fragile model; by design, the first pivot element will participate in $n - 1$ comparisons. Ordinary merge sort has fragile complexity $\Theta(n)$, though we show that this can be improved to $\mathcal{O}(\log^2 n)$ by using exponential search when merging. This is tight: by a simple adversary argument, we show that with *any*

merging strategy, binary merge sort will have fragile complexity $\Omega(\log^2 n)$. As with the case of minimum finding, randomization helps. Ordinary merge sort (with linear merging) will have $\mathcal{O}(\log n)$ fragile complexity with high probability if the input is first shuffled.

Using ideas from the AKS construction and careful sampling, we construct a selection[4] algorithm which performs selection in $\mathcal{O}(n)$ time complexity and $\mathcal{O}(\log n)$ fragile complexity. This shows a separation between the fragile model and comparison networks: using a lower bound for comparison networks which perform partition [5], we show that any selection network must have size $\Omega(n \log n)$. We also provide a randomized parameterized selection algorithm which offers different trade-off between the (expected) comparisons to the median element and all others (with expected linear time complexity). One interesting trade-off achieved by this is $\mathcal{O}(\log \log n)$ comparisons with the median element and $\mathcal{O}(\sqrt{n})$ with all others.

1.5.3 Open problems

As mentioned in the previous section, we refer to AKS networks for deterministically sorting with optimal fragile complexity $\mathcal{O}(\log(n))$. Such networks have notoriously large constants hidden by the asymptotic notation and are as such not considered to have any practical applications. A natural question is therefore whether there exists a "simple" sorting algorithm with fragile complexity $\mathcal{O}(\log(n))$ that does not rely on such networks.

As we use part of the machinery from AKS networks in our deterministic selection algorithm (which does, however, have better time complexity than simulating a full AKS sorting network), one could also ask whether this could be replaced with something "simpler". Note that by a simple padding argument, the lower bound of $\Omega(\log n)$ on the fragile complexity of deterministic minimum finding carries over to the selection problem.

Looking at the randomized results, the bounds for randomized minimum finding are not tight—an obvious open problem is to close this gap. The randomized median trade-offs we present are not quite good as the ones for minimum finding; it does not enable expected $\mathcal{O}(1)$ comparisons with the median element. Improving this or providing a matching (or indeed any) lower bound for the trade-off is still an open problem.

On the level of comparing models of computation, there is still an open question about the relationship between fragile complexity and depth of comparison networks. We do show that there are algorithms with fragile complexity matching the best depth of comparison networks while having lower time complexity than the corresponding network size. We do not, however, show any algorithms which have fragile complexity strictly lower than the best depth of comparison networks for the same problems. An interesting open problem is therefore to either identify a problem and an algorithm where this is the case or to provide a proof that maps algorithm with a given fragile complexity to comparison networks.

Taking a step back, the scope of our paper is limited to basic problems for comparison-based algorithms, motivated by the notion that a comparison can impact the elements being compared negatively. Perhaps further avenues of research could be opened by extending the concept of fragile complexity to other appropriate contexts.

[4] The algorithm finds the median element, but this can be used for arbitrary selection (finding the element of a desired rank) by a simple padding argument.

A different direction to extend the model is to add more flexibility to prioritizing certain elements. We give a number of results on trade-offs in terms of the comparisons impacting "the result" such as the minimum element in minimum finding or the median in median finding. This could be generalized, for example, to a model in which each element is assigned a weight and the fragile complexity is the maximum over the comparisons to each element scaled by its weight.

Bibliography

[1] Abello, Buchsbaum, and Westbrook. "A Functional Approach to External Graph Algorithms". In: *Algorithmica* 32.3 (Mar. 2002), pp. 437–458. ISSN: 0178-4617, 1432-0541. DOI: 10.1007/s00453-001-0088-5.

[2] James Abello, Adam L. Buchsbaum, and Jeffery R. Westbrook. "A Functional Approach to External Graph Algorithms". In: *European Symposium on Algorithms*. Springer, 1998, pp. 332–343.

[3] Alok Aggarwal and Jeffrey Vitter S. "The Input/Output Complexity of Sorting and Related Problems". In: *Communications of the ACM* 31.9 (Aug. 1, 1988), pp. 1116–1127. ISSN: 00010782. DOI: 10.1145/48529.48535.

[4] M. Ajtai, J. Komlós, and E. Szemerédi. "Sorting in $c \log n$ parallel steps". In: *Combinatorica* 3.1 (Mar. 1983), pp. 1–19. ISSN: 1439-6912. URL: https://doi.org/10.1007/BF02579338.

[5] V. E. Alekseev. "Sorting Algorithms with minimum memory". In: *Kibernetika* 5.5 (1969), pp. 99–103.

[6] Dan Alistarh and Rati Gelashvili. "Polylogarithmic-Time Leader Election in Population Protocols". In: *ICALP (2)*. Vol. 9135. LNCS. Springer, 2015, pp. 479–491. DOI: 10.1007/978-3-662-47666-6_38.

[7] Dan Alistarh et al. "Time-Space Trade-offs in Population Protocols". In: *SODA*. SIAM, 2017, pp. 2560–2579. DOI: 10.1137/1.9781611974782.169.

[8] Dana Angluin et al. "Computation in Networks of Passively Mobile Finite-State Sensors". In: *Distributed Comput.* 18.4 (2006), pp. 235–253. DOI: 10.1007/s00446-005-0138-3.

[9] Dana Angluin et al. "Computation in networks of passively mobile finite-state sensors". In: *PODC*. ACM, 2004, pp. 290–299. DOI: 10.1145/1011767.1011810.

[10] Dana Angluin et al. "The Computational Power of Population Protocols". In: *Distributed Comput.* 20.4 (2007), pp. 279–304. DOI: 10.1007/s00446-007-0040-2.

[11] Lars Arge. "The Buffer Tree: A New Technique for Optimal I/O-Algorithms". In: *Algorithms and Data Structures*. Springer Berlin Heidelberg, 1995, pp. 334–345. ISBN: 978-3-540-44747-4.

[12] Lars Arge, Gerth Stølting Brodal, and Laura Toma. "On External-Memory MST, SSSP, and Multi-Way Planar Graph Separation". In: *Algorithm Theory - SWAT 2000*. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 1851. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 433–447. ISBN: 978-3-540-67690-4 978-3-540-44985-0. DOI: 10.1007/3-540-44985-X_37.

[13] Lars Arge et al. “On External-Memory Planar Depth First Search”. In: *Algorithms and Data Structures*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2001, pp. 471–482. ISBN: 978-3-540-44634-7. DOI: 10.1007/3-540-44634-6_43.

[14] R. Bayer and E. McCreight. “Organization and Maintenance of Large Ordered Indices”. In: *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. SIGFIDET ’70. New York, NY, USA: Association for Computing Machinery, 1970, pp. 107–141. ISBN: 978-1-4503-7941-0. DOI: 10.1145/1734663.1734671.

[15] R. Bayer and E. M. McCreight. “Organization and Maintenance of Large Ordered Indexes”. In: *Acta Informatica* 1.3 (Sept. 1, 1972), pp. 173–189. ISSN: 1432-0525. DOI: 10.1007/BF00288683. URL: https://doi.org/10.1007/BF00288683.

[16] Michael A. Bender et al. “Two Simplified Algorithms for Maintaining Order in a List”. In: *Algorithms — ESA 2002*. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 2461. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 152–164. ISBN: 978-3-540-44180-9 978-3-540-45749-7. DOI: 10.1007/3-540-45749-6_17.

[17] Petra Berenbrink, George Giakkoupis, and Peter Kling. “Optimal Time and Space Leader Election in Population Protocols”. In: *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*. Chicago, IL, USA, 2020. ISBN: 978-1-4503-6979-4. DOI: 10.1145/3357713.3384312.

[18] Michael Blondin, Javier Esparza, and Stefan Jaax. “Expressive Power of Broadcast Consensus Protocols”. In: *CONCUR*. 2019.

[19] Gerth Stølting Brodal and Rolf Fagerberg. “Lower Bounds for External Memory Dictionaries”. In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA ’03. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003, pp. 546–554. ISBN: 0-89871-538-5. URL: http://dl.acm.org/citation.cfm?id=644108.644201.

[20] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. “Cache Oblivious Search Trees via Binary Trees of Small Height”. In: *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA ’02. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002, pp. 39–48. ISBN: 0-89871-513-X. URL: http://dl.acm.org/citation.cfm?id=545381.545386.

[21] Trevor Brown. “B-Slack Trees: Highly Space Efficient B-Trees”. In: (Dec. 13, 2017). arXiv: 1712.05020 [cs].

[22] Trevor Brown. “B-Slack Trees: Space Efficient B-Trees”. In: *Algorithm Theory – SWAT 2014*. Red. by David Hutchison et al. Vol. 8503. Cham: Springer International Publishing, 2014, pp. 122–133. ISBN: 978-3-319-08404-6. DOI: 10.1007/978-3-319-08404-6_11.

[23] Adam Buchsbaum et al. “On External Memory Graph Traversal”. In: *SODA ’00: Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms* (Nov. 2000). DOI: 10.1145/338219.338650.

[24] Jan Bulánek, Michal Koucký, and Michael Saks. "Tight Lower Bounds for the Online Labeling Problem". In: *SIAM Journal on Computing* 44.6 (Jan. 2015), pp. 1765–1797. ISSN: 0097-5397, 1095-7111. DOI: 10.1137/130907653. URL: http://epubs.siam.org/doi/10.1137/130907653.

[25] Bernard Chazelle. "A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity". In: *Journal of the ACM* 47.6 (Nov. 1, 2000), pp. 1028–1047. ISSN: 00045411. DOI: 10.1145/355541.355562. URL: http://portal.acm.org/citation.cfm?doid=355541.355562.

[26] Yi-Jen Chiang et al. "External-Memory Graph Algorithms". In: *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '95. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1995, pp. 139–149. ISBN: 0-89871-349-8. URL: http://dl.acm.org/citation.cfm?id=313651.313681.

[27] Douglas Comer. "Ubiquitous B-Tree". In: *ACM Comput. Surv.* 11.2 (June 1979), pp. 121–137. ISSN: 0360-0300. DOI: 10.1145/356770.356776. URL: http://doi.acm.org/10.1145/356770.356776.

[28] David Doty and David Soloveichik. "Stable Leader Election in Population Protocols Requires Linear Time". In: *DISC*. Vol. 9363. LNCS. Springer, 2015, pp. 602–616. DOI: 10.1007/978-3-662-48653-5_40.

[29] Kasper Eenberg, Kasper Green Larsen, and Huacheng Yu. "DecreaseKeys Are Expensive for External Memory Priority Queues". In: *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*. Montreal, Canada: ACM Press, 2017, pp. 1081–1093. ISBN: 978-1-4503-4528-6. DOI: 10.1145/3055399.3055437.

[30] Cüneyt F. Bazlamaçcı and Khalil S. Hindi. "Minimum-Weight Spanning Tree Algorithms A Survey and Empirical Study". In: *Computers & Operations Research* 28.8 (July 1, 2001), pp. 767–785. ISSN: 0305-0548. DOI: 10.1016/S0305-0548(00)00007-1.

[31] Rolf Fagerberg. "Binary Search Trees: How Low Can You Go?" In: *Algorithm Theory — SWAT'96*. Red. by G. Goos, J. Hartmanis, and J. Leeuwen. Vol. 1097. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 428–439. ISBN: 978-3-540-61422-7. DOI: 10.1007/3-540-61422-2_151.

[32] Rolf Fagerberg. "The Complexity of Rebalancing a Binary Search Tree". In: *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 1999, pp. 72–83.

[33] Leszek Gasieniec and Grzegorz Stachowiak. "Fast Space Optimal Leader Election in Population Protocols". In: *SODA*. SIAM, 2018. DOI: 10.1137/1.9781611975031.169.

[34] Leszek Gasieniec, Grzegorz Stachowiak, and Przemyslaw Uznanski. "Almost Logarithmic-Time Space Optimal Leader Election in Population Protocols". In: *SPAA*. ACM, 2019, pp. 93–102. DOI: 10.1145/3323165.3323178.

[35] Scott Huddleston and Kurt Mehlhorn. "Robust Balancing in B-Trees". In: *Theoretical Computer Science*. Springer, 1981, pp. 234–244.

[36] John Iacono, Riko Jacob, and Konstantinos Tsakalidis. "External Memory Priority Queues with Decrease-Key and Applications to Graph Algorithms". In: *27th Annual European Symposium on Algorithms (ESA 2019)*. Vol. 144. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 60:1–60:14. ISBN: 978-3-95977-124-5. DOI: 10.4230/LIPIcs.ESA.2019.60.

[37] William Jannen et al. "BetrFS: A Right-Optimized Write-Optimized File System". In: *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX Association, Feb. 2015, pp. 301–315. ISBN: 978-1-931971-20-1. URL: https://www.usenix.org/conference/fast15/technical-sessions/presentation/jannen.

[38] David R. Karger, Philip N. Klein, and Robert E. Tarjan. "A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees". In: *Journal of the ACM* 42.2 (Mar. 1, 1995), pp. 321–328. ISSN: 00045411. DOI: 10.1145/201019.201022.

[39] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN: 0-201-89685-0.

[40] V. Kumar and E.J. Schwabe. "Improved Algorithms and Data Structures for Solving Graph Problems in External Memory". In: *Proceedings of SPDP '96: 8th IEEE Symposium on Parallel and Distributed Processing*. SPDP '96: 8th IEEE Symposium on Parallel and Distributed Processing. New Orleans, LA, USA: IEEE Comput. Soc. Press, 1996, pp. 169–176. ISBN: 978-0-8186-7683-3. DOI: 10.1109/SPDP.1996.570330. URL: http://ieeexplore.ieee.org/document/570330/.

[41] Klaus Küspert. "Storage Utilization in B *-Trees with a Generalized Overflow Technique". In: *Acta Informatica* 19.1 (Apr. 1983). ISSN: 0001-5903, 1432-0525. DOI: 10.1007/BF00263927. URL: http://link.springer.com/10.1007/BF00263927.

[42] Anil Maheshwari and Norbert Zeh. "I/O-Optimal Algorithms for Planar Graphs Using Separators". In: *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '02. San Francisco, California: Society for Industrial and Applied Mathematics, Jan. 6, 2002, pp. 372–381. ISBN: 978-0-89871-513-2.

[43] David Maier and Sharon C. Salveter. "Hysterical B-Trees". In: *Information Processing Letters* 12.4 (Aug. 1981), pp. 199–202. ISSN: 00200190. DOI: 10.1016/0020-0190(81)90101-0. URL: http://linkinghub.elsevier.com/retrieve/pii/0020019081901010.

[44] Kurt Mehlhorn and Ulrich Meyer. "External-Memory Breadth-First Search with Sublinear I/O". In: *Algorithms — ESA 2002*. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 2461. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 723–735. ISBN: 978-3-540-44180-9 978-3-540-45749-7. DOI: 10.1007/3-540-45749-6_63.

[45] *Algorithms for Memory Hierarchies: Advanced Lectures*. Lecture Notes in Computer Science 2625. Berlin ; New York: Springer, 2003. ISBN: 978-3-540-00883-5.

[46] Kameshwar Munagala and Abhiram Ranade. "I/O-Complexity of Graph Algorithms". In: *SODA*. Vol. 99. 1999, pp. 687–694.

[47] Seth Pettie and Vijaya Ramachandran. "An Optimal Minimum Spanning Tree Algorithm". In: *Journal of the ACM (JACM)* 49.1 (2002), pp. 16–34.

[48] Jop Sibeyn et al. “Engineering an External Memory Minimum Spanning Tree Algorithm”. In: *Exploring New Frontiers of Theoretical Informatics*. Vol. 155. Boston: Kluwer Academic Publishers, 2004, pp. 195–208. ISBN: 978-1-4020-8140-8. DOI: 10.1007/1-4020-8141-3_17.

[49] Yuichi Sudo and Toshimitsu Masuzawa. “Leader Election Requires Logarithmic Time in Population Protocols”. In: (Nov. 2, 2019). arXiv: 1906.11121 [cs]. URL: http://arxiv.org/abs/1906.11121.

[50] Alastair J. Walker. “An Efficient Method for Generating Discrete Random Variables with General Distributions”. In: *ACM Math. Soft.* 3.3 (1977). DOI: 10.1145/355744.355749.

[51] Dan E. Willard. “A Density Control Algorithm for Doing Insertions and Deletions in a Sequentially Ordered File in a Good Worst-Case Time”. In: *Information and Computation* 97.2 (Apr. 1992), pp. 150–204. ISSN: 08905401. DOI: 10.1016/0890-5401(92)90034-D.

2 On Optimal Balance in B-trees: What Does it Cost to Stay in Perfect Shape?

Rolf Fagerberg*, David Hammer†, Ulrich Meyer‡

Abstract

Any B-tree has height at least $\lceil \log_B(n) \rceil$. Static B-trees achieving this height are easy to build. In the dynamic case, however, standard B-tree rebalancing algorithms only maintain a height within a constant factor of this optimum. We investigate exactly how close to $\lceil \log_B(n) \rceil$ the height of dynamic B-trees can be maintained as a function of the rebalancing cost. In this paper, we prove a lower bound on the cost of maintaining optimal height $\lceil \log_B(n) \rceil$, which shows that this cost must increase from $\Omega(1/B)$ to $\Omega(n/B)$ rebalancing per update as n grows from one power of B to the next. We also provide an almost matching upper bound, demonstrating this lower bound to be essentially tight. We then give a variant upper bound which can maintain near-optimal height at low cost. As two special cases, we can maintain optimal height for all but a vanishing fraction of values of n using $\Theta(\log_B(n))$ amortized rebalancing cost per update and we can maintain a height of optimal plus one using $O(1/B)$ amortized rebalancing cost per update. More generally, for any rebalancing budget, we can maintain (as n grows from one power of B to the next) optimal height essentially up to the point where the lower bound requires the budget to be exceeded, after which optimal height plus one is maintained. Finally, we prove that this balancing scheme gives B-trees with very good storage utilization.

2.1 Introduction

2.1.1 Motivation

B-trees are search trees particularly suited for data organization in external memory. They are widely employed in database systems and file systems [12] and since their introduction in 1972 by Bayer and McCreight [7], they have been the subject of much theoretical and practical study.

*University of Southern Denmark, rolf@imada.sdu.dk, Supported by the Independent Research Fund Denmark, Natural Sciences, grant DFF-7014-00041

†Goethe University Frankfurt and University of Southern Denmark, hammer@imada.sdu.dk, Supported by the Deutsche Forschungsgemeinschaft (DFG) under grants ME 2088/3-2 and ME 2088/4-2

‡Goethe University Frankfurt, umeyer@ae.cs.uni-frankfurt.de, Supported by the Deutsche Forschungsgemeinschaft (DFG) under grants ME 2088/3-2 and ME 2088/4-2

A standard B-tree of order B is a search tree where all leaves are on the same level and every internal node has between $\lceil B/2 \rceil$ and B children except for the root which may have any number of children between 2 and B. One common generalization is (a, b)-trees [16] where internal nodes have between a and b children for $2 \leq a \leq \lceil b/2 \rceil$. The standard operations for rebalancing B-trees are *split* of an overfull node into two nodes (increasing the degree of the parent by one), *merge* which is the reverse of split, and *share* which is a redistribution of keys among neighboring nodes.

In search trees, the cost of an operation is measured as the number of nodes accessed. For a search operation, this cost is determined by the height of the tree. For any search tree with maximal fanout B, $\lceil \log_B(n) \rceil$ is a lower bound on its height. In the static case, this height is easily achieved by a bottom-up linear (that is, $O(n/B)$ assuming presorted keys) cost construction method. In the dynamic case, standard B-trees maintain an upper bound of $\lfloor 1+\log_{\lceil B/2 \rceil}(n/2) \rfloor$ on their height using $O(\log_B(n))$ rebalancing cost per insertion and deletion. Using (a, b)-trees, the amortized rebalancing cost per update can be reduced to $O(1)$ for $a = \lfloor b/2 \rfloor$ and to $O(1/b)$ for $a = b/4$, at the price of a moderate increase in the height bound [16, 22].

The upper bound of $\lfloor 1 + \log_{\lceil B/2 \rceil}(n/2) \rfloor$ on the height is a constant factor of approximately $\log B/(\log B - 1)$ away from the lower bound. The optimal bound $\lceil \log_B(n) \rceil$ can of course be achieved by spending linear cost on a rebuilding after each update, but this cost is prohibitive in most situations and begs the question: can optimal height be maintained more cheaply?

Intuitively, maintaining optimal height should become harder as n approaches the next power of B, since the amount of available space in the tree decreases, giving less flexibility during rebalancing. Or put differently, the number of different trees of optimal height shrinks as n increases—when reaching a power of B, there is only one such tree. Overall, we can expect that there must be a trade-off between how full the tree is and how costly rebalancing to optimal height will be. The goal we pursue in this paper is to find this trade-off. More generally, we would like to know which height bounds can be maintained at which costs—a correlation which may be called the intrinsic rebalancing cost of B-trees.

2.1.2 Our contributions

For any n, let N denote the next power of B (i.e., $N = B^{\lceil \log_B n \rceil}$) and define ϵ by $n = N(1-\epsilon)$. Our first main result is a lower bound showing that for any B-tree of optimal height, there exists an insertion forcing $\Omega(1/(\epsilon B))$ nodes to be rebalanced before the tree can again have optimal height. This expression describes how the rebalancing cost must change from $\Omega(1/B)$ to $\Omega(n/B)$ as n approaches the next power of B. See Figure 2.1 for a visualization of this bound.

Our second main result is an almost matching upper bound, which maintains optimal height using amortized $O(\log^2 B/(\epsilon B))$ rebalancing per update, thereby proving the lower bound essentially tight.

This lower bound (and hence the upper bound) approaches linear cost as n approaches the next power of two. As our third main result, we give a variant rebalancing scheme that allows almost optimal height at much lower amortized cost. More precisely, for any rebalancing budget $f(n)$, we can maintain optimal height $\lceil \log_B(n) \rceil$ as n approaches the next power of B essentially up to the point where the lower bound result requires the budget to be exceeded,

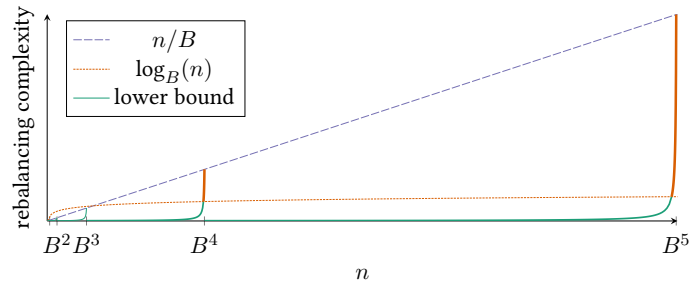


Figure 2.1: Plot illustrating how the lower bound for rebalancing cost increases as n approaches different powers of B. Intervals where the cost exceeds $\log_B(n)$ are highlighted in orange.

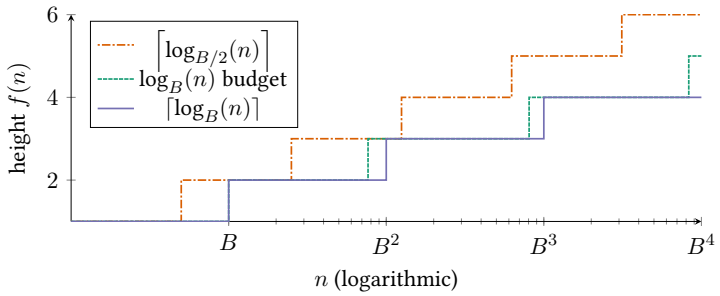
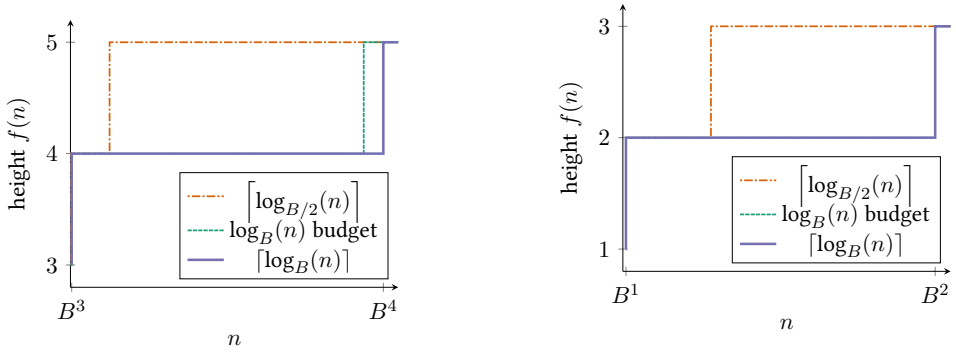


Figure 2.2: Plot showing different B-tree heights as a function of n. The functions represent the height of standard B-trees, the height achieved by our new scheme using a rebalancing budget of $O(\log_B(n))$, and the optimal height bound. For this particular plot, $B = 10$.

after which height $\lceil \log_B(n) \rceil + 1$ is maintained. To our knowledge, this is the first rebalancing scheme for B-trees with a height bound whose difference compared to optimal is an additive constant rather than a multiplicative constant.

One natural choice of budget is $f(n) = \log_B(n)$ as this matches the search cost. fig. 2.2 illustrates the height bound of this new scheme, the height bound of standard B-trees, and the optimal height bound. As is hinted by the figure, the fraction of values of n for which this scheme does not maintain optimal height $\lceil \log_B(n) \rceil$ is actually vanishing for growing n. Since in real life uses of B-trees in external memory the values of B are large and realistic values of n are fairly bounded in terms of possible powers of B, one may also want to look at the concrete improvements in height bounds for some practically occurring values of B and n. In database systems and file systems there are two main regimes, often termed OLTP and OLAP in the database setting. In the former, $B = 256$ is a typical value, in the latter, $B = 10^6$ is a typical value. For $B = 256$, the expression $B^3 < M \ll n \lesssim B^4$ (where M denotes the number of keys that can be held in internal memory) describes many real situations. fig. 2.3a repeats the plot of fig. 2.2 for the interval $B^3 < n < B^4$, but with $B = 256$ and the horizontal axis linear (not logarithmic). Standard B-trees and our scheme achieve the optimal height bound 4 for some part of the interval, but that part differs significantly in size: 12.16% versus 93.71%. For



(a) $B = 256$. Standard B-trees are optimal in 12.16% of the interval, our scheme is in 93.71% of the interval.

(b) $B = 10^6$. Standard B-trees are optimal in 27.49% of the interval, our scheme is in 99.98% of the interval. Note: two of the plots are essentially coincident.

Figure 2.3: Plots showing different B-tree heights as a function of n similar to fig. 2.2 but for two specific, realistic settings. The horizontal axes are non-logarithmic to more faithfully show proportions.

$B = 10^6$, the expression $B^1 < M \ll n \lesssim B^2$ describes many real situations. As illustrated by fig. 2.3b, the part of the interval $B^1 < n < B^2$ where optimal height is guaranteed is here 27.49% and 99.98% for the two schemes.

Another natural choice of budget is $f(n) = O(1/B)$ as this is the best possible amortized rebalancing cost (we can always make one node overflow at least every B insertions). With this budget, our scheme maintains height at most $\lceil \log_B(n) \rceil + 1$ for all values of n.

Finally, we prove that our near-optimal balancing scheme gives B-trees with very good storage utilization. Storage utilization is the fraction of the total space in the nodes allocated which is occupied by tree pointers, keys, and elements, i.e., the average use of the space in a node. High storage utilization is a desirable quality since it allows more of the tree to be cached in main memory and it has been the topic of much previous literature on B-trees, in particular in the database setting.

2.1.3 Previous work

Not much work has been done with an explicit focus on the height of B-trees. The concept of storage utilization is rather closely related, since optimizing the height requires increasing the average fanout of nodes, which again increases storage utilization. A number of previous results present different trade-offs between update complexity and storage utilization, and we now survey these. Standard B-trees have a worst-case storage utilization of approximately $1/2$ and a logarithmic rebalancing cost. It is well known that lowering the worst-case storage utilization of B-trees slightly by using (a, b)-trees with $a < \lceil b/2 \rceil$ allows for amortized constant rebalancing costs [22, 16]. The average storage utilization may be somewhat better than the worst-case. In [26], an analysis of 2–3 trees with random insertions suggests that such trees tend towards an expected storage utilization of $\ln 2 \approx 0.69$.

To improve the storage utilization in dynamic B-trees, Bayer and McCreight [7] proposed

an *overflow* technique: full nodes share their load with their siblings (when possible) instead of splitting (Knuth [18] refers to this variant as B*-trees). This improves the storage utilization in the worst case from $1/2$ to $2/3$, and the height bound from $\lfloor 1 + \log_{\lceil B/2 \rceil}(n/2) \rfloor$ to $\lfloor 1 + \log_{\lceil 2B/3 \rceil}(n/2) \rfloor$, at the price of increasing the update cost by a constant factor. As suggested in [7], this approach can be generalized to redistribution in bigger groups of nodes before splitting, thus further improving the storage utilization and the height bound at the cost of even higher rebalancing costs. However, no matter the group size, the height bound will be some constant factor away from optimal. The average storage utilization in a randomized setting using this approach is analyzed in [19], indicating that storage utilization converges to $m \ln((g+1)/g)$ for random trees under insertions where g is the overflow group size.

Similarly, an overflow scheme is tested in [25] which attaches overflow nodes to groups of nodes to delay splitting. They compute the average storage utilization to be $2g/(2g+3)$ when each overflow node is shared by a group of up to g leaf nodes. Both theoretical analysis and practical simulations show improved storage utilization at a cost of increased complexity during updates.

Rosenberg and Snyder [24] introduce so-called *compact B-trees* which are node-oriented trees shown to be close to optimal with respect to access costs while requiring minimal space. However, it is essentially a static structure as compaction incurs linear cost and no faster compactness-preserving update algorithm is known. The empirical analysis in a dynamic setting presented in [6] indeed shows that insertion into compact B-trees is very costly and that, without compaction, storage utilization rapidly degrades when the number of updates grows.

Recently, Brown [11, 10] has developed a practically motivated variation called *B-slack trees* which has amortized logarithmic update complexity while achieving high storage utilization. *Slack* for a node refers to the difference between its maximum degree and actual degree (number of children "missing"). Special to B-slack trees is that the children of any internal node have a combined slack of at most $B-1$ where $B > 4$ is the maximum degree of nodes. Maintaining this property is shown to ensure that a tree with n keys occupies at most $\frac{2B}{B-3.4}n$ words (which approaches $2n$—the optimal value given the model used—as B increases). The rebalancing cost is amortized logarithmic per update.

It is worth mentioning key compression techniques like those found in prefix B-trees [8] as an approach to improve fanout and thus potentially lower tree height. This and similar key compression techniques represent an orthogonal line of research which we do not pursue.

In contrast to B-trees, there for binary trees is a much stronger tradition for focusing on the height. In particular, there is a body of work [23, 5, 2, 1, 4, 3, 21, 20, 14, 15] focusing on rebalancing schemes with height bounds close to the optimal value $\lceil \log_2(n) \rceil$. The end result of this line of investigation is the matching upper and lower bounds in [14, 15]. Our results in this paper can be seen as a generalization of those methods and results to the case of B-trees—setting $B = 2$ in our bounds gives those of [14, 15]. One core new ingredient is the methods of Section 2.4.4 which are necessary for the upper bound to almost match the loss of a factor of B in the lower bound in Section 2.3 compared to the bound in [15].

2.2 Model

We describe our results for leaf-oriented B-trees, as these are standard in the literature. In leaf-oriented B-trees, internal nodes contain a total of $2B - 1$ fields: B pointers to subtrees and $B - 1$ search keys to guide the search (consistent with B-trees of order B as defined by Knuth [18]). Both keys and pointers can be nil. Since keys separate subtrees, the number of non-nil pointers is equal to the number of non-nil keys plus one. Leaf nodes contain the actual elements, including their search keys. For simplicity, we assume that leaf nodes contain up to B elements.[1] All leaf nodes appear at the same level.

A B-tree of height h can contain up to $N = B^h$ elements. We let T denote a B-tree with n elements and optimal height $h = \lceil \log_B n \rceil$ and we define ϵ by $n = N(1 - \epsilon) = B^h(1 - \epsilon)$. Since the height is optimal, we have $B^{h-1} < n \leq B^h$ and hence $0 \leq \epsilon < (B-1)/B$. A node is said to be modified by an update operation if any of its fields are changed. Clearly the number of modified nodes is a lower bound on the rebalancing cost of an update operation. Unlike in standard B-trees, we do not impose a lower bound on the number of keys. This only makes our lower bound stronger and it thus applies to standard B-trees. Our upper bounds can easily be adapted to conform to a lower bound on node contents.

2.3 Lower bound

In this section, we prove the following main theorem.

Theorem 1. *For any B-tree T of optimal height $h = \lceil \log_B n \rceil$, there exists an insertion into T such that rebalancing T to optimal height after the insertion will require modifying $\Omega(1/(\epsilon B))$ nodes, where ϵ is given by $n = B^h(1 - \epsilon)$.*

The proof is based on creating a mapping from B-trees into arrays (by suitably generalizing a mapping for binary trees in [14]). The mapping allows us to exploit the existence in any array of a "uniformly dense" position, which will point to an update position in T for which we can prove the lower bound stated in theorem 1.

2.3.1 Mapping into array

We create the mapping from elements of T to entries of an array $\psi(T)$ of length B^h as follows. For a full tree ($n = B^h$), this mapping is given by an inorder traversal which maps elements met during the traversal to increasing array entries. For a non-full tree, we embed T into a full tree of the same height before applying the mapping. We use the following specific embedding: keys fill up nodes from the left such that missing keys (and for internal nodes the corresponding missing subtrees) for non-full nodes will be on the right. This embedding and the resulting mapping is illustrated in fig. 2.4.

The mapping has the property that any subtree of the full tree is mapped to a contiguous interval in $\psi(T)$. In particular, for the full tree, a subtree with root at height h' spans an interval

[1] In practice, the size of elements relative to the size of keys may vary between data sets (and leaves may also store pointers to elements instead of elements themselves), but it is straight-forward to adapt our statements accordingly.

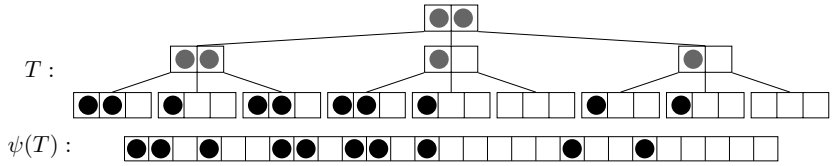


Figure 2.4: Illustration of the mapping of a tree T (with $B = 3$) into $\psi(T)$. Grey circles represent keys in internal nodes, black circles represent elements (in leaves).

of size $B^{h'}$ in $\psi(T)$. This implies that if such a subtree of T is moved during a rebalancing, (say, if siblings of the parent of the subtree are added or removed in T), the positions in $\psi(T)$ of its nodes will be shifted by a multiple of $B^{h'}$.

We denote intervals of array indices (that is, intervals of integers) as $[a; b]$ and the length $b - a + 1$ of such intervals as $l([a; b])$. The following lemma regarding density of subsets of intervals is from Dietz et al. [13].

Lemma 1. *For any two indices a and b and any $S \subseteq [a; b]$, there is an index $i \in [a; b]$ such that for any indices s and t where $a \leq s \leq i \leq t \leq b$, the following holds:*

$$\frac{|S \cap [s; t]|}{l([s; t])} \leq 2\frac{|S|}{l([a; b])}.$$

Consider the subset $S = \{j \in [a; b] \mid \psi(T)[j] \text{ is empty}\}$ where a and b are the endpoints of the array $\psi(T)$. Applying lemma 1 with this S, a, and b shows the existence of an index i where *any* interval around i in $\psi(T)$ will have a density of "holes" (empty array entries) of at most two times the global density of holes in $\psi(T)$, i.e. of at most 2ϵ. For this i, setting $s = t = i$ and $s = i - 1, t = i + 1$ in lemma 1 shows that $\psi(T)[i]$ and at least one of its direct neighbors are non-empty if $\epsilon < 1/3$. We insert a new element in the tree with a key x lying between $\psi(T)[i]$ and this neighbor. The rest of the proof assumes w.l.o.g. that $\psi(T)[i] < x < \psi(T)[i + 1]$.

2.3.2 Counting node changes

Consider any rebalancing operations ensuing from the insertion of x into the original tree T and let T' be the tree after rebalancing. Let all nodes in T modified by the rebalancing be colored blue and let unmodified nodes be colored white. We apply a recursive splitting procedure on all blue nodes layer for layer in a top-down manner. This procedure will maintain a set of *parts*. Each part is a tuple which contains a connected subgraph of T and a contiguous subset of the elements in $\psi(T)$ which we call a segment. As the subgraph in a part is connected, it has a unique highest node which will be called the root of the part.

Initially, the set of parts contains a single tuple consisting of T and a segment containing all elements of $\psi(T)$. Each splitting step on a blue node splits a part into new parts containing subsets of the original part.

We now describe a splitting step on a blue node v. Let T_v be the subtree of v and let S_v be the elements corresponding to the elements contained in the leaves of T_v mapped into $\psi(T)$. Let (G_p, S_p) be the existing part (where p is the root of G_p) containing v and its subtree. The existing part (G_p, S_p) is replaced by the following new parts.

For each subtree under v, add a new part consisting of the subtree and all elements in the subtree in $\psi(T)$. Refer to fig. 2.5 for an illustration of these subtree parts. When splitting blue

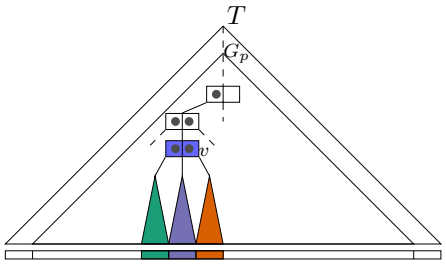
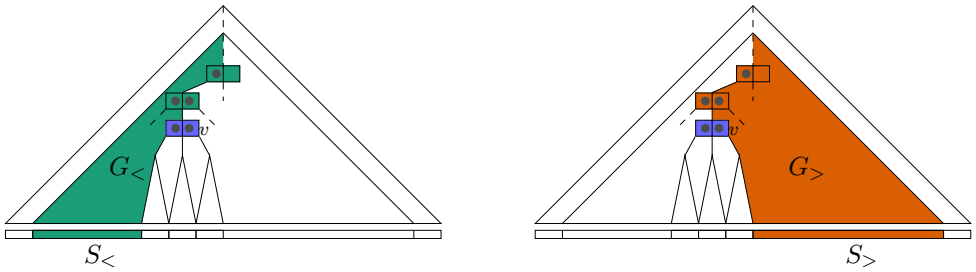


Figure 2.5: Illustrating of a general case of a splitting step on the node v. The subtree parts created in this step (both subgraphs and segments) have been highlighted.

leaf nodes, we create a part for each element stored in the node. Such a part will consist of an empty subgraph and a singleton segment containing the element. Generally, G_p may be a proper superset of T_v (see fig. 2.5). In this case, we create up to two boundary parts for $S_p \setminus S_v$. The segment $S_<$ for the left boundary part $(G_<, S_<)$ consists of all elements in S_p left of S_v. The subgraph $G_<$ consists of all nodes on the path from v to p and all nodes in G_p to the left of this path. Creating the right boundary part is done similarly. See fig. 2.6 for examples of these boundary parts. For later use, we note that this splitting procedure results in $O(B)$ segments



(a) $(G_<, S_<)$ (b) $(G_>, S_>)$

Figure 2.6: Illustration of the boundary parts of fig. 2.5.

per blue node.

The following simple invariants for the splitting procedure are easy to verify: 1) When processing layer k, no part will have a subgraph containing a blue node on any layer $k' > k$. 2) For each part (G, S) where $|S| > 1$, elements of S are in leaves of G. 3) Until the leaves are reached, the process does not create any parts with empty subgraphs.

After the splitting procedure, there will by 1) be no blue nodes within connected subgraphs of the remaining parts. This can be used to show that such subgraphs are not moved up or down.

Lemma 2. *After splitting is done, for parts (G, S) with $|S| > 1$, G will appear in T and T' with the same height.*

Proof. By 3), G will be a non-empty subgraph consisting of white nodes. Hence, this subgraph must appear again in T'. Due to 2), G contains some leaves, hence they appear on the same

height in both T and T' (all leaves appear on the same level, so G cannot have moved vertically between T and T'). □

We now focus on the segments of parts as the construction will impose restrictions on their ability to move around from $\psi(T)$ to $\psi(T')$. The previous lemma implies that segments can only be *shifted* when going from $\psi(T)$ to $\psi(T')$ (meaning that elements appearing in a segment will have the same relative positions to each other in both arrays). How far a segment is shifted can be bounded from below by its size.

Lemma 3. *A segment containing s elements which has different position in $\psi(T)$ compared to $\psi(T')$ has moved by a distance of $\Omega(s)$.*

Proof. For segments of at most one element, the statement is obvious. Consider a part (G, S) with $s = |S| > 1$ and let v be the topmost node of G. By lemma 2, G will be found in T', too, with v having the same height in both trees. As discussed in section 2.3.1, nodes on a given height can only appear in specific positions. When v moves, it translates the elements of S by (at least) a multiple of the size (in number of elements in leaves) of a full subtree under v. By (2), this size is at least s. □

Proof of theorem 1. Suppose that the newly inserted element x lies between two segments, S_{l-1} and S_l. As no holes were available, at least one of S_{l-1} and S_l must have moved in $\psi(T')$ compared to $\psi(T)$. Suppose w.l.o.g. that S_l has moved to the right (potentially along with some segments to the right of S_l). Let j be the index of the right-most element in the last consecutive segment S_r right of i which has moved to the right and let $\ell = l([i; j])$. Due to the choice of i, at most $2\epsilon\ell$ of the entries in $[i; j]$ are empty, so there are at least $(1 - 2\epsilon)\ell$ elements in this interval of $\psi(T)$. All of these elements are in the segments $S_l, S_{l+1}, \ldots, S_r$ by choice of S_r. Segments can only be translated if there is room in the array (available holes). Since E_{r+1} did not move to the right, this implies that none of the segments $S_l, S_{l+1}, \ldots, S_r$ can have moved more than $2\epsilon\ell$, so by lemma 3, they can then only contain $O(2\epsilon\ell)$ elements each. This means that the number of segments $S_l, S_{l+1}, \ldots, S_r$ must be $\Omega\left(\frac{(1-2\epsilon)\ell}{2\epsilon\ell}\right) = \Omega\left(\frac{1}{\epsilon}\right)$. As each blue node accounts for $O(B)$ segments, this implies the existence of $\Omega(1/(\epsilon B))$ blue nodes. □

2.4 Upper bound

In this section, we present a rebalancing scheme for performing insertions into a B-tree containing $B^h(1 - \epsilon)$ elements while maintaining optimal height h. The basic scheme is building on ideas from a rebalancing scheme for binary search tree in [15]. It may also be viewed as a (highly non-trivial) extension of the overflow technique analyzed in [19]. Adding ideas from density keeping algorithms [17], we arrive at the final scheme.

For convenience of notation in the proof, we introduce the parameter $k = 1/\epsilon$. Also, ϵ from now on denotes a value fixed over a sequence of updates (it will be used in such a way that it is never more than a constant factor from its previous meaning, defined by $n = B^h(1 - \epsilon)$).

Theorem 2. *For any ϵ with $0 < \epsilon < (B-1)/B$ there exists a rebalancing scheme for maintaining optimal height h in a B-tree while its size ranges between $(1-\epsilon)B^h$ and $(1-\epsilon/2)B^h$ which has an amortized rebalancing cost per update of*

$$O\left(\frac{k\log^2(\min\{k,B\})}{B}\right),$$

where $k = 1/\epsilon$.

In section 2.4.1, we describe the initial setup of the scheme, and in section 2.4.2 we describe its rebalancing operations. In section 2.4.3, we show how these rebalancing operations can maintain the structure at an amortized rebalancing cost of $O(k)$ node updates per update. In section 2.4.4, we combine the scheme with density keeping methods in order to lower the amortized rebalancing cost to $O(k\log^2(\min\{k,B\})/B)$. We focus on insertions, since rebalancing after deletions can be handled by simply running the operations in reverse. We assume $n = (1-\epsilon)B^h$ initially.

2.4.1 Layout

The main mechanism employed in the structure is the distribution and redistribution of *holes* in the tree. A hole in a node is simply a missing entry—for leaves, this means that an element field is NIL, for internal nodes this means that a search key field and a corresponding pointer field are NIL. We define the *weight* of a hole in a node of height h' to be $B^{h'}$, i.e., the capacity of a subtree of height h'. This is the number of elements missing from the full tree due to this hole.

Since we initially have $n = (1-\epsilon)B^h$, we must initialize the tree such that the sum of weights of all holes in the tree is ϵB^h. We call this sum our weight budget. On each level of the tree, we divide the nodes into horizontal *groups* of contiguous nodes. It will be an invariant of the rebalancing scheme that each group contains between 0 and B holes. Initially, this value is B. The bigger the groups, the more sparse the holes will be. On the leaf level we set the group size (the number of nodes in a group) to $\Theta(k)$. On the levels above, we double the group size for each new level. If we conceptually consider this to happen in the full tree of degree B, the B holes per group will on the leaf level correspond to a constant fraction of the weight budget. By tuning the exact group size on the leaf level, we make this fraction be at most $1/8$. While the weight of a hole increases by a factor B per level, the width of the layers in the full tree decreases by the same factor. Thus, doubling the group size means that the total weight of holes in a level decreases by a factor of two for each level. Hence, the level sums in the full tree form a geometrically decreasing series with total sum at most twice the weight of the leaf level, i.e., at most $1/4$ of the weight budget. Some level $h_{\max}$ will be the last where the group size does not exceed the total number of nodes on that level. On this level, we place $3/4$ of the total weight budget (arbitrarily positioned on the level). We will refer to this as the *reservoir*.

The actual initial layout is built top-down: Above $h_{\max}$ no weight is placed and all nodes have degree B. At level $h_{\max}$, $3/4$ of the total weight budget is placed, which compared to the full tree removes nodes (by removing entire subtrees) on the levels below. The weight on the lower levels (which removes further nodes of the tree during the top-down process) is given by

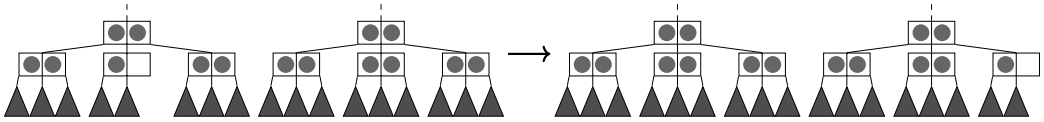


Figure 2.7: Illustration of horizontally sliding a hole within a (sub)tree.

the group sizes defined above and the rule that each group has B holes. Due to the top-down removal of nodes in the tree, the resulting tree will be thinner than the full tree, hence the level sums in the actual tree produced will be smaller than the $1/4$ of the total budget. Hence, after this top-down procedure, there is some budget of left. This is distributed evenly as holes on the leaf level (these holes are for simplicity of argumentation in proofs considered inactive, i.e., they will not take part in the rebalancing process described below and will not be considered in the invariant that groups have at most B holes). Once the shape of the tree has been produced, it is filled with elements and search keys in a bottom-up fashion. Assuming the elements given in sorted order (for instance during a global rebuild), the above process can be done at linear cost $O(n/B)$.

The rebalancing scheme we describe below works until there are no more holes in the reservoir. By the invariant on the number of holes per group in layers below the reservoir, the total weight outside of the reservoir will never exceed the fraction $1/4$, so emptying the reservoir requires a number of insertions proportional to the initial weight in the reservoir. Hence, the scheme will last at least until size $(1 - \epsilon/2)B^h$, as required for theorem 2.

2.4.2 Rebalancing operations

Two basic operations will be used in the rebalancing scheme to move holes around within the tree: *horizontal sliding* and *vertical redistribution*. As the names imply, horizontal sliding will move a hole from one location to another (within a group) on the same level while vertical redistribution moves a hole from one level to another.

The horizontal sliding procedure is illustrated in fig. 2.7. To efficiently access the nodes to be slid on the sliding level, we maintain horizontal level-pointers between nodes which are neighbors on the level. As groups do not necessarily align with subtree boundaries, neighbors on the level may have a lowest common ancestor which is further away than the sliding distance. Since this ancestor is among the nodes whose keys should be changed due to the slide (to maintain search tree order), we maintain pointers to these ancestors for all pairs of neighboring nodes on a level which are not siblings (these pointers and the level-pointers are not shown in fig. 2.7). When sliding, subtrees below the slide level must be moved along with the keys to maintain search tree order of the keys (as shown in fig. 2.7). In that process, the ancestor pointers between the edges of these subtrees may have to be updated, which for each subtree is a number of pointers proportional to its height. Thus, sliding a hole from another node in the current group on a level at height i will require accessing $O(i2^ik)$ nodes, since the group size is 2^ik.

A vertical redistribution transforms one hole on a level $l + 1$ into B holes on level l. It is illustrated in fig. 2.8. This operation is the same as the split operation in standard B-trees. To perform a vertical redistribution, one clearly needs to access less nodes than for a slide at the

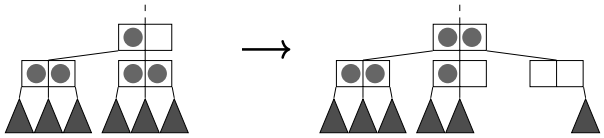


Figure 2.8: Illustrating the vertical redistribution. A hole is moved down one layer, thus allowing the creation of a new node on said layer. This corresponds directly to splitting in standard B-trees.

same level.

2.4.3 Insertion

We now describe how to insert a new element with a given key. As in standard B-trees, we first search for the key in the tree to find a leaf node, v. If v contains a hole, we simply add the element to v as in a standard B-tree and no further work is done. Otherwise, a hole is first moved to v to make room for the new element. This is done by a searching the group of v for a hole to slide to v. If none exists, we ask for the parent of v to obtain a hole, which we then can redistribute to v's level as B holes. If the parent has no hole, the process is repeated recursively. This recursive process, called REQUEST, is described as algorithm 3. The recursion ends at the latest when the reservoir is reached.

Algorithm 3

```
procedure REQUEST(v)
    if any node in v's group has a hole
        slide this hole to v
    else
        REQUEST(parent node of v)
        redistribute a hole from parent
        if v's group is too big
            split v's group
```

Moving down holes from higher levels via a vertical redistribution increases the size of the receiving group by one (see fig. 2.8). To keep the sizes of groups, and hence the cost of sliding within groups, we split a group in two when a redistribution discovers that the size of the receiving group has doubled compared to its initial group size from section 2.4.1. Groups are simply implemented by marking the border nodes of groups, so this splitting is straight-forward to carry out as part of the operation.

As each redistribution provides B holes to a group, algorithm 3 will only recurse upwards (the **else** case) from a group when B new calls of algorithm 3 to nodes in the group have been issued since the last recursion upwards from this group. Recall that a horizontal slide on a level at height i has a cost of $O(i2^i k)$ and that vertical redistributions are also covered by this bound. From this follows an amortized bound on the rebalancing cost of:

$$k + \frac{1 \cdot 2k}{B} + \frac{2 \cdot 2^2 k}{B^2} + \cdots + \frac{i \cdot 2^i k}{B^i} + \cdots + \frac{h_{\max} \cdot 2^{h_{\max}} k}{B^{h_{\max}}} \tag{2.1}$$

which is $O(k)$ (assuming $B > 2$). As is standard, this can formally be proven by a potential function which amounts to each insertion bringing an amount of "coins" equal to eq. (2.1). Coins are conceptually stored in groups and will pay for all rebalancing work. When splitting a group, the new group has an additional need for coins not covered by eq. (2.1). As groups grow slowly, the extra amount compared to eq. (2.1) is low order in its terms, so just doubling eq. (2.1) is more than enough.

2.4.4 Achieving log squared amortized rebalancing

We now describe how to modify the rebalancing procedure such that the amortized cost per insertion becomes:

$$O\left(\frac{k\log^2(\min\{k,B\})}{B}\right)$$

Note that the amortized cost of rebalancing on all non-leaf levels is already as low as $O\left(\frac{k}{B}\right)$. This can be seen by excluding the first term in eq. (2.1). Since $k = \Theta(1/\epsilon)$, that value matches the lower bound, so we only need to lower the cost of rebalancing on the leaf level.

The overall plan is to distribute holes more evenly within leaf groups, both when constructing the tree initially and when vertically redistributing holes from the next higher level. We will use a density keeping scheme [17] on the nodes within each leaf group. Concretely we will use the version described in section 3.1 of [9] (to which we refer for full details). The scheme maintains a distribution of holes in a binary search tree by enforcing density (number of keys relative to the maximum for a given height) bounds on the levels of the tree. In this scheme, each insertion will require redistribution within an interval of size $O\left(\log^2(n)/\tau_1\right)$ (amortized) where n is the size of the array and τ_1 is upper limit on the density for the array (τ_1 can be set to 1/2 here).

We apply this scheme to each group of k leaves such that each leaf node within a group is treated like a leaf node in the binary tree of the density keeping scheme. The binary tree is implicitly overlaid on the group and not actually constructed. The analysis in [9] requires that redistributing everything in a subtree takes linear time in the size of the subtree. We point out that this requirement is satisfied in this setting as the number of nodes involved in rebuilding an interval on the leaf level will be dominated by the distance on the leaf level (as described regarding sliding in section 2.4.3).

For $B \geq k$, we must ensure that a node will sustain $\Theta\left(B/k\right)$ insertions per request for more holes to achieve the desired amortized cost. To this end, we specify how many holes a node gets from its neighbors via a request (slide) and how many holes a node must have in order to be able to give some holes away (this specifies whether the density keeping scheme should consider the node full). We split the B/k holes in each node into two equal portions: one to handle insertions into the node and one to service a request from another node. This means that a node issues a request only after $\Theta\left(B/k\right)$ keys have been inserted into it and that a node is given $\Theta\left(B/k\right)$ holes when it issues a request for holes. A node can only help another node (potentially itself) once—this is also the case in the setting of [9] as each node has capacity one. The amortized number of insertions between each request within the group is now $\Omega(k/B)$ and hence the amortized cost is $O\left(\frac{k\log^2(k)}{B}\right)$.

For $k > B$, not all nodes can get a hole per vertical redistribution. Instead, we apply the density keeping scheme to subgroups of k/B nodes where each such subgroup gets one hole after a vertical redistribution. The distance to a hole within a subgroup is $O(k/B)$ and as there will be B subgroups per actual group, using the density keeping scheme will mean that insertions will have an amortized cost of $O\left(\frac{k\log^2(B)}{B}\right)$.

It is important to note that, in both cases, $\Theta(B)$ of the holes are used before a vertical redistribution is requested (this is necessary to cover the cost of global rebuilding).

To handle deletions, the operations can be performed in reverse, as mentioned earlier. This implies moving holes upwards in the tree (equivalent to merge in standard B-trees) when the number of holes in a group grows sufficiently big.

2.5 Global rebalancing scheme

We here describe how repeated global rebuilding can be used with the scheme presented in section 2.4 to achieve a global rebalancing scheme (i.e., not bound to a specific range of tree sizes as in theorem 2).

Immediately before performing a global rebuild, the tree contains $n = N(1-\epsilon)$ elements for some ϵ, where N is the next power of B. We set up the system from section 2.4 by performing a global rebuild at linear cost (i.e., $O(n/B)$ node updates). We use this set-up while $N(1-2\epsilon) \leq n \leq N(1-\epsilon/2)$. Once one of these bounds have been reached, we rebuild again, updating ϵ by increasing or decreasing it by a factor of two. There will be at least $\Theta(N\epsilon)$ updates between each rebuilding. Thus, the amortized cost incurred by the rebuilding process per update will be $O(1/(B\epsilon))$, which does not increase the upper bound from section 2.4.

Applying this scheme during insertions (with ϵ successively decreasing by factors of two along the way) allows us to maintain optimal height until the tree is arbitrarily full and the update cost hence is arbitrarily close to $\Theta(n/B)$. In practice, one would presumably choose to allow suboptimal height when the complexity of rebalancing exceeds a certain rebalancing budget. fig. 2.1 illustrates how the complexity (lower bound) behaves as n approaches powers of B and how this complexity lower bound relates to a logarithmic budget. Suppose we want to limit the update complexity to a given budget $f(n)$. We can do that by keeping optimal height from $n = B^{h-1}$ up to the point where the budget is exceeded and from that point up to $n = B^h$ allowing the height to be optimal plus one. Specifically, we define ϵ' by $k\log^2(\min\{k, B\})/B = f(n)$ and $k = 1/\epsilon'$. For most $f(n)$, this has for $B^{h-1} < n \leq B^h$ one solution ϵ' for each h, similarly to fig. 2.1. We assume this to be the case for the $f(n)$ in question. We adjust the scheme above in the following way:

- If n exceeds $B^h(1-\epsilon'/2)$, rebuild the tree with the height incremented by one and use $\epsilon = 1/2$.
- If n falls below $B^h(1-\epsilon')$, rebuild the tree with the height decremented by one and use $\epsilon = \epsilon'$.

This gives the result below, where ϵ' is defined as above.

Theorem 3. *There exists a rebalancing scheme such that, given a rebalancing budget $f(n)$, a B-tree can be maintained with perfect height from B^{h-1} to $(1-\epsilon')B^h$ and with perfect height plus one up to B^h.*

We highlight two interesting special cases. The first case is $f(n) = \Theta(\log_B(n))$, which is a natural choice since it allocates the same cost for rebalancing as for searching time. For this choice, $\epsilon' = o(1)$, leading to:

Corollary 1. *There exists a rebalancing scheme that given a rebalancing budget of $\Theta(\log_B(n))$ maintains a B-tree which has optimal height for all but a vanishing fraction of values of n. For the remaining values of n the height is optimal plus one.*

A comparison of the result of corollary 1 to optimal height and to the height bound obtained by standard B-trees is given in fig. 2.2.

The second case is $f(n) = \Theta(1/B)$, for which we have $\epsilon' = \Theta(1)$, leading to:

Corollary 2. *There exists a rebalancing scheme that given a rebalancing budget of $\Theta(1/B)$ maintains a B-tree with optimal height plus one.*

2.6 Storage utilization

We here consider the connection between the parameter ϵ and the storage utilization.

Our upper bound rebalancing scheme directly implies excellent storage utilization. When we lay out a tree with a given k this tree will have a storage utilization of at least $(k-1)/k = 1-\epsilon$. This is trivially true for the leaf level as each group consisting of a multiple of k leaves has at most one empty node worth of holes (refer to section 2.4.1). As holes are more sparse on higher levels of the tree, these levels have higher storage utilization than the leaves. The total weight of holes in the reservoir is at most a constant times the total weight of holes in the leaves. Since the number of nodes allocated in the reservoir is bounded by the number of allocated leaves and the weight per hole in the reservoir will be greater (by a factor of a power of B), the storage utilization among nodes in the reservoir is at least that of the leaves, too.

The scheme in section 2.5 changes ϵ and thus the guaranteed storage utilization while growing a tree. To get a consistent high storage utilization while growing a tree, one can instead choose a desired low ϵ' (lower than needed initially for $n = (1-\epsilon)N$), lay out the holes among the leaves and internal layers according to this ϵ', and leave the extra free storage this would yield in the reservoir. By the same arguments as before, the storage utilization on the leaf and inner levels of the tree would then be $1-\epsilon'$. On the reservoir level, one could compact the nodes such that at most one node would be non-full.

Our lower bound results do not translate directly into a statement about storage utilization. This result is about the amount of room (weight) in a tree of a particular height—the actual number of nodes allocated within the tree is not considered. As a counter-example to implications for storage utilization, take for instance a B-tree with all nodes completely full except the root which has only a tiny fraction of the B possible children. This tree will have excellent storage utilization—it will approach 1 for increasing n—while ϵ will be large as the number of keys could be increased massively without increasing tree height.

Bibliography

[1] Arne Andersson. “Efficient Search Trees”. PhD Thesis. Department of Computer Science, Lund University, Sweden, 1990.

[2] Arne Andersson. “Optimal bounds on the dictionary problem”. In: *Optimal Algorithms*. Vol. 401. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 106–114. ISBN: 978-3-540-46831-8. DOI: 10.1007/3-540-51859-2_10. URL: http://link.springer.com/10.1007/3-540-51859-2_10.

[3] Arne Andersson and Tony W. Lai. “Comparison-efficient and write-optimal searching and sorting”. In: *ISA'91 Algorithms*. Springer Berlin Heidelberg, 1991, pp. 273–282. ISBN: 978-3-540-46600-0.

[4] Arne Andersson and Tony W. Lai. “Fast updating of well-balanced trees”. In: *SWAT 90*. Springer Berlin Heidelberg, 1990, pp. 111–121. ISBN: 978-3-540-47164-6.

[5] Arne Andersson et al. “Binary search trees of almost optimal height”. en. In: *Acta Informatica* 28.2 (Feb. 1990), pp. 165–178. ISSN: 0001-5903, 1432-0525. DOI: 10.1007/BF01237235. URL: http://link.springer.com/10.1007/BF01237235.

[6] David M. Arnow and Aaron M. Tenenbaum. “An empirical comparison of B-trees, compact B-trees and multiway trees”. en. In: *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. ACM Press, 1984, p. 33. ISBN: 978-0-89791-128-3. DOI: 10.1145/602259.602265. URL: http://portal.acm.org/citation.cfm?doid=602259.602265.

[7] R. Bayer and E. M. McCreight. “Organization and maintenance of large ordered indexes”. In: *Acta Informatica* 1.3 (Sept. 1972), pp. 173–189. ISSN: 1432-0525. DOI: 10.1007/BF00288683. URL: https://doi.org/10.1007/BF00288683.

[8] Rudolf Bayer and Karl Unterauer. “Prefix B-trees”. In: *ACM Transactions on Database Systems* 2.1 (Mar. 1977), pp. 11–26. ISSN: 03625915. DOI: 10.1145/320521.320530. URL: http://portal.acm.org/citation.cfm?doid=320521.320530.

[9] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. “Cache Oblivious Search Trees via Binary Trees of Small Height”. In: *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '02. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002, pp. 39–48. ISBN: 0-89871-513-X. URL: http://dl.acm.org/citation.cfm?id=545381.545386.

[10] Trevor Brown. “B-slack trees: Highly Space Efficient B-trees”. en. In: *arXiv:1712.05020 [cs]* (Dec. 2017). arXiv: 1712.05020. URL: http://arxiv.org/abs/1712.05020.

[11] Trevor Brown. "B-slack Trees: Space Efficient B-Trees". en. In: *Algorithm Theory – SWAT 2014*. Vol. 8503. Cham: Springer International Publishing, 2014, pp. 122–133. ISBN: 978-3-319-08404-6. DOI: 10.1007/978-3-319-08404-6_11. URL: http://link.springer.com/10.1007/978-3-319-08404-6_11.

[12] Douglas Comer. "Ubiquitous B-Tree". In: *ACM Comput. Surv.* 11.2 (June 1979), pp. 121–137. ISSN: 0360-0300. DOI: 10.1145/356770.356776. URL: http://doi.acm.org/10.1145/356770.356776.

[13] Paul F. Dietz, Joel I. Seiferas, and Ju Zhang. "A tight lower bound for on-line monotonic list labeling". In: *Algorithm Theory — SWAT '94*. Springer Berlin Heidelberg, 1994, pp. 131–142. ISBN: 978-3-540-48577-3.

[14] Rolf Fagerberg. "Binary search trees: How low can you go?" In: *Algorithm Theory — SWAT'96*. Vol. 1097. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 428–439. ISBN: 978-3-540-61422-7. DOI: 10.1007/3-540-61422-2_151. URL: http://link.springer.com/10.1007/3-540-61422-2_151.

[15] Rolf Fagerberg. "The complexity of rebalancing a binary search tree". In: *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 1999, pp. 72–83.

[16] Scott Huddleston and Kurt Mehlhorn. "Robust balancing in B-trees". In: *Theoretical Computer Science*. Springer, 1981, pp. 234–244.

[17] Alon Itai, Alan G. Konheim, and Michael Rodeh. "A sparse table implementation of priority queues". In: *Automata, Languages and Programming*. Springer Berlin Heidelberg, 1981, pp. 417–431. ISBN: 978-3-540-38745-9.

[18] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN: 0-201-89685-0.

[19] Klaus Küspert. "Storage utilization in B *-trees with a generalized overflow technique". en. In: *Acta Informatica* 19.1 (Apr. 1983). ISSN: 0001-5903, 1432-0525. DOI: 10.1007/BF00263927. URL: http://link.springer.com/10.1007/BF00263927.

[20] Tony W. Lai and Derick Wood. "Updating almost complete trees or one level makes all the difference". In: *STACS 90*. Springer Berlin Heidelberg, 1990, pp. 188–194. ISBN: 978-3-540-46945-2.

[21] Tony Wen Hsun Lai. "Efficient Maintenance of Binary Search Trees". PhD Thesis. Waterloo, Ont., Canada, Canada: University of Waterloo, 1990.

[22] David Maier and Sharon C. Salveter. "Hysterical B-trees". en. In: *Information Processing Letters* 12.4 (Aug. 1981), pp. 199–202. ISSN: 00200190. DOI: 10.1016/0020-0190(81)90101-0. URL: http://linkinghub.elsevier.com/retrieve/pii/0020019081901010.

[23] H.A. Maurer, Th. Ottmann, and H.-W. Six. "Implementing dictionaries using binary trees of very small height". en. In: *Information Processing Letters* 5.1 (May 1976), pp. 11–14. ISSN: 00200190. DOI: 10.1016/0020-0190(76)90094-6. URL: https://linkinghub.elsevier.com/retrieve/pii/0020019076900946.

[24] Arnold L. Rosenberg and Lawrence Snyder. "Compact B-trees". en. In: *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. ACM Press, 1979, p. 43. ISBN: 978-0-89791-001-9. DOI: 10.1145/582095.582102. URL: http://portal.acm.org/citation.cfm?doid=582095.582102.

[25] Balasubramaniam Srinivasan. "An Adaptive Overflow Technique to Defer Splitting in B-trees". In: *The Computer Journal* 34.5 (1991), pp. 397–405. DOI: 10.1093/comjnl/34.5.397. URL: http://dx.doi.org/10.1093/comjnl/34.5.397.

[26] Andrew Chi-Chih Yao. "On random 2–3 trees". In: *Acta Informatica* 9.2 (June 1978), pp. 159–170. ISSN: 1432-0525. DOI: 10.1007/BF00289075. URL: https://doi.org/10.1007/BF00289075.

3 Engineering an Expected $O(\mathbf{Sort}(m))$ External Connected Components Algorithm.

Gerth Stølting Brodal*, Rolf Fagerberg†, David Hammer‡, Ulrich Meyer§, Manuel Penschuck¶, Hung Tran‖

abstract

We empirically investigate algorithms for solving Connected Components in the external memory model. In particular, we study if the randomized $O(\text{sort}(E))$ 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, Meyer and collaborators.

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-Meyer 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 faster in many cases. Those cases are marked by higher density (the threshold is around five, but varying among the graph classes). Also, larger graph sizes seems to benefit Karger-Klein-Tarjan relative to Sibeyn-Meyer, which is in line with its better asymptotic bound on the expected I/O cost. Borůvka's algorithm is not competitive with the two others.

*Aarhus University, gerth@cs.au.dk, Supported by the Independent Research Fund Denmark, grant 9131-00113B.

†University of Southern Denmark, rolf@imada.sdu.dk, Supported by the Independent Research Fund Denmark, Natural Sciences, grant DFF-7014-00041.

‡Goethe University Frankfurt and University of Southern Denmark, hammer@imada.sdu.dk, Supported by the Deutsche Forschungsgemeinschaft (DFG) under grants ME 2088/3-2 and ME 2088/4-2.

§Goethe University Frankfurt, umeyer@ae.cs.uni-frankfurt.de, Supported by the Deutsche Forschungsgemeinschaft (DFG) under grants ME 2088/3-2 and ME 2088/4-2.

¶Goethe University Frankfurt, mpenschuck@ae.cs.uni-frankfurt.de, Supported by the Deutsche Forschungsgemeinschaft (DFG) under grants ME 2088/3-2 and ME 2088/4-2.

‖Goethe University Frankfurt, htran@ae.cs.uni-frankfurt.de, Supported by the Deutsche Forschungsgemeinschaft (DFG) under grants ME 2088/3-2 and ME 2088/4-2.

3.1 Introduction

The Connected Components (CC) problem is a fundamental algorithmic task on undirected graphs and has a large number of uses, 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 straight-forward to solve in linear time using DFS or BFS. It is a long-standing open problem whether MSF also can be solved in linear time. A large body of work has been devoted to the question (see e.g. the references in [17]), leaving the impression that in internal memory, MSF is harder to tackle than CC, at least in terms of the algorithmic sofistication needed (and potentially also in terms of the complexity of the problem).

In external memory, there is a lower bound for CC and MSF of $\Omega(E/V \cdot \text{sort}(V))$ [13] and a number of algorithms that come within at most a logarithmic factor of $O(\text{sort}(E))$. No deterministic algorithm is known to match the lower bound, but a randomized algorithm with $O(\text{sort}(E))$[1] expected cost exists [11, 5]. Unlike the situation in internal memory, the known algorithms in external memory for CC 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 $O(\text{sort}(E))$ algorithm, where a fairly involved subroutine for its MSF variant becomes straight-forward for its CC variant.

The randomized $O(\text{sort}(E))$ external memory algorithm seems never empirically investigated. The aim of this paper is do exactly this, focusing on the CC setting, where the discrepancy mentioned above gives the algorithm the largest opportunity of being competitive in practice. In more detail, we want to investigate implementations and tuning options for this algorithm and to compare the best of our implementations with tuned versions of the practically most promising of the remaining set of (asymptotically slightly worse, but often simpler) external CC algorithms. Due to the large size of internal memory in most current computer systems, it is not clear at all whether a small asymptotic advantage of at most a logarithmic factor will ever materialize in practice for graphs of (very) large, but still plausible sizes. At the same time, our investigations will provide guidance for practitioners wanting to solve CC for large graphs on current hardware.

3.1.1 Previous work

In the semi-external case, where $V < M$, scanning the edges and maintaining the components via a Union-Find data structure will solve CC in $O(\text{scan}(E))$ I/Os. The classic Borůvka MSF algorithm was externalized by Chiang et al. [5] by showing how to implement a Borůvka step in $O(\text{sort}(E))$ I/Os, leading to $O(\log(V/M) \cdot \text{sort}(E))$ cost for the entire algorithm. A simpler method for implementing a Borůvka step in $O(\text{sort}(E))$ I/Os was later given by Arge et al. [3]. Munagala and Ranade [13] gave a CC algorithm using $O(\log\log(VB/E) \cdot \text{sort}(E))$ I/Os, and

[1] Using sparsification, the algorithm can be implemented to have $O(E/V \cdot \text{sort}(V))$ cost [5], matching the lower bound exactly. In this paper, we will consider its $O(\text{sort}(E))$ version, as the two bounds are very close and we want the algorithmic complexity to be kept down.

also proved the above-mentioned lower bound. The algorithm was later generalized to MSF by Arge et al. [3], keeping the I/O bound.

In [11], Karger, Klein, and Tarjan gave an internal MSF algorithm with expected $O(E)$ running time, using a linear time MSF verification algorithm as a central subroutine. Chiang et al. [5] gave an external algorithm for MSF verification. Based on this, sorting, scanning, and external Borůvka steps, it is fairly straight-forward to convert the algorithm of [11] to an external MSF algorithm with expected cost $O(\text{sort}(m))$ [5].

For the CC, it is an easy observation (already made in [2]) that the MSF verification part can be substituted by a simple contraction step, thereby reducing the implementation complexity considerably. To our knowledge, neither the CC nor the MSF variant of this expected $O(\text{sort}(m))$ algorithm has been tested empirically.

A very simple randomized MSF algorithm with an expected I/O cost of $O(\log(V/M) \cdot \text{sort}(E))$ was given by Sibeyn and Meyer and was first reported in Schultes [18]. It was further described and empirically tested by Dementiev et al. [20]. A variant for CC was studied by Sibeyn [21] theoretically and empirically (it was also described and tested empirically in Schultes [19]). Due to its simplicity, the algorithm is likely to have very competitive constants in its I/O-cost expression, which is argued theoretically in [21] and substantiated by the experiments in [20, 21]. However, none of the experiments [20, 21] compared to other external memory algorithms.

3.1.2 Our contribution

We implement the CC version of the $O(\text{sort}(E))$ randomized, external algorithm by Karger, Klein, and Tarjan [11] 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, Meyer, and collaborators [18, 20, 21, 19]. Our experiments are executed on a large set of different graph classes, including random graphs (the Gilbert type where each edge is picked with equal probability), grids, geometric graphs, and hyperbolic graphs.

Among our findings are: The Sibeyn-Meyer 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. Still, with the right tunings, the Karger-Klein-Tarjan algorithm can be implemented to be faster in many cases. Those cases seems marked by higher density (the threshold is around five, but varying among the graph classes). Also, larger graph sizes seems to benefit Karger-Klein-Tarjan relative to Sibeyn-Meyer, which is in line with its better asymptotic bound on the expected I/O cost. Borůvka's algorithm is not competitive with the two others.

3.2 Definitions

In this paper, all graphs $G = (V, E)$ are undirected. As is common, we are overloading the symbols V and E: When used as a set, V denotes the set of nodes, when used as a number, V denotes the number of nodes (and similarly with E and edges).

The cost of algorithms are analysed in the standard I/O-model of Aggarwal and Vitter [1], where M denotes the size of the internal memory and B denotes the block size. By $\text{scan}(N)$ we

denote the I/O cost $\Theta(N/B)$ of scanning and by sort(N) the I/O cost $\Theta(N/B \log_{M/B}(N/M))$ of sorting.

The Connected Components (CC) problem is to identify the partition of V induced by the equivalence relation $\sim$ defined by: $u \sim v$ iff u is connected to v by a path in G. As input, we will assume the standard external memory representation of a graph as a list of edges.[2] All our algorithms need the edges to be sorted lexicographically. To avoid an initial sorting step, we assume the input to be sorted—this can only make differences in running times between algorithms larger. Unless otherwise stated, we also assume that each unordered edge (u, v) is stored once and in *normalized* form, i.e. with $u \leq v$. As output, we will require a mapping $f : [0 \dots V-1] \to [0 \dots V-1]$ where $f(v) = f(u)$ iff u and v are in the same connected component. We choose that this output should be returned as the pairs $\{(v, f(v)) \mid v \in V\}$, i.e., in the form of a graph consisting of stars. In all algorithms, each star center will be some chosen representative node from the corresponding connected component.

By a *relabeling* of a graph $G = (V, E)$ by a mapping $f : [0 \dots V-1] \to [0 \dots V-1]$, we mean applying f to all edge endpoints and then removing parallel edges and self-loops in the resulting graph. As a relabeling can be performed by $O(1)$ sorts and scans of E, this has cost sort(E).

By a *contraction* G/E' of a graph $G = (V, E)$ by a subset $E' \subseteq E$ of its edges we mean solving CC on $G' = (V, E')$ and then relabeling G by the mapping f' returned.

Note that if we solve CC on the contracted graph G/E' and by f'' denote the mapping represented by the graph of stars returned, then we can solve CC on the original graph G as follows: Use the graph of stars representing f'' to relabel the edges in the graph of stars representing f' (only one endpoint of each edge is affected by the relabeling). Then merge those relabeled edges with the edges of the graph of stars representing f''. It is easy to see that this will produce a graph of stars representing the solution f to CC on the original graph G. All recursive algorithms in the current paper use that as their overall framework.

3.3 Algorithms

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

Union-Find In the semi-external case, where $V < M$, scanning the edges once while maintaining a Union-Find data structure on V in internal memory allows the CC to be solved in $O(\text{scan}(E))$ I/Os (and $O(E\alpha(E))$ CPU time [22], where α is the inverse Ackermann function). We will use this algorithm 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 the graph $G' = (V, E')$, each node is in a connected component of size at least two, so the number of nodes is at least halved in the step. Assuming that a Borůvka step can be implemented

[2] This means that isolated nodes should be handled separately by the user, which is straight-forward as they constitute their own connected components.

at $O(\text{sort}(E))$ cost, this leads to a recursive algorithm which will use $O(\log_2(V/M)\text{sort}(E))$ I/Os before the semi-external base case is reached. This is Borůvka's algorithm.

The first part of a Borůvka step is to find E'. This is done as follows: double E during a scan to make it contain for each edge both its original normalized version and its reversal, then choose a minimum incident edge for all nodes via a single sort and scan of this doubled version of E. This has $O(\text{sort}(E))$ cost.

To implement the remainder of a Borůvka step, one can exploit that $G' = (V, E')$ 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 met. If we assume that all edge weights are unique, the weights along any path are strictly decreasing, except when closing a two-cycle by traversing the same undirected edge in two directions. Hence, the cycles must all be two-cycles. By sorting E' using a comparator which first normalizes its two input edges, all two-cycles $(u, v), (v, u)$ can be found. Removing the edge (v, u) from E' leaves the corresponding connected component as a tree with v as root. Declaring v to be the ID of that connected component, the task is now to distribute those IDs of connected components in G' to the rest of the nodes of the trees, in order to be able to return the star graph representing the mapping f' needed for the contraction.

For this task, early external methods for Borůvka steps [5, 2] used algorithms for Euler tours of trees which again are based on list ranking algorithms. A simpler method was described in [3], which is the one we will use. It is based on the fact that edge weights are strictly increasing on root-to-leaf paths in the trees. This allows edge weights to be used as a "time line" for coordinating the step-wise propagation of IDs downwards in the trees (the entire process is often called "time forward processing"). The propagation is done for all trees simultaneously by maintaining a set of ID propagation signals in an external priority queue, with the priority queue key being the edge weight of the edge above the destination node v of the signal. When a signal is removed from the priority queue by a delete-min operation, we scan the child edges of v and for each insert a signal with the same ID and with the key being the weight of the child edge. The priority queue is initialized by for each tree root inserting signals for all its children.

As a preprocessing step, we first create a list L of all 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. This preprocessing can be done via a few sorts and scans of the tree edges. After this preprocessing, all accesses to child edges during the propagation process combined constitute a single scan of L. Hence, the second part of a Borůvka step has $O(\text{sort}(V))$ cost. For further details, see [3].

In the CC setting, the above algorithm for a Borůvka step can be implemented by (formally) assigning to all edges their 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. Also note that this weight assignment fulfills the assumption of edge weights being unique.

Karger-Klein-Tarjan We now describe the CC version of the $O(\text{sort}(E))$ randomized, external algorithm based on Karger, Klein, and Tarjan [11]. It is a recursive algorithm with the following structure:

1. Perform three Borůvka steps in the input graph. Let the result be $G = (V, E)$.

2. Sample $E_1 \subseteq E$ by independently sampling each edge in E with probability $1/2$. Let $G_1 = (V_1, E_1)$ be the graph represented by E_1.
3. Compute the connected components $\mathsf{CC}(G_1)$ recursively.
4. Contract G by $\mathsf{CC}(G_1)$. Let the result be G'.
5. Recursively compute $\mathsf{CC}(G')$.
6. Relabel $\mathsf{CC}(G_1)$ using $\mathsf{CC}(G')$ and merge the result with $\mathsf{CC}(G')$ (as per the last paragraph of Section 3.2).
7. Perform the corresponding relabelings and merges based on the three contractions of the initial Borůvka steps (as per the last paragraph of Section 3.2).

Note that in step 4, the stars found representing $\mathsf{CC}(G_1)$ are not impacted in contraction which means that only the edges $E \setminus E_1$ need to be processed.

The input to the second recursive call is $G' = G/\mathsf{CC}(G_1)$ represented by its edge list. Having been contracted by $\mathsf{CC}(G_1)$, the only edges remaining are edges between connected components of G_1. The crux of the Karger-Klein-Tarjan algorithm is that the number of these edges is $O(n)$ in expectation. The argument for this is as follows.

To bound the expected number of edges left in G', consider each edge $e \in E_1$ as if the edges were taken in a Kruskal-like fashion, building up a spanning forest F of G_1 one edge at a time. As the sampling is done independently for each edge, we can analyse this process in arbitrary order of the edges.

1. If e forms a cycle with already sampled edges, we need not count it
   - If it were sampled, it (or another edge on the same cycle) would not become part of F
   - If it were not sampled, it would become an intra-CC edge and thus not end in G'
2. If e does not close a cycle and *is* sampled in E_1, it counts towards F
3. If e does not close a cycle and *is not* sampled in E_1, it counts towards G'

The number of edges counted towards G' in the third case might be an overestimate: It may be the case that later edges are sampled which, with edges already counted towards G', would form cycles. There may also be several edges in G' between any pair of computed CCs; these are easy to identify as parallel edges given the CC IDs.

To bound the expected number of remaining edges, note first that $|F| \leq V - C$ where C is the number of connected components of G. The decision of whether or not to include an edge in the sample is independent of previous samples, so cases 2 and 3 are equally likely. Thus, the expected size of E'' is at most the size of $|F|$.

This argument is analogous to Lemma 2.1 of [11] for the MSF version. The rest of the argument in [11] for the expected cost carries over almost verbatim, with $O(E)$ time substituted by $O(\text{sort}(E))$ I/O cost.

Sibeyn-Meyer The MSF algorithm presented in [20] is designed to be simple yet I/O-efficient for practical applications. The algorithm works by repeatedly letting one node select its minimum incident edge and contracting it. The contractions are done in a lazy fashion using the time-forward processing method, with the time dimension being the node IDs.

The original algorithm is described in two versions using different main data structure(s): one using buckets and the other using a priority queue. As the bucket version involves parameter choices during implementation which potentially makes it break for some graphs, we in this paper focus on the version based on priority queues.

By first randomizing the node IDs, the algorithm is shown [20] to generate expected $O(E \log(V/V'))$ priority queue operations when contracting the node set V to a smaller node set V' (i.e. for contracting $V - V'$ nodes). Stopping the contractions when the semi-external $V' < M$ case is met and finishing using Kruskals algorithm gives an expected $O(\log(V/M) \cdot \text{sort}(E))$ algorithm.

The priority queue version of the algorithm can be described as follows:

- Initialize Q as a PQ containing all the input edges (u, v, w), ordered by u and w
- while $|Q| > 0$:
  - $(u, v, w) \leftarrow \text{ExtractMin}(Q)$
  - Output (u, v, w) as an MSF-edge
  - while $|Q| > 0$ and the minimum edge left in Q has source u
    - $(u, v', w') \leftarrow \text{ExtractMin}(Q)$
    - $\text{Insert}(Q, (\min\{v, v'\}, \max\{v, v'\}', w'))$

As mentioned, the algorithm in effect uses time-forward processing to propagate information from smaller to larger node IDs. When a new source u appears as the minimum, all edges incident to u have already been "sent" to u and the minimum-weight edge among these is the first one extracted. Note that to get a correct MSF, the forwarded edges will in the original algorithm be annotated with the original nodes.

In our setting, where the goal is to just compute the connected components of a graph, the algorithm can be simplified in a number of ways (see [19]). The tree that the algorithm will output only matters for connectivity and as such, the "edges" of this tree need not be edges from the original input E, so there is no need to annotate forwarded edges with original node IDs. As edge weight is not relevant for connectivity, one can choose an arbitrary edge out of the "current" source u as the new target to forward edges to. A natural choice would be 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 neighbour (or known reachable node due to forwarded edges) immediately.

The final CC information should be represented as a set of stars, so some post-processing has to be done on the trees output by this modified algorithm. Observing that the node IDs give a topological ordering of the tree edges, one can simply reverse all the tree edges and use time-forward processing in the opposite direction of the initial algorithm:

- Initialize a priority queue Q ordered by decreasing u.
- for each reversed tree edge (u, v)
  - $(u, v) \leftarrow \text{ExtractMax}(Q)$
  - Output (u, v, w) as an MSF-edge
  - Remove from Q any messages (u', v') where $u' > u$
  - If there is a message (u, v') on top of Q, then $v \leftarrow v'$
  - Output (v, u)
  - Add message (v, u) to Q

Randomized Borůvka A standard Borůvka step has a first part where each node selects an outgoing edge, and a second part where the connected components of this edge set E' are found via time-forward processing and then returned as a mapping represented by a star graph. A Borůvka step is guaranteed to contract V by at least a factor of two.

We now describe a novel randomized method for the second part which is simpler than time-forward processing, at the cost of a lower contraction factor. In the empirical part of the paper, we then investigate if this trade-off is beneficial for the overall I/O cost when using Borůvka steps (either as part of Borůvka's algorithm or in the Karger-Klein-Tarjan algorithm).

The idea is simple: let 1) each node keep its selected edge with probability p and 2) in the set E'' of remaining edges mark all edges (u, v) 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.

We now bound the expected size of E'''. E' has size V, so the expected size of E'' is pV. If we for each edge (w, u) in E' count a mark if (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^2V. Thus, the expected size of E''' is at least $p(1-p)V$, 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 expected at least $1(1-1/2)V = V/4$ nodes are removed. Thus, the expected contraction factor is at least $1/(1-1/4) = 4/3$. The actual contraction for a given graph may of course be larger than this, just as for standard Borůvka steps and its lower bound of two on the contraction factor. We will in the empirical section investigate what typically happens in practice.

We note in passing that when finding E' in the first part of a Borůvka step, there is for the method above no need to choose a minimum incident edge according to some assigned unique edge weights, since there is no need for the cycles of the connected components to have size two with the above method (as opposed to the method based on time-forward processing on trees).

3.4 Tuning Options

We investigate several variations of the algorithms with potential for impact on their 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 be expected to 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 Karger-Klein-Tarjan algorithm (before its first recursive call), nodes are contracted. We investigate if time-forward process based Borůvka steps or the proposed randomized version will be the fastest. The general form of the running time argument in [20, 18] states that if the Sibeyn-Meyer algorithm is run until the number of nodes has been contract from V to V', it uses expected $O(\log(V/V') \cdot \text{sort}(E))$ I/Os. Thus, another possible contraction sub-routine in Karger-Klein-Tarjan is to use Sibeyn-Meyer.

Omitting node contractions at the root in Karger-Klein-Tarjan From the details of the cost analysis of Karger-Klein-Tarjan [11], it seems likely that the initial contraction in the root node of its recursion tree will dominate the running time. The asymptotic result of expected $O(\text{sort}(E)$ cost will still hold 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 constitute two sorts and scans of E (if we enter the base case in the second recursive call, we can even save one of the sorts, 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 others 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, an option could be to make p depend on the density (lower p when higher density).

Which neighbor to contract in Sibeyn-Meyer In each step of Sibeyn-Meyer, 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 [21], it may be beneficial to choose the neighbor with largest node ID. We want to investigate what is the best choice and the gains possible, and we compare empirically choosing

a neighbor with largest node ID, a neighbor with smallest node ID, and a random neighbor (which corresponds to the MSF version).

3.5 Implementation

All algorithms are implemented in C++ using the STXXL library [6], which offers highly tuned external memory versions of fundamental algorithmic building blocks like sorting and priority queues. It also supports pipelining which avoids the need for unnecessary round trips to disk. For the external priority queue used in several places in the algorithms implemented we also use the one provided by STXXL.

In order to accommodate different contraction schemes as replacements in the contraction of the recursive Karger-Klein-Tarjan algorithm, we implemented a general framework which generically performs the sampling, contraction, relabeling and merging in the algorithm's execution depending on the given contraction. As of now, currently considered contraction schemes are Sibeyn-Meyer, Karger-Klein-Tarjan and randomized Borůvka contractions. The framework comes in two flavors: as a purely vector-based and a pipelined stream-based implementation. Despite the fact that the stream-based implementations perform superiorly, the extent of their benefits were a priori not clear. Additionally, as our framework does not primarily depend on a priority-queue, unlike the pure Sibeyn-Meyer algorithm, several small optimizations can be added.

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 efficient data structures of STXXL. For this, any generated data is not saved in an explicit vector but fed to a container that functions as a data stream which can only be accessed in a read-only fashion. 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 runtime and I/O-complexity [4].

Semi-external base case While we assume that the number of nodes in the original input is known exactly, this is not true during any recursive subcall in the programs execution. As edge reductions are performed due to either node contractions, relabeling or sampling, the exact number of nodes in the current subproblem is not known without any additional computational efforts. While computing the exact number of nodes only requires an additional cost of a few scanning and sorting steps, the addition leads to infeasibly long running times. We therefore only upper bound the currently present number of nodes in the processed edges

and invoke the semi-external base case accordingly. For the contractions we use the pessimistic worst-case contraction factor in order to guarantee good worst-case behaviour, i.e. a factor of 2 for an original Borůvka step and a factor of $4/3$ for a randomized Borůvka step, respectively. During sampling, we count the number of distinct source nodes and update the upper bound accordingly[3].

By using these upper bounds we can additionally save I/Os in the relabeling as relabeled edges may immediately be piped into the semi-external base case without the otherwise required final sorting step.

3.6 Graph classes

For our experiments, we turn to a variety of different synthetic graph models. In order to cover an extensive range of different parameters, we generate sufficiently large graphs using scalable graph generators. For a recent overview of such generators, see [16]. We consider four types of graphs: the Gilbert type classic random graphs, random geometric and random hyperbolic graphs, both belonging to the class of spatial network models, and finally deterministically generated grid graphs.

Gilbert graphs In the $\mathcal{G}(n, p)$ model of Gilbert [9], each edge is present independently with probability p. While the $\mathcal{G}(n, p)$ model can emit graphs with a varying number of connected components for sufficiently small p, its degree distribution is considered atypical for real-world instances.

Random Geometric graphs Random Geometric graphs (RGGs) [8, 14] are a simple case of spatial networks where graphs are projected onto Euclidean space. In RGGs 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 [7].

Random Hyperbolic graphs Random Hyperbolic graphs (RHGs) [12, 10] are a special case of spatial networks where graphs are projected onto hyperbolic space. We describe the threshold model, the simplest RHG variant [10]. 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 RHGs is that the node degrees follow a power law distribution which is often observed 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 [10]. To generate graphs in this model, we use the HyperGen generator [15].

[3] We note that streaming algorithms can be employed to improve the estimates. The additional internal work should however be limited as the sampling process is I/O-bound.

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.

3.7 Experiments

3.7.1 Evaluating the Randomized Borůvka algorithm

We first aim to investigate the contraction ratio achieved by our simple randomized Borůvka steps described in section 3.3. We consider a number of variants motivated by various possible edge list representations:

- Simple undirected version: each edge $\{u, v\}$ is represented by both (u, v) and (v, u) in the edge list. When sampling an edge for a node u, we actually sample from all edges incident to u as described in section 3.3.
- Directed version: each edge is only represented once as (u, v). This saves space and I/O volume—in our large-scale timing experiments, edges in the edge list are only represented once. For this algorithm, this means that some nodes may not sample an edge even though they have degree greater than zero.
- Directed oriented version: edges are additionally all oriented from the node with lower ID to the node with higher ID. This representation corresponds to the storage format we use for the large-scale experiments. Note that this guarantees that the sampled edges cannot form cycles. For pathological node IDs, this may however give rise to long paths.

For the directed variants, one can additionally consider randomly flipping the edge orientation (for the simple directed version) or randomly permuting the node IDs before orienting the edges (for the directed orienting version) to mitigate pathological behavior.

Recall that in the analysis in section 3.3, p is the probability that each node keeps its chosen edge. For each of the following plots, we performed edge sampling based on the variants described above for different values of p. We then removed all marked edges, breaking paths of length greater than one and counted the number of edges remaining: these remaining star edges would each contract one node. This allows us to compute the ratio of nodes which is contracted by one randomized Borůvka step. We compare this to the contraction ratio for a standard deterministic Borůvka step.

We did this for various types of graphs. The simplest case is that of a path graph—results are plotted for this in fig. 3.1. The node IDs given by the generator are increasing along the path,

which makes this the worst-case input for the "1-way, ordered" variant; for p=1, it samples the entire path and will only contract one node along this path. Fitting the analysis from section 3.3, the best choice of p for this variant on this input is $p = 1/2$ where approximately $1/4$ of the nodes would be contracted. Permuting the node IDs first ("1-way, permuted, ordered") effectively mitigates this problem to the extent that $p = 1$ seems the best choice.

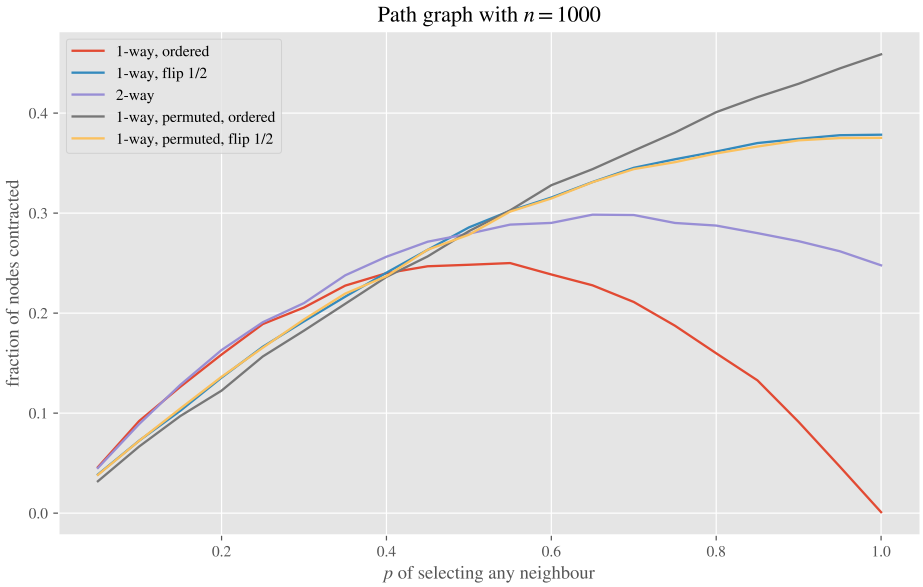


Figure 3.1: Contraction ratios achieved by variants of a randomized Borůvka step on a path graph. In comparison, a standard Borůka step achieved a contraction rate of more than 0.74 on average.

For the remaining graphs tested, preemptively discarding edges by setting $p < 1$ generally does not seem worthwhile. In random Gilbert graphs, permuting the node IDs does not seem necessary; see figs. 3.2 and 3.3. The directed, oriented variants outperform the others which all behave very similarly. Increasing the density of the random Gilbert graph (from 2 in fig. 3.2 to 50 in fig. 3.3) slightly increases the contraction ratio for the randomized variants; for the best variant, approximately half the nodes would be contracted. Note however that it also increases the contraction ratio achieved by a full Borůvka step to quite close to 1.

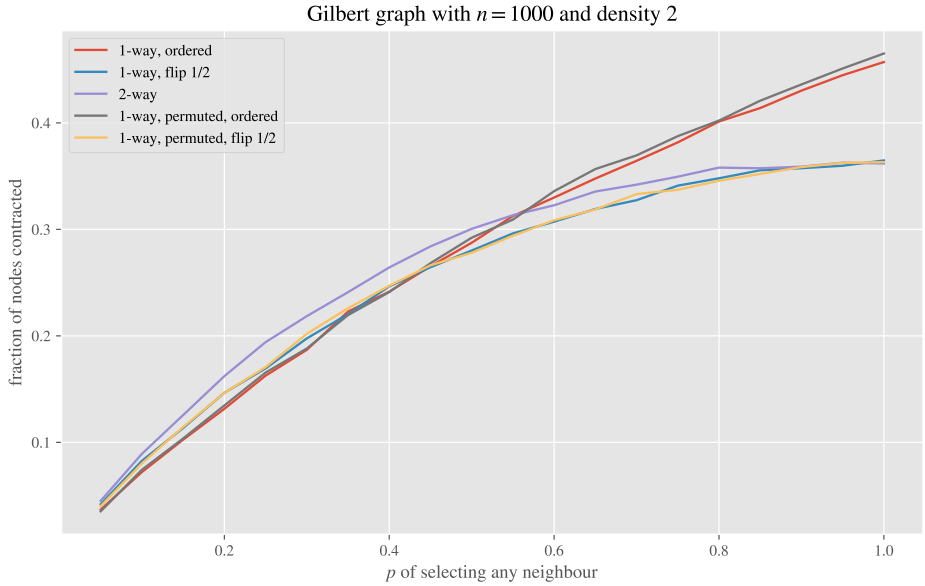


Figure 3.2: Contraction ratios achieved by variants of a randomized Borůvka step on a sparse random Gilbert graph. In comparison, a standard Borůka step achieved a contraction rate of more than 0.86 on average.

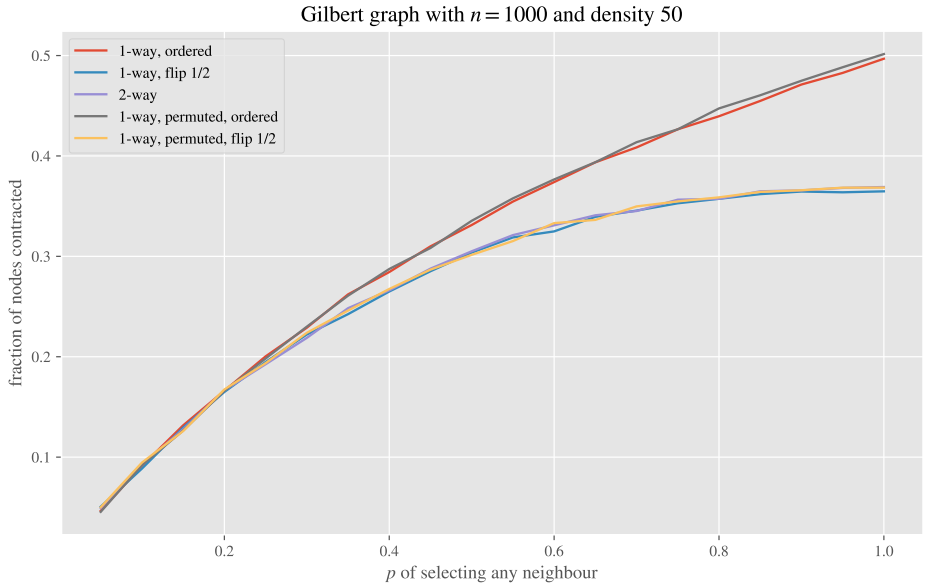


Figure 3.3: Contraction ratios achieved by variants of a randomized Borůvka step on a somewhat dense random Gilbert graph. In comparison, a standard Borůka step achieved a contraction rate of more than 0.99 on average.

In our timing experiments, we found that running the Karger-Klein-Tarjan with randomized Borůvka steps (specifically the directed oriented version, which here seems the best overall choice) as the node contraction was consistently worse than using the Sibeyn-Meyer algorithm for the node contraction instead. Depending on the instance, the version using randomized Borůvka steps was slower than the version using Sibeyn-Meyer node contraction by a factor between 1.5 and 2 (plots omitted here).

Our overall conclusion from the investigations above is that the randomized Borůvka steps do not offer any improvements over the standard version. Our tuning choices for our remaining experiments therefore exclude this variant.

3.7.2 Evaluating the impact of contraction edges for Sibeyn-Meyer

As mentioned in section 3.3, for the CC version of Sibeyn-Meyer, there is a free choice of which neighbor to send information to (i.e., which incident edge to contract along). One choice suggested as a good in [21, 19] is to send as far forward in time as possible. We want to experimentally investigate the impact of this choice. For this, we run the node contraction phase of Sibeyn-Meyer on a various graphs until all nodes have been contracted (without switching to a base case). During this, we record how many messages were inserted into the PQ for each node.

For choosing which incident edge to contract along, we consider three possible choices: first, random or last. Here, first and last refers to sending information to the nearest, respectively farthest neighbor (this is including the messages to later nodes available). Random means picking a neighbor at random—this is to simulate the work done in the MST algorithm where, due to node ID permutation, the lightest incident edge is essentially random.

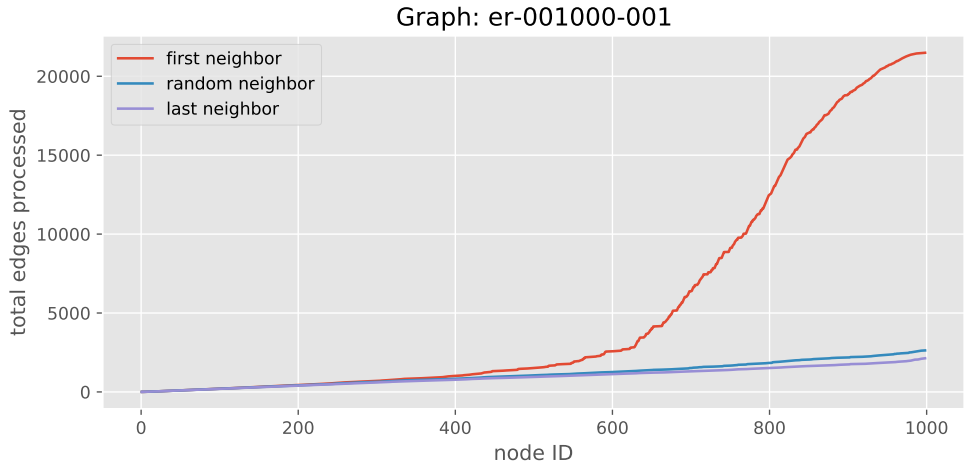


Figure 3.4: Message volume for the three different choices of neighbor selection when run on a very sparse random graph.

As seen clearly in fig. 3.4, forwarding information to the closest neighbor in time leads to vastly increased message volume. We omit this option from the remaining plots in fig. 3.5.

Instead, we plot the cumulative degrees for the nodes processed as a basis for evaluating the total amount of messages. As the sum of the degrees of all nodes is $2E$, matching the cumulative degrees essentially means processing a number of messages linear in E.

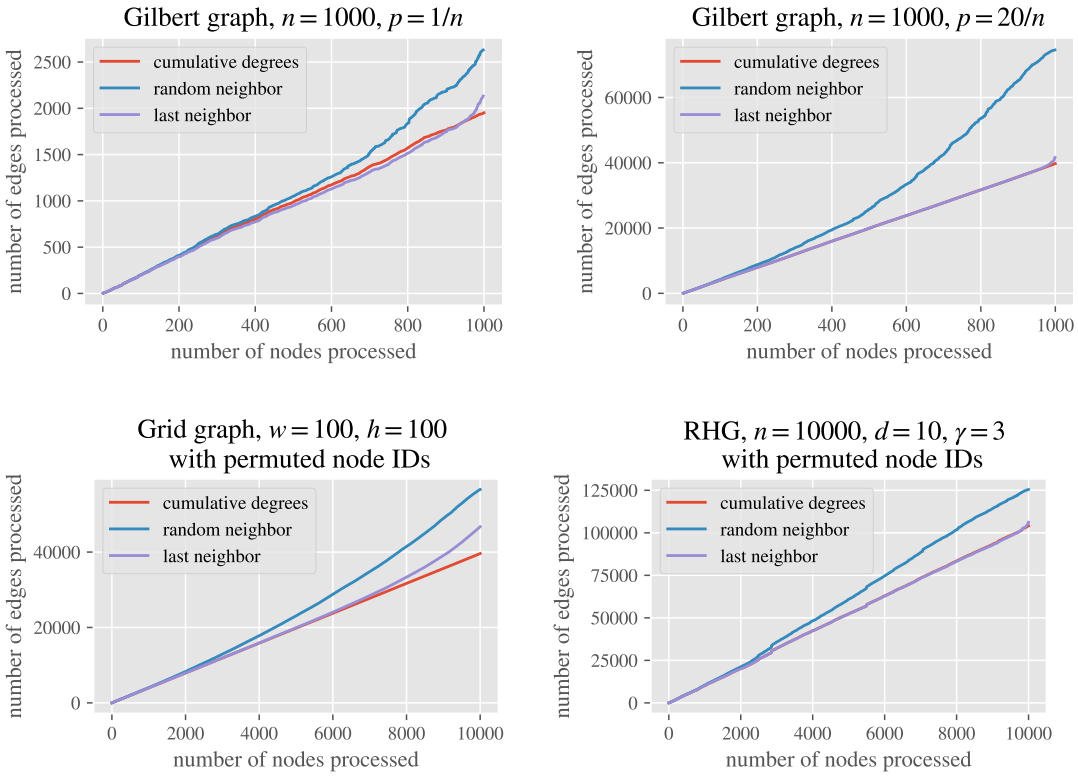


Figure 3.5: Message volume of forwarded messages for different graphs.

The expected upper bound given in [20] on the number of edges processed by the algorithm when contracting from V nodes down to V' nodes is $2E \ln(V/V')$. We suspect that choosing the last neighbor for time forward processing improves the algorithm asymptotically below this bound. We generated a number of random Gilbert graphs with different V, all with the same density. In fig. 3.6, the total number of messages handled by the PQ (that is, after contracting until the PQ is empty) divided by the upper bound is plotted against V.

We observe that the plot for last neighbor selection does not seem to converge to some constant value whereas the plot for random neighbor selection does. This suggests that – at least for Gilbert graphs – sending information to the last neighbor yields an improvement in the asymptotic behavior of the algorithm.

3.7.3 Timing experiments for fully external CC computation

We now proceed to our main set of experiments, all on graphs fully in external memory ($M < \min\{V, E\}$).

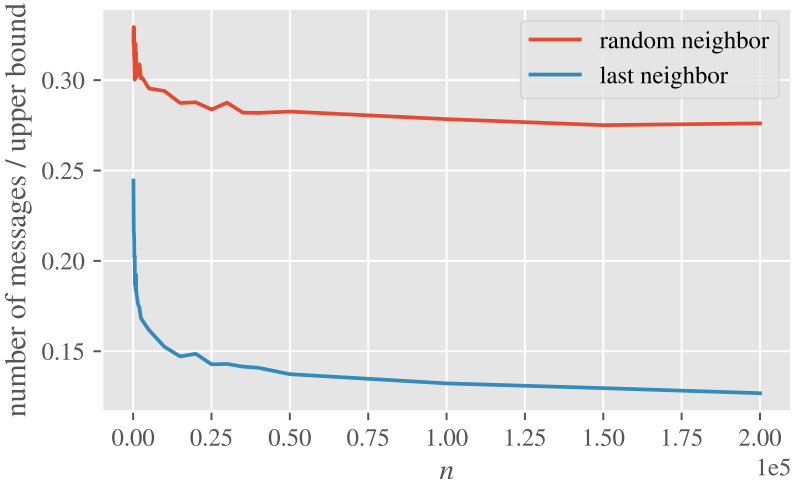


Figure 3.6: Normalized number of forwarded messages for $\mathcal{G}(n, p)$ graphs for increasing values of n. The value p is chosen s.t. a density of 5 is fixed.

Experimental setup

The experiments were run on nodes in the Goethe-HLR cluster. Individual nodes each have an Intel Xeon Skylake Gold 6148 CPUs and 192 GB of RAM (though, to compare algorithms based on their I/O performance within a reasonable time frame, we did not utilize the full amount of RAM available). 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`.

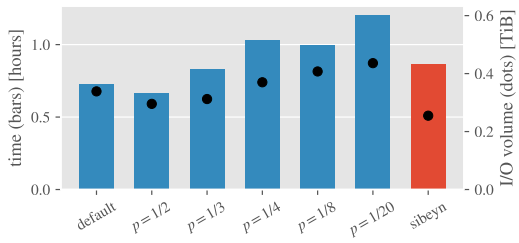
In each run, the input graph was first loaded onto the local hard drive in the appropriate STXXL data structure: an edge stream for the stream-based implementations and an STXXL vector for the vector-based implementations. The threshold for switching to the semi-external base case was for all the algorithms 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 plots in the following subsections 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 limit the amount of memory used, 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, however, have a mechanism to enforce a strict bound on the memory actually allocated. To force disk accesses rather than additional buffering, the `direct` flag was used for the STXXL disk file.

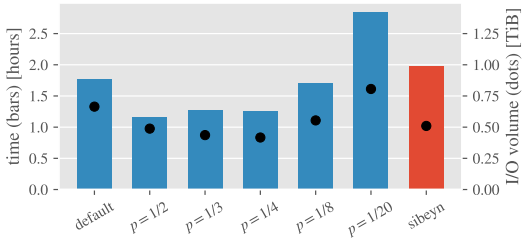
We measured several variants of the Karger-Klein-Tarjan algorithm. As mentioned in section 3.4, it may be worthwhile to skip node contraction in the root. Additionally, we vary the sampling probability. In the plots, "default" refers to the variant which *does* perform contrac-

tion in all levels of recursion (including the root) and samples with $p = 1/2$. The five other variants do *not* perform contraction in the root and have varying p from $1/2$ down to $1/20$. Though our implementation does support setting p locally in each recursive call based on estimated graph density, the measurements presented here use fixed p values in each variant across all calls.

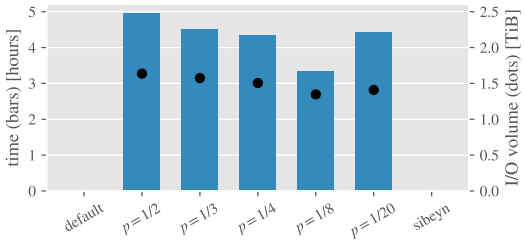
Gilbert graphs



(a) $m = 1.34 \times 10^9$, $m/n = 2.04$, graph size 20.00 GiB



(b) $m = 3.36 \times 10^9$, $m/n = 5.00$, graph size 50.00 GiB



(c) $m = 1.34 \times 10^{10}$, $m/n = 20.00$, graph size 199.99 GiB

Figure 3.7: Running times and I/O volume for $\mathcal{G}(n, p)$ graphs with a node set size of 5 GiB and varying density. Missing entries are due to running times exceeding a time limit of at least ten hours.

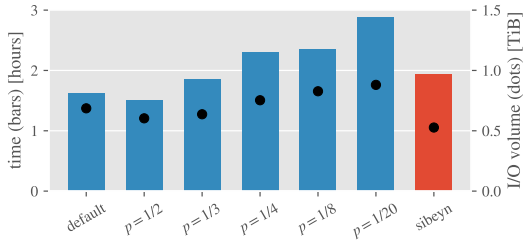
We measured the performance of the Karger-Klein-Tarjan algorithm and the Sibeyn-Meyer algorithm on a number of Gilbert graphs, varying the number of nodes and density. Generally, the Sibeyn-Meyer performs extremely well on sparse graphs whereas the Karger-Klein-Tarjan algorithm (using Sibeyn-Meyer node contraction) performs better as density increases. This pattern is most noticeable for Gilbert graphs.

Looking at graphs with 5 GB of node IDs (671,088,640 nodes), we see that Sibeyn-Meyer has lower I/O volume for density 2 (see fig. 3.7a) whereas variants of the Karger-Klein-Tarjan algorithm win for density 5 as seen in fig. 3.7b. Interestingly, despite having higher I/O volume for density 2 than Sibeyn-Meyer, the best variant of the Karger-Klein-Tarjan algorithm has slightly lower wall time.

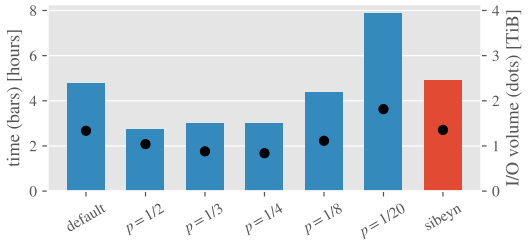
Increasing the density, the Sibeyn-Meyer algorithm performs worse. The algorithms were

run on a graph with the same number of nodes and density 20—the results for the Karger-Klein-Tarjan variants are found in fig. 3.7c, but the Sibeyn-Meyer algorithm timed out after 20 hours.

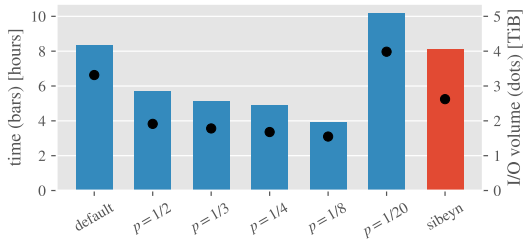
Increasing the number of nodes, the same general pattern can be observed when varying the density. We see that the advantage for I/O volume which Sibeyn-Meyer has for density 2 gets relatively smaller for larger graphs with the same density; compare figs. 3.7a, 3.8a, 3.9a and 3.10a.



(a) $m = 2.68 \times 10^9$, $m/n = 2.04$, graph size 40.00 GiB



(b) $m = 6.71 \times 10^9$, $m/n = 5.00$, graph size 100.00 GiB



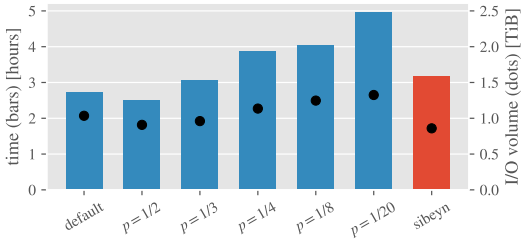
(c) $m = 1.34 \times 10^{10}$, $m/n = 10.00$, graph size 199.99 GiB

Figure 3.8: Running times and I/O volume for $\mathcal{G}(n, p)$ graphs with a node set size of 10GiB and varying density.

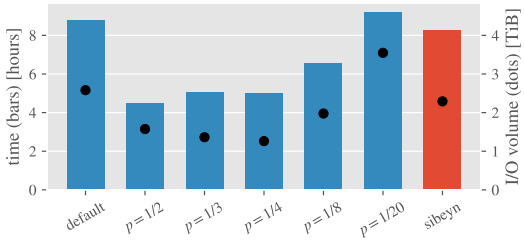
Generally, looking at the performance of the Karger-Klein-Tarjan algorithm, lowering the sampling parameter p seems better when density increases. In most cases—this is seen when comparing figs. 3.10a and 3.10b in particular—the best choice of p seems inversely proportional to the density, as one would expect. In fig. 3.10a, we see that $p = 1/2$ performs the best when density is 2 while in fig. 3.10b, $p = 1/4$ is the best choice when density is 5. Note that this pattern is more consistent in I/O volume than wall time. This suggests that adapting p based on the estimated density in each call may be beneficial.

Grids

The simple grid graphs with only horizontal and vertical edges are rather sparse, having a density of 2 (not accounting for nodes on the boundary of the grid). For these, Sibeyn-Meyer performs extremely well—see fig. 3.11a. Path graphs are grids with one dimension set to one;

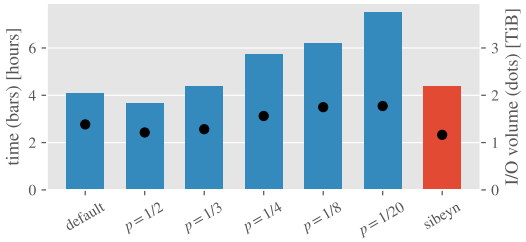


(a) $m = 4.03 \times 10^9$, $m/n = 2.04$, graph size 60.00 GiB

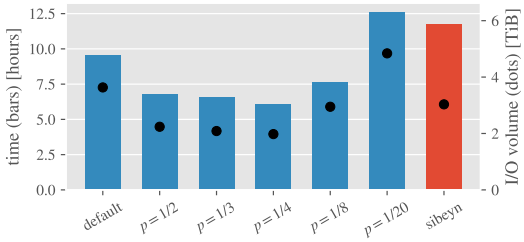


(b) $m = 1.01 \times 10^{10}$, $m/n = 5.00$, graph size 150.00 GiB

Figure 3.9: Running times and I/O volume for $\mathcal{G}(n, p)$ graphs with a node set size of 15GiB and varying density.



(a) $m = 5.37 \times 10^9$, $m/n = 2.04$, graph size 80.00 GiB



(b) $m = 1.34 \times 10^{10}$, $m/n = 5.00$, graph size 200.01 GiB

Figure 3.10: Running times and I/O volume for $\mathcal{G}(n, p)$ graphs with a node set size of 20GiB and varying density.

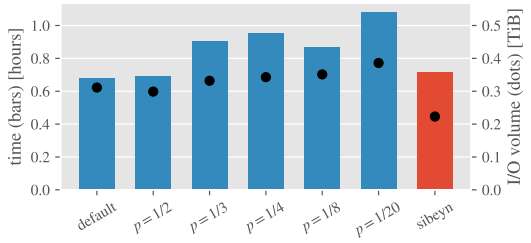
in this case, the gap between Sibeyn-Meyer and the best variant of our Karger-Klein-Tarjan implementation is significant (see fig. 3.11b).

Adding edges to all neighbors of distance 2 as described in section 3.6 increases the density to approximately 12. At this point, the Karger-Klein-Tarjan variants outperform Sibeyn-Meyer; see fig. 3.12a. Note, however, that the Karger-Klein-Tarjan variants win by a smaller margin here than on Gilbert graphs of comparable density; compare for instance figs. 3.8c and 3.12a where the latter has density 10, but a much larger performance gap.

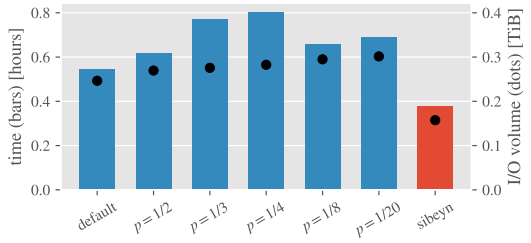
Adding more components (layers) to the generalized grid graphs as described in section 3.6 does not seem to change much. In fig. 3.12b, there are 100 layers and the dimensions have been changed to get approximately the same file size. The relative performance of the algorithms measured is almost identical to the results in fig. 3.12a which had only one component.

Random Geometric graphs

For testing on random geometric graphs, we ran the algorithms on a sequence of graphs with 2^{30} nodes and varying densities. As with the other classes considered so far, Sibeyn-Meyer is

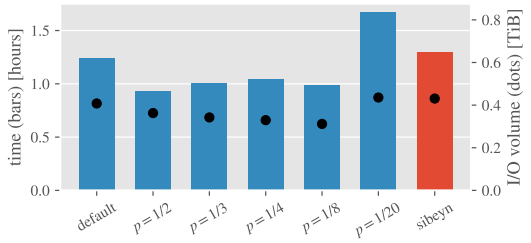


(a) $m = 1.34 \times 10^9$, $m/n = 2.00$, graph size 20.00 GiB

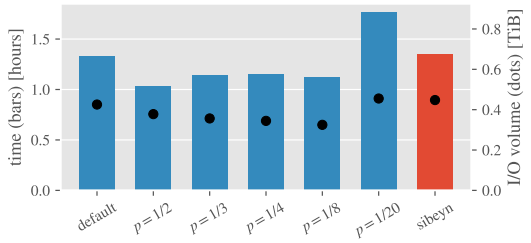


(b) $m = 1.34 \times 10^9$, $m/n = 1.00$, graph size 20.00 GiB

Figure 3.11: Running times and I/O volume 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) $m = 3.89 \times 10^9$, $m/n = 11.96$, graph size 57.93 GiB



(b) $m = 4.05 \times 10^9$, $m/n = 12.00$, graph size 60.35 GiB

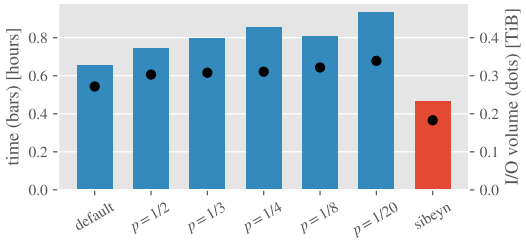
Figure 3.12: Running times and I/O volume for cubes with the parameters (a) one layer and $(w, h, d) = (18\,000, 18\,000, 2)$ and (b) 100 layers and $(w, h, d) = (2600, 1300, 2)$.

faster for sparse graphs; clearly winning in fig. 3.13a where density is close to 1 and fig. 3.13b.

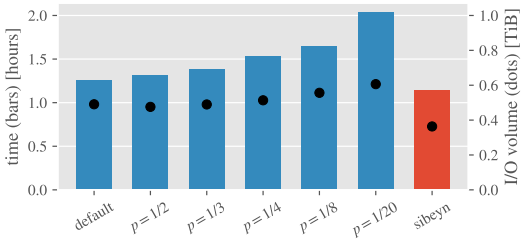
For higher densities, the Karger-Klein-Tarjan variants start outperforming Sibeyn-Meyer as seen in figs. 3.13c and 3.13d. As we observed for grid graphs, the improvement over Sibeyn-Meyer is smaller on RGGs than for Gilbert graphs of comparable density. The RGG used for fig. 3.13c has approximately density 5 and a number of nodes between that of the Gilbert graph for fig. 3.7b and the one for fig. 3.8b. For those two Gilbert graphs, the Karger-Klein-Tarjan algorithm without node contraction in the root and p between $1/2$ and $1/4$ clearly outperform the Sibeyn-Meyer. For the RGG, however, the variants with $p = 1/2$ and $p = 1/3$ have only slightly better wall time and slightly higher I/O volume than Sibeyn-Meyer.

Random Hyperbolic graphs

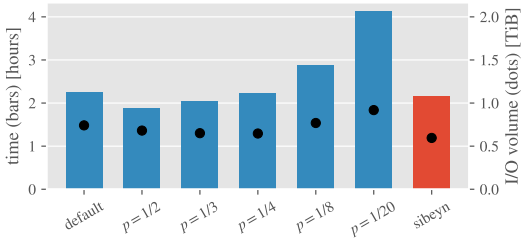
For random hypegeometric graphs, the Sibeyn-Meyer algorithm performs extremely well. Though the variants of Karger-Klein-Tarjan algorithm do seem to be competitive in terms of wall time already for density between 3.5 and 4 (figs. 3.14a and 3.14b), their I/O volume i consistently



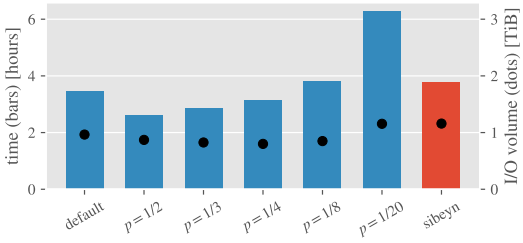
(a) $n = 9.18 \times 10^8$, $m = 1.04 \times 10^9$, $m/n = 1.14$, graph size 15.55 GiB



(b) $n = 1.06 \times 10^9$, $m = 2.62 \times 10^9$, $m/n = 2.46$, graph size 38.99 GiB



(c) $n = 1.07 \times 10^9$, $m = 5.48 \times 10^9$, $m/n = 5.11$, graph size 81.66 GiB



(d) $n = 1.07 \times 10^9$, $m = 8.14 \times 10^9$, $m/n = 7.59$, graph size 121.24 GiB

Figure 3.13: Running times and I/O volume for RGGs with roughly $n = 2 \times 10^{30}$ and varying density. Node IDs are permuted.

higher than that of Sibeyn-Meyer.

Increasing density to approximately 8 (figs. 3.14c and 3.14d), they start achieving comparable I/O volume and in better wall time. Interestingly, lowering the sampling probability p below $1/2$ for these two graphs does not improve wall time.

When density is further increased to around 17—see figs. 3.14d and 3.14e)—Karger-Klein-Tarjan variants noticeably outperform Sibeyn-Meyer. The relationship between p, wall time, and I/O volume is much less consistent than for the other graph classes reviewed. In both of these plots, we see that the best wall time is achieved for the variant with $p = 1/8$ whereas the lowest I/O volume is achieved for the variant with $p = 1/3$—which has the worst wall time among all the variants without root node contraction.

3.8 Conclusion

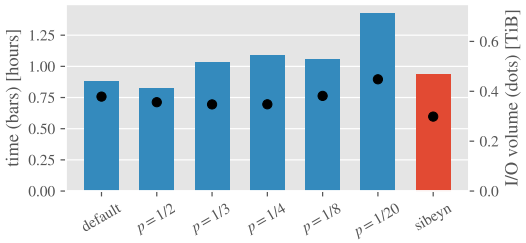
Among our main findings are:

- The Sibeyn-Meyer algorithm is a strong contender. One reason is that it is very simple, using essentially only a priority queue. 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 contracting each node with the last of its

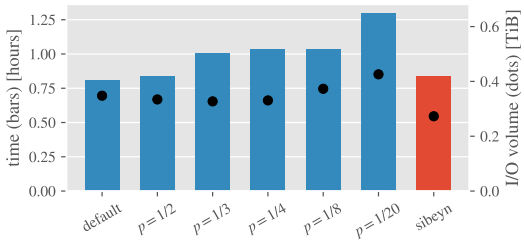
neighbors seems to lower the number of signals generated, as also argued in [21]. From the plots in Section 7.2, it seems to only exceed $2E$ signals in total at the very end of the execution, which in the external version is preempted by entering the semi-external base case. Hence, it in practice is making less than $2E$ priority queue operations. Very few sorts and scans of the input edge list can be performed by a competing algorithm before it will lose to Sibeyn-Meyer.

- Still, with the right tunings, the Karger-Klein-Tarjan algorithm can be implemented to be faster in many cases. For all graph classes in our experiments, there seems to be a density threshold (generally not far from five) above which Karger-Klein-Tarjan is the fastest of the two algorithms. The threshold is lowest for random Gilbert graphs, highest for random hyperbolic graphs, and somewhere in between for grid graphs and random geometric graphs. For a fixed density, larger graph sizes seems to benefit Karger-Klein-Tarjan relative to Sibeyn-Meyer, which is in line with its better asymptotic bound on the expected I/O cost.
- Borůvka's algorithm was not able to compete with the above two.

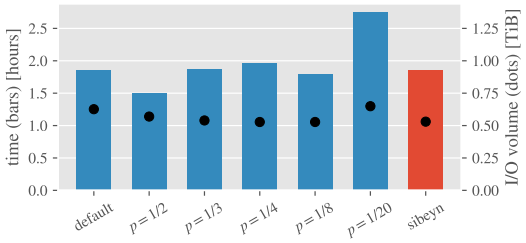
Note that we found Sibeyn-Meyer to be the best contraction algorithm for use in Karger-Klein-Tarjan. We also found that for dense graphs, not making contractions at the root is an advantage for Karger-Klein-Tarjan. A natural next step will be to investigate an adaptive hybrid algorithm, which based on V, E, and possibly the graph class (if known by the user) makes choices between strong contraction at the root via Sibeyn-Meyer (essentially using this for most of the work), or skipping contraction at the root. Also the sampling parameter should probably be chosen adaptively.



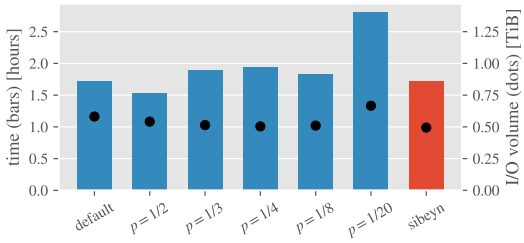
(a) $n = 6.54 \times 10^8$, $m = 2.61 \times 10^9$, $m/n = 3.99$, graph size 38.83 GiB, $\gamma = 3$



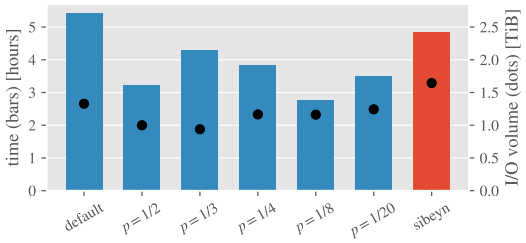
(b) $n = 6.46 \times 10^8$, $m = 2.36 \times 10^9$, $m/n = 3.65$, graph size 35.12 GiB, $\gamma = 4$



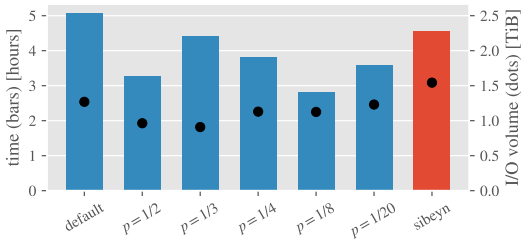
(c) $n = 6.70 \times 10^8$, $m = 5.57 \times 10^9$, $m/n = 8.30$, graph size 82.95 GiB, $\gamma = 3$



(d) $n = 6.70 \times 10^8$, $m = 5.22 \times 10^9$, $m/n = 7.80$, graph size 77.81 GiB, $\gamma = 4$



(e) $n = 6.71 \times 10^8$, $m = 1.18 \times 10^{10}$, $m/n = 17.61$, graph size 176.07 GiB, $\gamma = 3$



(f) $n = 6.71 \times 10^8$, $m = 1.14 \times 10^{10}$, $m/n = 17.01$, graph size 170.13 GiB, $\gamma = 4$

Figure 3.14: Running times and I/O volume for RGGs with roughly $n = 2 \times 10^{40}$, degree exponent $\gamma \in \{3, 4\}$ and varying density. Node IDs are permuted.

Bibliography

[1] Alok Aggarwal and Jeffrey Scott Vitter. "The Input/Output Complexity of Sorting and Related Problems". In: *Commun. ACM* 31.9 (1988), pp. 1116–1127. DOI: 10.1145/48529.48535. URL: https://doi.org/10.1145/48529.48535.

[2] Susanne Albers, Andreas Crauser, and Kurt 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.

[3] Lars Arge, Gerth Stølting Brodal, and Laura Toma. "On external-memory MST, SSSP and multi-way planar graph separation". In: *J. Algorithms* 53.2 (2004), pp. 186–206. DOI: 10.1016/j.jalgor.2004.04.001. URL: https://doi.org/10.1016/j.jalgor.2004.04.001.

[4] Andreas Beckmann, Roman Dementiev, and Johannes Singler. "Building a parallel pipelined external memory algorithm library". In: *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*. IEEE, 2009, pp. 1–10. DOI: 10.1109/IPDPS.2009.5161001. URL: https://doi.org/10.1109/IPDPS.2009.5161001.

[5] Yi-Jen Chiang et al. "External-memory Graph Algorithms". In: *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '95. event-place: San Francisco, California, USA. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1995, pp. 139–149. ISBN: 0-89871-349-8. URL: http://dl.acm.org/citation.cfm?id=313651.313681.

[6] Roman Dementiev, Lutz Kettner, and Peter Sanders. "STXXL: standard template library for XXL data sets". In: *Softw. Pract. Exp.* 38.6 (2008), pp. 589–637. DOI: 10.1002/spe.844. URL: https://doi.org/10.1002/spe.844.

[7] Daniel Funke et al. "Communication-free Massively Distributed Graph Generation". In: *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21 – May 25, 2018*. 2018.

[8] E. N. Gilbert. "Random Plane Networks". In: *Journal of the Society for Industrial and Applied Mathematics* 9.4 (1961), pp. 533–543. DOI: 10.1137/0109045.

[9] Edgar N. Gilbert. "Random Graphs". In: *Ann. Math. Statist.* 30.4 (Dec. 1959), pp. 1141–1144. DOI: 10.1214/aoms/1177706098. URL: https://doi.org/10.1214/aoms/1177706098.

[10] Luca Gugelmann, Konstantinos Panagiotou, and Ueli Peter. "Random Hyperbolic Graphs: Degree Sequence and Clustering". In: *Proceedings of the 39th International Colloquium Conference on Automata, Languages, and Programming - Volume Part II*. ICALP'12. Warwick, UK: Springer-Verlag, 2012, pp. 573–585. ISBN: 9783642315848. DOI: 10.1007/978-3-642-31585-5_51. URL: https://doi.org/10.1007/978-3-642-31585-5_51.

[11] David R. Karger, Philip N. Klein, and Robert E. Tarjan. "A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees". In: *Journal of the ACM* 42.2 (Mar. 1, 1995), pp. 321–328. ISSN: 00045411. DOI: 10.1145/201019.201022.

[12] Dmitri Krioukov et al. "Hyperbolic geometry of complex networks". In: *Phys. Rev. E* 82 (3 Sept. 2010), p. 036106. DOI: 10.1103/PhysRevE.82.036106. URL: https://link.aps.org/doi/10.1103/PhysRevE.82.036106.

[13] Kameshwar Munagala and Abhiram Ranade. "I/O-Complexity of Graph Algorithms". In: *SODA*. Vol. 99. 1999, pp. 687–694.

[14] Mathew D. Penrose. *Random Geometric Graphs*. Oxford University Press, 2003. ISBN: 9780198506263. DOI: 10.1093/acprof:oso/9780198506263.001.0001. URL: http://www.maths.bath.ac.uk/%5C%7Emasmdp/rgg.html.

[15] Manuel Penschuck. "Generating Practical Random Hyperbolic Graphs in Near-Linear Time and with Sub-Linear Memory". In: *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*. Vol. 75. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 26:1–26:21. DOI: 10.4230/LIPIcs.SEA.2017.26. URL: https://doi.org/10.4230/LIPIcs.SEA.2017.26.

[16] Manuel Penschuck et al. "Recent Advances in Scalable Network Generation". In: *CoRR* abs/2003.00736 (2020). arXiv: 2003.00736. URL: https://arxiv.org/abs/2003.00736.

[17] Seth Pettie and Vijaya Ramachandran. "An optimal minimum spanning tree algorithm". In: *J. ACM* 49.1 (2002), pp. 16–34. DOI: 10.1145/505241.505243. URL: https://doi.org/10.1145/505241.505243.

[18] Dominik Schultes. "External Memory Minimum Spanning Trees". http://algo2.iti.kit.edu/schultes/emmst/emmst_short.pdf. Bachelor thesis. Universität des Saarlandes, 2003.

[19] Dominik Schultes. "External Memory Spanning Forests and Connected Components". en. http://algo2.iti.kit.edu/dementiev/files/cc.pdf. 2003.

[20] Jop Sibeyn et al. "Engineering an External Memory Minimum Spanning Tree Algorithm". en. In: *Exploring New Frontiers of Theoretical Informatics*. Vol. 155. Boston: Kluwer Academic Publishers, 2004, pp. 195–208. ISBN: 978-1-4020-8140-8. DOI: 10.1007/1-4020-8141-3_17. URL: http://link.springer.com/10.1007/1-4020-8141-3_17.

[21] Jop F. Sibeyn. "External Connected Components". en. In: *Algorithm Theory - SWAT 2004*. Vol. 3111. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 468–479. ISBN: 978-3-540-22339-9 978-3-540-27810-8. DOI: 10.1007/978-3-540-27810-8_40. URL: http://link.springer.com/10.1007/978-3-540-27810-8_40.

[22] Robert Endre Tarjan. “Efficiency of a Good But Not Linear Set Union Algorithm”. In: *J. ACM* 22.2 (1975), pp. 215–225. DOI: 10.1145/321879.321884. URL: https://doi.org/10.1145/321879.321884.

4 Simulating Population Protocols in Sub-Constant Time per Interaction

Petra Berenbrink*, David Hammer†, Dominik Kaaser‡, Ulrich Meyer§, Manuel Penschuck¶, Hung Tran‖

The version of the paper reproduced here is the short version accepted into ESA 2020. A longer version [18] of the paper is available at `http://arxiv.org/abs/2005.03584`.

Abstract

We consider the efficient simulation of population protocols. In the population model, we are given a system of n agents modeled as identical finite-state machines. In each step, two agents are selected uniformly at random to interact by updating their states according to a common transition function. We empirically and analytically analyze two classes of simulators for this model. First, we consider sequential simulators executing one interaction after the other. Key to the performance of these simulators is the data structure storing the agents' states. For our analysis, we consider plain arrays, binary search trees, and a novel Dynamic Alias Table data structure. Secondly, we consider batch processing to efficiently update the states of multiple independent agents in one step. For many protocols considered in literature, our simulator requires amortized sub-constant time per interaction and is fast in practice: given a fixed time budget, the implementation of our batched simulator is able to simulate population protocols several orders of magnitude larger compared to the sequential competitors, and can carry out 2^{50} interactions among the same number of agents in less than 400 s.

4.1 Introduction

We consider the *population model*, introduced by Angluin et al. [8] to model systems of resource-limited mobile agents that interact to solve a common task. Agents are modeled as finite-state machines. The computation of a *population protocol* is a sequence of pairwise interactions of agents. In each interaction, the two participating agents observe each other's states and update their own state according to a transition function common to all agents.

*Universität Hamburg, Germany, petra.berenbrink@uni-hamburg.de
†University of Southern Denmark and Goethe University Frankfurt, Germany, hammer@imada.sdu.dk
‡Universität Hamburg, Germany, dominik.kaaser@uni-hamburg.de
§Goethe University Frankfurt, Germany, umeyer@ae.cs.uni-frankfurt.de
¶Goethe University Frankfurt, Germany, mpenschuck@ae.cs.uni-frankfurt.de
‖Goethe University Frankfurt, Germany, hung@ae.cs.uni-frankfurt.de

Typical applications of population protocols are networks of passively mobile sensors [8]. As an example, consider a flock of birds, where each bird is equipped with a simple sensor. Two sensors communicate whenever their birds are sufficiently close. An application could be a distributed disease monitoring system raising an alarm if the number of birds with high temperature rises above some threshold. Further processes which resemble properties of population protocols include chemical reaction networks [38], programmable chemical controllers at the level of DNA [22], or biochemical regulatory processes in living cells [21].

While the computational power of population protocols with constantly many states per agent is well understood by now (see below), less is known about the power of protocols with state spaces growing with the population size. In this setting, much interest has been on analyzing the runtime and state space requirements for *probabilistic* population protocols, where the two interacting agents are sampled in each time step independently and uniformly at random from the population. This notion of a probabilistic scheduler allows the definition of a *runtime* of a population protocol. The runtime and the number of states are the main performance measures used in the theoretical analysis of population protocols.

For the theoretical analysis of population protocols, a large toolkit is available in the literature. Consequently, the remaining gaps between upper and lower bounds for many quantities of interest have been narrowed down: for many protocols, the required number of states has become sub-logarithmic, while the runtime approaches more and more the (trivial) lower bounds for any meaningful protocol. So far, when designing new protocols, simulations have always proven a versatile tool in getting an intuition for these stochastic processes. However, once observables are of order $\log \log n$ and below, naive population protocol simulators fail to deliver the necessary insights (e.g., $\log \log n \leq 5$ for typical input sizes of $n \leq 2^{32}$). Our main contribution in this paper is a new simulation approach allowing to execute a large number of interactions even if the population size exceeds 2^{40}. In the remainder of this section, we first give a formal model definition in section 4.1.1 and then describe our main contributions and related work in sections 4.1.2 and 4.1.3.

4.1.1 Formal Model Definition

In the *population model*, we are given a distributed system of n agents modeled as finite-state machines. A *population protocol* is specified by a state space $Q = \{q_1, \ldots, q_{|Q|}\}$, an output domain Y, a transition function $\delta\colon Q \times Q \to Q \times Q$, and an output function $\gamma\colon Q \to Y$. At time t, each agent i has a state $s_i(t) \in Q$, which is updated during the execution of the protocol. The current output of agent i in state $s_i(t)$ is $\gamma(s_i(t))$. The *configuration* $C(t) = \{s_1(t), \ldots, s_n(t)\}$ of the system at time t contains the states of the agents after t interactions. For the sake of readability, we omit the parameter t in $C(t)$ and $s_i(t)$ when it is clear from the context. The *initial configuration* is denoted $C(0) = C_0$.

The *computation* of a population protocol runs in a sequence of discrete time steps. In each time step, a *probabilistic scheduler* selects an ordered pair of agents (u, v) independently and uniformly at random to *interact*. Agent u is called the *initiator* and agent v is the *responder*. During this *interaction*, both agents u and v observe each other's state and update their states according to the transition function δ such that $(s_u(t{+}1),\ s_v(t{+}1)) \leftarrow \delta\,(s_u(t),\ s_v(t))$.

A given problem for the population model specifies the agents' initial states, the output do-

Table 4.1: Simulating N interactions among n agents in $|Q|$ states. For MULTIBATCHED, we restrict $|Q| = \omega(\sqrt{\log n})$. Values indicated by † hold in expectation.

	Simulator	Section	Time Complexity	Space Complexity (bits)
Sequential	SEQ$_{\text{Array}}$	section 4.2	$\Theta(N)$	$\Theta(n \log \lvert Q \rvert)$
	SEQ$_{\text{Linear}}$	section 4.2	$\mathrm{O}(N\lvert Q \rvert)$	$\Theta(\lvert Q \rvert \log n)$
	SEQ$_{\text{BST}}$	section 4.2	$\Theta(N \log \lvert Q \rvert)$	$\Theta(\lvert Q \rvert \log n)$
	SEQ$_{\text{Alias}}$	section 4.2	$\Theta(N)$ w.h.p.	$\Theta(\lvert Q \rvert \log n)$
Batch	BATCHED	section 4.4	$\mathrm{O}(N(\log n + \lvert Q \rvert^2)/\sqrt{n})^\dagger$	$\Theta(\lvert Q \rvert \log n)$
	MULTIBATCHED	section 4.5	$\mathrm{O}(N\lvert Q \rvert \sqrt{\log n}/\sqrt{n})^\dagger$	$\Theta(\lvert Q \rvert \log n)$

main O, and (a set of) *desired (output) configurations* for a given input. As an example, consider the MAJORITY problem. Each agent is initially in one of two states q_A and q_B corresponding to two opinions A and B. Assuming that A is the initially dominant opinion, the protocol concludes once all agents u give $\gamma(s_u) = A$ as their output. Any configuration in which all agents output the initially dominant opinion is a desired configuration.

This notion of a desired configuration allows to formally define two notions of a *runtime* of a population protocol. The *convergence time* T_C is the number of interactions until the system enters a desired configuration and never leaves the desired configurations in a given run. The *stabilization time* T_S is the number of interactions until the system enters a desired *stable* configuration for which there does not exist any sequence of interactions due to which the system leaves the desired configurations. A population protocol is *stable*, if it always eventually reaches a desired output configuration.

A number of variants of this model are commonly used. For *symmetric protocols*, the order of the interacting agents is irrelevant for the transition. In particular, this means that if $\delta(q_u, q_v) = (q'_u, q'_v)$, then $\delta(q_v, q_u) = (q'_v, q'_u)$. In *protocols with probabilistic transition functions*, the outcome of an interaction may be a random variable. In *one-way protocols*, only the initiator updates its states such that $\delta(q_u, q_v) = (q'_u, q_v)$ for any interaction.

Model Assumptions We assume a *meaningful* protocol which converges after at most $N = \text{poly}(n)$ interactions, has an $\mathrm{O}(1)$ time transition function δ, and uses $|Q| < \sqrt{n}$ states (observe that many relevant protocols only use $|Q| = \mathrm{O}(\text{polylog}\, n)$ states; see section 4.1.3).

4.1.2 Our Contributions

In this paper, we present a new approach for simulating population protocols. Our simulator allows us to efficiently simulate a large number, N, of interactions for large populations of size n. Our findings are summarized in table 4.1.

Sequential Simulators As a baseline, we directly translate the population model into a sequential algorithm framework SEQ: SEQ selects for each interaction two agents uniformly at random, updates their states, and repeats. We analyze the runtime and memory consumption of various variants in section 4.2.

Batch Processing To speed up the simulation, we introduce and exploit *collision-free runs*, a sequence of interactions where no agent participates more than once. Our algorithms Batched and MultiBatched coalesce these independent interactions into batches for improved efficiency. Batched first samples the length ℓ of a collision-free run. It then randomly pairs ℓ independent agents, adds one more interaction —the collision— reusing one of the run's agents, and finally repeats. Batched is presented in section 4.4 and extended into MultiBatched in section 4.5. We discuss practical details and heuristics in section 4.6.

Dynamic Alias Tables The simulation of population protocols often needs an *urn-like* data structure to efficiently sample random agents (marbles) and update their states (colors). The *alias method* [41, 40] enables random sampling from arbitrary discrete distributions in $O(1)$ time. However, it is static in that the distribution may not change over time. Thus, we extend it and analyze a *Dynamic Alias Table* in section 4.2. It supports sampling with and without replacement uniformly at random (u.a.r.) and addition of elements (if the urn is sufficiently full). Due to its practical performance and simplicity compared to more general solutions [30, 33], we believe this data structure might be of independent interest and show:

Theorem 4. *Let U be a Dynamic Alias Table that stores an urn of n marbles, where each marble has one of k possible colors. U requires $\Theta(k \log n)$ bits of storage. If $n \geq k^2$, we can*

- *select a marble u.a.r. from U with replacement in expected constant time,*
- *select a marble u.a.r. from U without replacement in expected amortized constant time,*
- *and add a marble of a given color to U in amortized constant time.*

4.1.3 Related Work

Population Protocols The population model was introduced in [8], assuming a constant number of states per agent. Together with [7, 9], they show that all semilinear predicates are stably computable in this model. In the following, we focus on two prominent problems, Majority and Leader Election. For a broad overview, we refer to surveys [12] and [26].

In [5] a Majority protocol with three states is presented where the agents agree on the majority after $O(n \log n)$ interactions w.h.p. (*with high probability* $1 - n^{-\Omega(1)}$), if the initial numbers of agents holding each opinion differ by at least $\omega(\sqrt{n} \log n)$. In [34, 25], four-state protocols are analyzed that stabilize in expectation in $O(n^2 \log n)$ interactions. In a recent series of papers [35, 4, 1, 3, 19, 15, 16, 13], bounds for the Majority problem have been gradually improved. The currently best known protocol [13] solves Majority w.h.p. in $O(n \log^{3/2} n)$ interactions using $O(\log n)$ states. Regarding lower bounds, [4] shows that protocols with less than $(\log \log n)/2$ states require in expectation $(n^2/\operatorname{polylog}(n))$ interactions to stabilize. In [1] it is shown that any Majority protocol that stabilizes in $n^{2-(1)}$ expected interactions requires $(\log n)$ states under some natural assumptions.

The goal for Leader Election protocols is that exactly one agent is in a designated leader state. Doty and Soloveichik [24] show that any population protocol with a constant number of states that stably elects a leader requires (n^2) expected interactions, a bound matched by a

natural two-state protocol. Upper bounds for protocols with a non-constant number of states per agent were presented in [2, 4, 19, 1, 17, 28, 29, 14]. In [28] a Leader Election protocol that stabilizes w.h.p. in $O(n \log^2 n)$ interactions, using $O(\log \log n)$ states (matching a corresponding lower bound [4]) is presented. The core idea is to synchronize the agents using a *phase-clock*. The currently best known protocol for Leader Election is due to [14], stabilizing in expected $O(n \log n)$ interactions using $O(\log \log n)$ states per agent.

As a tool for self-synchronization, so-called *phase-clocks* have been explored in a wide range of related areas, see, e.g., the seminal paper [10]. In the population model, the concept of phase-clocks was first introduced in [6] under the assumption that a leader is present. These clocks were generalized in [28] to a *junta* of n^{ε} agents. In section 4.7 we empirically analyze a variant of this phase-clock process.

Simulations have proven a versatile tool to get an intuitive understanding of population protocols. This is also reflected in the related work: See, e.g., [6, 5, 3, 2] for some examples of papers that also present empirical data. However, to the best of our knowledge, our paper is the first systematic analysis of simulators for population protocols.

Sampling from Discrete Distributions The methods to sample non-uniform variates heavily depend on the modeling and properties of the required probability distributions.

If the distribution is governed by a closed-form density function $f(x)$, "numerical tricks" can yield efficient algorithms with small memory footprints; this is the case for most well-known distributions [23, 20]. A standard approach is the *inverse sampling technique* [23]. It needs to compute the inverse F^{-1} of f's cumulative density function $F(x) = \int_{-\infty}^{x} f(y)\mathrm{d}y$ (cf. section 4.6.1). Another concept is *rejection sampling* [20, Sec. 5.2.5/6]. It obtains a sample x from a suitable simpler distribution g. In order to generate the distribution f, the sample is accepted only with probability proportional to $f(x)/g(x)$. The process repeats until a sample is obtained (cf. section 4.3). A special case is the *ratio-of-uniforms* method, commonly used to obtain hypergeometric random variates [39].

Dedicated data structures support sampling from arbitrary discrete distributions. Given a finite universe $U = \{1, \ldots, u\}$ with probabilities $(p_1, \ldots, p_u)$ with $\sum_i p_i = 1$, Walker's *alias tables* [41] allow sampling in constant expected time, and can be constructed in $O(u)$ time [40]; recently Hübschle-Schneider et al. [31] discussed engineering aspects and parallel construction.

In this paper, we model an urn with n balls with $O(\sqrt{n})$ colors (typically much less) as a distribution over k colors where p_i is proportional to the number of balls with color i. While alias tables work in the static case, they do neither support removal nor insertion of balls without rebuilding the data structure.

A suited binary tree can be constructed in $O(k)$ time, and supports updates and sampling in $O(\log k)$ time. Two independent but similar data structures by [30] and [33] support updates and sampling in expected constant time. They partition the input into groups such that the probabilities of elements within a group differ by at most a factor of two. This allows rejection sampling *within* a group with an acceptance rate of at least $1/2$. Since updates to the distribution can affect the partition sizes, over-provisioning and table doubling involving garbage collection [33, App. B] is used.

The approaches differ in the way they select a group to sample from. They are however both

Algorithm 4 SEQ: The algorithmic framework for sequential simulation.

Require: configuration C, transition function δ, number of steps N
  for $t = 1$ to N
    sample and remove agents i and j without replacement from C
    add agents in states $\delta(s_i, s_j)$ to C

recursive in the sense that the group selection is carried out with another instance of the data structure itself. To achieve constant access times, the recursion is stopped after constantly many layers, and the remaining very small problem is treated as a special case. As a result, the data structures are quite complex, and were excluded in preliminary experiments due to performance considerations. By constraining the supported distributions, simpler schemes can be obtained [36, 27]. However, in our use case, they incur impractically high rejection rates, as we cannot give non-trivial bounds on the number of balls per color.

It is note-worthy that Hagerup et al. [30] operate on integer weights (modeled as *generalized distributions* lacking the normalization constraint), and require integer arithmetic only.

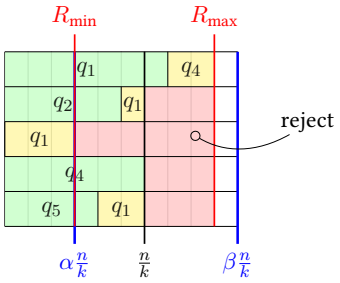
4.2 Sequential Simulation

As a baseline, we first consider variants of SEQ, a sequential approach defined in section 4.2. It is a direct translation of the machine model discussed in section 4.1.1. SEQ carries out N steps in a fully serialized manner. For each interaction, it selects two agents uniformly at random, computes their new states based on their current ones, and updates the configuration.

Under the realistic assumption that the transition function δ can be evaluated in constant time, SEQ's runtime and memory footprint is dominated by storing, sampling from, and updating the configuration C. We therefore consider appropriate data structures. In the population model, agents typically are anonymous, i.e., we cannot distinguish two agents in the same state. Hence, we can store a configuration C as an unordered multiset $\hat{C}$ and maintain multiplicities rather than individual states. To this end, SEQ requires an *urn-like* data structure which efficiently supports (i) weighted sampling (with and without replacement) and (ii) adding of single agents. In the following, we consider various data structures and their impact on the complexity of the sequential approach.

Array $\text{SEQ}_{\text{Array}}$ maintains the configuration C in an array $A[1 \dots n]$ where $A[i]$ holds s_i, the state of the i-th agent. Sampling with replacement is trivial, as we only draw a uniform variate $X \in [n]$ and return $A[X]$. Sampling without replacement works analogously: we overwrite $A[i]$ with $A[n]$ and remove the array's last element $A[n]$. Adding new elements is possible by appending. (Note that we do not grow the memory since we always store at most n agents in the array.) This leads to an $O(N)$ time algorithm and a memory footprint of $O(n \log |Q|)$ bits, which can be prohibitively large if simulating large populations in parallel.

Linear Search $\text{SEQ}_{\text{Linear}}$ maintains the multiset $\hat{C}$ in an array A such that $A[i]$ holds the number of agents in state q_i. Sampling requires a linear search on A in $O(|Q|)$ per sample.



$F[i]$	$S[i]$	$A[i]$	rejection probability
7	2	q_4	0
5	1	q_1	$3/9$
0	3	q_1	$6/9$
6	0	q_1	$3/9$
4	2	q_1	$3/9$

Figure 4.1: Dynamic Alias Table storing $\hat{C} = (q_1 : 13,\ q_2 : 5,\ q_3 : 0,\ q_4 : 8,\ q_5 : 4)$, i.e., $n = 30$ and $k = 5$. This imbalanced configuration will soon need rebuilding, e.g., after the next decrease of q_1 in row 3, or after adding two more agents in state q_1 in row 1.

This results in a worst-case simulation time of $\Theta(N|Q|)$. Nevertheless, in practice $\textsc{SeqQ}_{\textsf{Linear}}$ is among the fastest sequential variants for small $|Q|$ (see section 4.7). Compared to $\textsc{SeqQ}_{\textsf{Array}}$, it has a significantly smaller memory footprint of $\mathrm{O}(|Q| \log n)$ bits.

Binary Search Tree $\textsc{SeqQ}_{\textsf{BST}}$ maintains the multiset $\hat{C}$ using a balanced binary search tree. The i-th leaf (from left to right) encodes $\hat{C}_i$, the number of agents in state i. Each inner node v stores the number ℓ_v of agents in its left subtree. To randomly sample an agent, we draw an integer X from $\{0, \dots, n-1\}$ uniformly at random and compare it to the root's value ℓ_r. If $X < \ell_r$, the sample is in the interval covered by the left sub-tree, and we descend accordingly. Otherwise, we update $X \leftarrow X - \ell_r$ and descend into the right subtree. We recurse until some leaf i is reached, where we emit an agent of state i.

Each operation on the tree involves a simple path from the root to a leaf of length $\Theta(\log |Q|)$. Since the work per level is constant, all operations take $\Theta(\log |Q|)$ time. Thus, $\textsc{SeqQ}_{\textsf{BST}}$ requires $\Theta(N \log |Q|)$ total time and $\mathrm{O}(|Q| \log n)$ bits of memory.

$\textsc{SeqQ}_{\textsf{Alias}}$ combines the linear runtime of $\textsc{SeqQ}_{\textsf{Array}}$ (w.h.p.) with the small memory footprint of $\textsc{SeqQ}_{\textsf{BST}}$, provided $|Q| < \sqrt{n}$. At the heart of $\textsc{SeqQ}_{\textsf{Alias}}$ lies a *Dynamic Alias Table* introduced in the following section.

4.3 Dynamic Alias Tables

Dynamic Alias Tables In this section we introduce our Dynamic Alias Table data structure. Our goal is to model an urn which initially contains n marbles, each of which has one of k possible colors. We assume that the colors are identified by numbers in $\{1, \dots, k\}$. The urn defines a probability distribution D for the color of a marble drawn uniformly at random: let p_i be the probability that we sample a marble of color i.

Original Alias Method The alias method [41] uses a table with two columns and k rows, one row for each element (color) in the distribution D. Each row i has two entries corresponding to two elements. Each element has a weight in $[0, 1]$, and the two weights sum up to 1 in each row. The first element of row i is always element i. It has weight $F[i]$. The second element of

row i is stored in $A[i]$. It has a weight of $1 - F[i]$. This means that the original alias method uses only the two arrays, F and A, to store the distribution $p_1, \ldots, p_k$.

To sample from D in the original alias method, we first sample a row i uniformly at random from $\{1, \ldots, k\}$. Then we draw a random real $X \in [0, 1)$. If $X < F[i]$, we return the left element, i. Otherwise, we return the right element, $A[i]$. In the following, we modify the alias method and call the resulting data structure Dynamic Alias Table.

Dynamic Alias Tables Recall that we assume that our distribution corresponds to an urn storing n marbles of k different colors. First, we explicitly add a second weight array $S[i]$ which stores the weight of the alias. Now instead of storing just one real value $F[i]$ for each row i, we store the exact numbers of marbles as integers for the first and the second entry of row i in $F[i]$ and $S[i]$, respectively (cf. generalized distributions of [30]). As before, the first entry of row i corresponds to color i and the second entry of row i corresponds to color $A[i]$. As illustrated in fig. 4.1, the rows are constructed in such a way that the total weight of each row no longer adds up to the real value 1, but to the integer value $\lfloor n/k \rfloor$ or $\lceil n/k \rceil$ such that all rows in total add up to n.

Observation 1. *Let U be a Dynamic Alias Table encoding an urn with n marbles and k colors. The data structure U can be constructed in $\mathrm{O}(k)$ time.*

Proof. The algorithm by Vose [40] can generate an (original) alias table representation of such a discrete probability distribution D in $\mathrm{O}(k)$ time. It is straightforward to define a mapping between the weights of the original alias method as computed in [40] and the two integer values used in our Dynamic Alias Table. It follows that the Dynamic Alias Table (using integer weights) can be constructed in $\mathrm{O}(k)$ time. □

Updating and Sampling from Dynamic Alias Tables Our modified data structure now allows us to sample elements with and without replacement. Let $R_i = F[i] + S[i]$ denote the weight of row i and define $R_{\min}$ and $R_{\max}$ as smallest and largest row weights, respectively. Observe that in general $R_{\min} \neq R_{\max}$.

In order to sample from U, we first select row i uniformly at random from $\{1, \ldots, k\}$. Then we draw a uniform variate X from $\{0, \ldots, R_{\max} - 1\}$.[1] There are three possible events: If $X < F[i]$, we emit the first element i. If $F[i] \leq X < R_i$, we emit the second element $A[i]$. Otherwise, we reject the trial and restart the sampling process.

If we sample from U without replacement, we decrement the weight of the element just sampled. This is always possible, since only elements with strictly positive weights can be sampled in the first place. If we add a new element with color i to U, we increment the weight of the first element of row i.

In order to guarantee expected constant sampling time, we ensure that the fraction between $R_{\min}$ and $R_{\max}$ does not exceed a certain value. Let $0 < \alpha < 1$ and $\beta > 1$ be two parameters chosen such that $\beta/\alpha = \mathrm{O}(1)$. After each update to U we require

$$\alpha \lfloor n/k \rfloor \leq R_{\min} \leq R_{max} \leq \beta \lceil n/k \rceil. \tag{4.1}$$

[1] Observe that in a practical implementation one can draw a single uniform variate U' from $\{1, \ldots, k \cdot R_{\max} - 1\}$, and derive U and X from it.

Otherwise, we rebuild the data structure in $O(k)$ time.

We are now ready to show theorem 4.

Proof of theorem 4. We start with the memory complexity. The Dynamic Alias Table U stores the values of k, n, and R_{max} as well as three arrays. Array $F[1 \dots k]$ stores the weight of the first column, array $S[1 \dots k]$ stores the weight of the second column, and array $A[1 \dots k]$ stores the alias, i.e., the element of the second column. All entries are integers from $\{0, \dots, n\}$ (recall that we assume $k \leq \sqrt{n}$). Thus, the Dynamic Alias Table requires $\Theta(k \log n)$ bits of memory.

Let us now consider the sampling procedure. First, we consider the rejection probability. Recall that we first sample a row i and then draw a uniform variate X from $\{0, \dots, R_{\text{max}} - 1\}$. As before, we denote the total weight of row i as R_i with $R_i = F[i] + S[i]$. A sampling trial in row i is rejected if $X \geq R_i$, i.e., with probability R_i / R_{max}. Therefore, the probability to reject a sample from any row is at most $R_{\text{min}}/R_{\text{max}}$. From the conditions in eq. (4.1) we get that the rejection probability is at most $\alpha/\beta = O(1)$ and, conversely, we have at least a constant success probability of $(\beta - \alpha)/\beta$. The number of trials until we emit an element is therefore geometrically distributed and has an expected value of at most $\beta/(\beta - \alpha) = O(1)$.

It remains to show that we emit an element of color i with probability $p_i = \hat{C}_i/n$, where $\hat{C}_i$ is the number of marbles of color i in the Dynamic Alias Table U. We consider a single sampling trial. Observe that in each trial we are given a uniform probability space $= \{(i, x) : 1 \leq i \leq k \text{ and } 0 \leq x < R_{\text{max}}\}$. From this probability space we draw the row i and the value X uniformly at random. Fix a color c and let $\mathcal{S}_c$ be the set of all events (i, x) which lead to emission of an element of color c for this probability space . An event (i, x) is in $\mathcal{S}_c$ if and only if (i) $i = c$ and $x < F[i]$ or (ii) $A[i] = c$ and $F[i] \leq x < F[i] + S[i]$.

The Dynamic Alias Table U is constructed such that the total weight for each color c always equals $\hat{C}_c$. Therefore, counting all elementary events gives us $|\mathcal{S}_c| = \hat{C}_c$. Observe that is a uniform probability space since the row i and the value X are drawn uniformly. It has size $|| = kR_{\text{max}}$. Hence, all events in $\mathcal{S}_c$ have equal probability $1/(kR_{\text{max}})$, and we get $\Pr[\mathcal{S}_c] = |\mathcal{S}_c|/(kR_{\text{max}}) = \hat{C}_c/(kR_{\text{max}})$.

Let $\mathcal{R}$ be the event that a trial is rejected. Analogously to before, we enumerate over all elementary events and obtain $|\mathcal{R}| = \sum_i (R_{\text{max}} - R_i) = kR_{\text{max}} - n$. For the complementary event $\overline{\mathcal{R}}$ we get $|\overline{\mathcal{R}}| = n$, which matches the intuition that the urn contains n marbles. Hence, we have $\Pr[\overline{\mathcal{R}}] = n/(kR_{\text{max}})$. Observe that $\mathcal{S}_c$ and $\mathcal{R}$ are mutually exclusive and hence $\mathcal{S}_c \cap \overline{\mathcal{R}} = \mathcal{S}_c \setminus \mathcal{R} = \mathcal{S}_c$.

Rejected trials emit no element, but are repeated. Hence, we condition on $\overline{\mathcal{R}}$ and obtain

$$p_c = \Pr\left[\mathcal{S}_c | \overline{\mathcal{R}}\right] = \frac{\Pr\left[\mathcal{S}_c \cap \overline{\mathcal{R}}\right]}{\Pr\left[\overline{\mathcal{R}}\right]} = \frac{\Pr\left[S_c\right]}{\Pr\left[\overline{\mathcal{R}}\right]} = \frac{\hat{C}_c}{kR_{\text{max}}} \cdot \frac{kR_{\text{max}}}{n} = \frac{\hat{C}_c}{n}.$$

This means that a color c is indeed emitted with the correct probability $p_c = \hat{C}_c/n$.

Finally, we consider the amortized costs of rebuilding the Dynamic Alias Table U ever so often. Observe that U has to be rebuilt whenever the condition in eq. (4.1) is violated. This can happen in two possible ways.

- Case 1: $R_{\text{min}} < \alpha \lfloor n/k \rfloor$.

In this case there must exist a row i for which $R_i < \alpha\lfloor n/k \rfloor$. Observe that by assumption of the theorem we have $n \geq k^2$, and after rebuilding U we have $R_{\min} = \lfloor n/k \rfloor$. In order to have $R_i < \alpha\lfloor n/k \rfloor$, at least $\lfloor n/k \rfloor - \alpha\lfloor n/k \rfloor \geq k(1-\alpha)$ elements must have been deleted from row i. As $0 < \alpha < 1$, this happens only after at least $k(1-\alpha) = \Omega(k)$ sampling operations. Now according to observation 1, rebuilding takes time $\mathrm{O}(k)$. Together this implies that rebuilding U takes amortized constant time per update of U.

- Case 2: $R_{\max} > \beta\lceil n/k \rceil$.

  The second case follows analogously to the first case. If $R_{\max} > \beta\lceil n/k \rceil$, at least $\beta\lceil n/k \rceil - \lceil n/k \rceil \geq k(\beta - 1)$ elements must have been added. This takes at least $k(\beta-1) = \Omega(k)$ insertions, and hence rebuilding U takes amortized constant time per insertion.

Removal and insertion operations can be arbitrarily mixed and interact only beneficially towards the amortization arguments. This concludes the proof of the theorem. □

4.4 Batch Processing

So far, we discussed algorithms to simulate a population protocol step-by-step. These simulators can output the population's configuration $C(t)$ for each time step $1 \leq t \leq N$. With a time complexity of $\mathrm{O}(N)$, the simulators SeqArray and SeqAlias are optimal in this sense. In practice, however, it often suffices to obtain a configuration snapshot every $\Theta(n)$ steps. In this setting, we can achieve sub-constant work per interaction under mild assumptions.

Recall that SeqBST has a small memory footprint but a sub-optimal time complexity of $\Theta(N \log |Q|)$. Observe, however, that the underlying binary search tree can update the multiplicity of any existing state in time $\mathrm{O}(\log |Q|)$ *independently* of the changed quantity. Here, we introduce the new algorithm Batched (see algorithm 5) to exploit this observation. The algorithm uses a binary search tree to store the configuration. It updates $\Omega(\sqrt{n})$ agents in expectation with each access and therefore reduces the time complexity to $\mathrm{O}(N(\log n + |Q|^2)/\sqrt{n})$ which is $\mathrm{o}(N)$ for $|Q| = \mathrm{o}(n^{1/4})$ and $N = \Theta(\mathrm{poly}(n))$.

Batching interactions In order to coalesce individual updates into batches, Batched uses the notion of *collision-free runs* as illustrated in fig. 4.2. We interpret the execution of a protocol as a sequence $i_1, i_2, \ldots$ where at time t agents i_{2t-1} and i_{2t} interact.

Let ℓ be the largest index such that all $i_1, \ldots, i_\ell$ are distinct. Then, the first $\lfloor \ell/2 \rfloor$ interactions are independent of each other and can be rearranged in any order. We refer to them as a collision-free run of length ℓ. If ℓ is odd, the first agent of the $(\lfloor \ell/2 \rfloor + 1)$-th interaction is also considered collision-free. Since we are free to reorder the interactions, we can group all interactions of states (q_i, q_j) together, evaluate $\delta(q_i, q_j)$, and update all accordingly affected states in one step.

Now instead of sampling a sequence of agents and partitioning the sequence into collision-free runs, we take the opposite direction. We first sample only the length ℓ of a collision-free run from the appropriate probability distribution (see below). Then, we randomly match ℓ

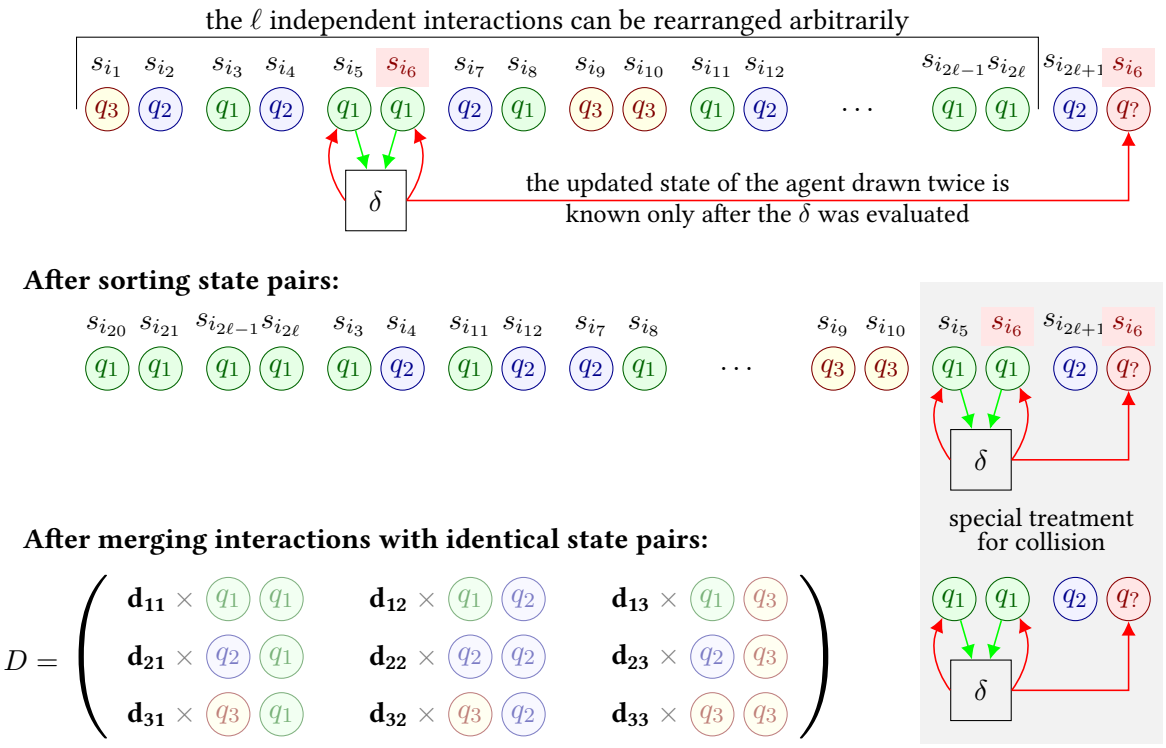


Figure 4.2: Batch processing uses collision-free runs, long sequences of independent interactions, which can be rearranged and grouped together.

Algorithm 5 BATCHED: The algorithmic framework for simulation in batches.

Require: configuration C, transition function δ, number of steps N
  $t \leftarrow 0$
  while $t < N$ $\ell \leftarrow$ sample length of a collision-free run
    let $D = (d_{ij})$ be a $|Q| \times |Q|$ matrix and sample d_{ij} as the number of interactions (q_i, q_j) among ℓ interactions ▷ batch processing
    let C' be an empty configuration
    for $(q_i, q_j) \in Q^2$
      remove from C: d_{ij} agents in states q_i, and d_{ij} agents in states q_j
      $(q_i', q_j') \leftarrow \delta(q_i, q_j)$
      add to C': d_{ij} agents in states q_i', and d_{ij} agents in states q_j'
    if ℓ is even ▷ plant a collision
      sample agent c_1 without replacement from C' ▷ collision at c_1
      merge C' into C
      sample agent c_2 without replacement from C
    else
      sample agent c_1 without replacement from C
      sample agent c_2 without replacement from C' ▷ collision at c_2
      merge C' into C
    add agents $\delta(c_1, c_2)$ to C $t \leftarrow t + \ell + 1$

agents as discussed below. Finally, we reuse one of the agents from the matching in order to plant a collision. These steps are repeated until at least N interactions are simulated.

Matching Agents We simulate sampling ℓ agents without replacement to construct a collision-free run of length ℓ. While we cannot afford to draw the agents individually, we only need to know how many interactions n_{ij} of each state pair (q_i, q_j) we encountered. Thus, a run can be modeled by a $|Q| \times |Q|$ matrix $D = (n_{ij})$ with $\sum_{ij} n_{ij} = \lfloor \ell/2 \rfloor$. (If ℓ is odd, we remove one agent and treat it individually.)

To obtain D, we first sample the row sums $D_i = \sum_j n_{ij}$ of the matrix from a multivariate hypergeometric distribution. This simulates sampling $\lfloor \ell/2 \rfloor$ initiating agents without replacement. We then sample values within each row analogously to find the matching responding agents. Sampling D takes $\mathrm{O}(|Q|^2)$ time in total since each individual sample from a hypergeometric distribution can be computed in $\mathrm{O}(1)$ time [39].

For correctness, note that our sampling approach corresponds to first selecting $\lfloor \ell/2 \rfloor$ agents as initiators and then $\lfloor \ell/2 \rfloor$ agents as responders. That is, we first sample agents $i_1, i_3, \ldots, i_{2\lfloor \ell/2 \rfloor - 1}$ and then agents $i_2, i_4, \ldots, i_{2\lfloor \ell/2 \rfloor}$ (instead of the natural interleaved variant $i_1, i_2, \ldots, i_{2\lfloor \ell/2 \rfloor}$). Since, each draw is taken uniformly at random, the permutation does not change the distribution (see full version [18] for a formal proof).

Length of a Collision-Free Run In the following, we analyze the length ℓ of a collision-free run. Observe that the following analysis is similar to the analysis of a generalized variant of

the birthday problem [32]. We consider a generalization which we also use in section 4.5. We assume that r agents have already interacted and ask how many more collision-free agents can be added. Formally we define the distribution $\mathsf{coll}(n, r)$ as follows.

Definition 1. *Consider a sequence $a_1, a_2, \ldots$ of agents sampled independently and uniformly at random. Let A_0 be a set of r initially prescribed agents and let $A_i = A_{i-1} \cup \{a_i\}$ be the set of agents after i draws. We define the random variable ℓ as the smallest index s.t. $a_\ell \in A_{\ell-1}$. We say $\ell \sim \mathsf{coll}(n, r)$, where n is the total number of agents and r is the number of prescribed agents.*

In section 4.6.1 we discuss how we can sample from this distribution using the inverse sampling technique [23]. We find that sampling ℓ takes $\mathrm{O}(\log n)$ time. In practice, this is comparable to the time it takes to sample a hypergeometric random variate. In the following, we show basic properties of $\mathsf{coll}(n, r)$ in order to show bounds on the runtime of Batched.

Lemma 4. *Let $\ell \sim \mathsf{coll}(n, r)$. Then ℓ has support $\{1, \ldots, n - r\}$ and distribution* $\Pr[\ell = k] = n^{-(k+1)}(n - r)!(r + k)/(n - r - k)!$.

Proof. Consider an urn with n marbles. Initially, r marbles are red, while the remaining $n - r$ marbles are green. We now take out one marble at a time: if it is green, we keep on going (think of a traffic light) and put a red one back in. If we take a red marble, we stop. Observe that the number of marbles we take out is exactly ℓ as above, as the green marbles represent new unconsidered agents while the red ones represent agents in $A_{\ell-1}$.
This directly leads to the acclaimed distribution:

$$\Pr[\ell = k] \quad = \quad \underbrace{\prod_{i=0}^{k-1} \frac{(n-r)-i}{n}}_{\text{select } k \text{ out of } n-r} \quad \cdot \quad \underbrace{\frac{r+k}{n}}_{(k+1)\text{-th is red}}$$

□

Lemma 5. *Let $\ell \sim \mathsf{coll}(n, 0)$. Then $\mathbb{E}[\ell] = \Theta(\sqrt{n})$.*

Proof. We first upper bound $\mathbb{E}[\ell] = \mathrm{O}(\sqrt{n})$ and then give a matching lower bound $\mathbb{E}[\ell] = \Omega(\sqrt{n})$. In both cases, we write $\mathbb{E}[\ell] = \sum_{i=0}^{n} \Pr[\ell \geq i]$ and split the sum at $\sqrt{n}$. Then we bound both terms appropriately. Observe that for some fixed value i we have $\Pr[\ell \geq i] = \prod_{j=0}^{i-1}(1 - j/n)$. For the upper bound on $\mathbb{E}[\ell]$ we get

$$\mathbb{E}[\ell] = \sum_{i=0}^{n} \Pr[\ell \geq i] = \sum_{i=0}^{n} \prod_{j=0}^{i-1} \left(1 - \frac{j}{n}\right) \leq \sum_{i=0}^{\sqrt{n}-1} 1 + \sum_{i=\sqrt{n}}^{\infty} \left(1 - \frac{\sqrt{n}}{n}\right)^i \leq 2\sqrt{n}.$$

Similarly, we get for the lower bound on $\mathbb{E}[\ell]$ that

$$\begin{aligned}\mathbb{E}[\ell] = \sum_{i=0}^{n} \Pr[\ell \geq i] &= \sum_{i=0}^{n} \prod_{j=0}^{i-1} \left(1 - \frac{j}{n}\right) \geq \sum_{i=0}^{\sqrt{n}} \prod_{j=0}^{i-1} \left(1 - \frac{\sqrt{n}}{n}\right) = \sum_{i=0}^{\sqrt{n}} \left(1 - \frac{1}{\sqrt{n}}\right)^i \\ &= \sqrt{n}\left(1 - \left(1 - \frac{1}{\sqrt{n}}\right)^{\sqrt{n}+1}\right) \geq \sqrt{n}\left(1 - e^{-1}\right).\end{aligned}$$

Therefore we have $\mathbb{E}[\ell] = \Theta(\sqrt{n})$. □

Using lemma 5, we are now ready to bound the runtime and space complexity of BATCHED.

Theorem 5. *Let n be the number of agents and $|Q|$ the number of states. BATCHED simulates N interactions in $O(N(|Q|^2 + \log n)/\sqrt{n})$ expected time using $\Theta(|Q| \log n)$ bits.*

Proof. According to lemma 5, each batch simulates $\Theta(\sqrt{n})$ interactions in expectation. It takes $O(\log n)$ time to sample the length of a collision-free run ℓ (see section 4.6.1) and $O(|Q|^2)$ time (cf. [39]) to sample the interaction numbers and process the interactions for all pairs of states. This implies the runtime complexity. The space complexity follows immediately from the binary search tree used to store the configuration. □

4.5 Merging Batches

In an empirical evaluation, we found that our implementation of algorithm BATCHED spends most time in the batch processing step (to sample and transition the $|Q| \times |Q|$ matrix D); this is especially true for complex protocols with non-trivial state space sizes. As the matrix sampling cost is independent of the length ℓ of the underlying collision-free run, we modify the algorithm to support more than one collision per batch processing step.

Introducing Epochs An execution of the improved algorithm MULTIBATCHED logically consists of several epochs. For each epoch, the algorithm samples the *lengths* ℓ_1, ℓ_2, ..., ℓ_ρ of multiple collision-free runs $R_1, \ldots, R_\rho$. As no agent may appear twice in the union of those collision-free sequences, later runs become shorter in expectation ($\mathbb{E}[\ell_{i+1}] < \mathbb{E}[\ell_i]$), naturally limiting the number ρ of runs per epoch. After each run R_i, we plant one collision, i.e., an interaction with an agent that was already considered in the current epoch. An epoch concludes with a single batch processing step, in which matrix D is sampled and processed analogously to algorithm BATCHED.

Tracking Dependencies While algorithm BATCHED only reorders and groups together independent interactions, our improved algorithm MULTIBATCHED delays most interactions until the end of the epoch. To do so, the algorithm conceptually assigns each agent one of three types, and updates these labels as it progresses through the epoch:

- untouched agents did not interact in the current epoch. Hence, all agents are labeled untouched at the beginning of an epoch.
- updated agents took part in at least one interaction that was already evaluated. Thus, updated agents are already assigned their most recent state.
- delayed agents took part in *exactly one* interaction that was not yet evaluated. Thus, delayed agents are still in the same state they had at the beginning of the epoch, but are scheduled to interact at a later point in time. We additionally require that their interaction partner is also labeled delayed.

Analogously to algorithm BATCHED, we maintain two urns C and C'. Urn C' contains updated agents, while urn C stores untouched and delayed agents (or in other words, all agents whose state was not updated in the current epoch). At any point in time, an agent is either in C or C' meaning that $|C| + |C'| = n$. Due to symmetry, we do not explicitly differentiate untouched from delayed agents. We rather maintain only the number T of delayed agents and lazily select them while planting collisions or during batch processing.

If a delayed agent a is selected while planting a collision, it takes part in a second interaction and —by definition— cannot be labeled delayed any more. Thus, we randomly draw a second delayed agent b, evaluate their transition, store the updated state of b in C', and directly evaluate a again in the planted collision. Finally, we decrease $T \leftarrow T - 2$ as agents a and b changed their labels from delayed to updated. Observe that we might repeat this step in the (unlikely) case that a planted collision involved two formerly delayed agents.

Length of an Epoch We now analyze the length of an epoch. We start by extending the analysis of coll (n, r) to the $r = \Omega(\sqrt{n})$ regime (reached after $\mathrm{O}(1)$ runs w.h.p.). The following lemmas establish expected value and concentration.

Lemma 6. *Let $\ell \sim \mathsf{coll}\,(n, r)$ and $r = \Omega(\sqrt{n})$. Then $\mathbb{E}\,[\ell] = \Theta(n/r)$.*

Proof. The proof follows analogously to lemma 5. Again, we start with the upper bound.

$$\mathbb{E}\,[\ell] = \sum_{i=0}^{n-r} \Pr[\ell \geq i] = \sum_{i=0}^{n-r} \prod_{j=0}^{i-1} \left(1 - \frac{j+r}{n}\right) \leq \sum_{i=0}^{\infty} \left(1 - \frac{r}{n}\right)^i = \frac{n}{r}.$$

For the lower bound we derive a general result for arbitrary r.

$$\begin{aligned}\mathbb{E}\,[\ell] &= \sum_{i=0}^{n-r} \Pr[\ell \geq i] = \sum_{i=0}^{n-r} \prod_{j=0}^{i-1} \left(1 - \frac{j+r}{n}\right) \geq \sum_{i=0}^{r-1} \prod_{j=0}^{i-1} \left(1 - \frac{j+r}{n}\right) \geq \sum_{i=0}^{r-1} \left(1 - \frac{2r}{n}\right)^i \\ &= \frac{n}{2r}\left(1 - \left(1 - \frac{2r}{n}\right)^r\right) \geq \frac{n}{2r}(1 - e^{-2r^2/n}).\end{aligned}$$

The last inequality holds since $e^{-2r^2/n}$ constitutes an upper bound on $(1 - 2r/n)^r$ as it can be rewritten as $(1 - 2r/n)^{n \cdot r/n}$ and $(1 - 2r/n)^n \leq e^{-2r}$. For $r = \Omega(\sqrt{n})$ the second factor $(1 - \exp(-2r^2/n))$ is $\Omega(1)$ which proves the claim. □

Lemma 7. *Let $\ell \sim \mathsf{coll}\,(n, r)$ and $r = \Omega(\sqrt{n})$. Then $\ell = \Theta(n/r)$ with probability $1 - o(1)$.*

Proof. We prove the claim by showing that $\Pr[\ell < t]$ and $\Pr[\ell > t]$ are $o(1)$ for $t = o(n/r)$ and $t = \omega(n/r)$, respectively. We have

$$\Pr[\ell < t] = 1 - \Pr[\ell \geq t] = 1 - \prod_{i=0}^{t-1} \left(1 - \frac{i+r}{n}\right) \leq 1 - \left(1 - \sum_{i=0}^{t-1} \frac{i+r}{n}\right) \leq \frac{2tr + t^2}{2n},$$

where the first inequality follows from the Weierstrass product inequality. Furthermore, with $r = \Omega(\sqrt{n})$ we have $n/r = O(\sqrt{n})$ and thus $t = o(n/r)$ such that $t^2 = o(n)$ and $tr = o(n)$. For the second part, we have

$$\Pr[\ell > t] = \prod_{i=0}^{t} \left(1 - \frac{i+r}{n}\right) \leq \left(1 - \frac{r}{n}\right)^t \leq e^{-rt/n},$$

and for $t = \omega(n/r)$ and thus $rt/n = \omega(1)$ the claim follows. □

Intuitively, lemma 6 shows that for sufficiently many prescribed agents r, the probability of drawing a colliding agent remains approximately r/n throughout the run. Similar to a geometric distribution, this results in a concentrated expected length of $\Theta(n/r)$. We now estimate the number of agents sampled after ρ runs.

Lemma 8. *Let $L_k = \sum_{i=1}^{k} \ell_i$ be the number of agents drawn in an epoch with k runs. Then, for $|Q| = \omega(\sqrt{\log n})$, $|Q| = o(\sqrt{n \log n})$ and $\rho = \mathrm{O}(|Q|^2 / \log n)$ we have $\mathbb{E}[L_\rho] = \Theta(\sqrt{\rho n})$.*

Proof. The variable L_k equivalently corresponds to the number of marbles $B(k, n)$ that need to be drawn in the birthday problem s.t. k coincidences occur. The asymptotics of $\mathbb{E}[B(k, n)]$ have first been studied by Kuhn and Struik [32] for the cases that $k = o(n^{1/4})$. Their results have since been improved by Arratia et al. [11] where the asymptotic bounds on the moments of $B(k, n)$ have been calculated for more general conditions on k. By [11, Corollary 12] for k a function of n, i.e., $k = k_n$ where $k_n \to \infty$ and $k_n/n \to 0$ it holds that

$$\mathbb{E}[B(k_n, n)] \sim \sqrt{2nk_n} \quad \text{as } n \to \infty.$$

By assumption the conditions are met since $\rho = \omega(1)$ and $\rho = o(n)$, thus $\mathbb{E}[L_\rho] = \Theta(|Q|\sqrt{n/\log n}) = \Theta(\sqrt{\rho n})$. □

Complexity In order to analyze MultiBatched's runtime, we first establish the time required per epoch, and then bound the total expected runtime and memory requirements.

Lemma 9. *MultiBatched takes time $\mathrm{O}(\rho \log n + |Q|^2)$ for an epoch of ρ runs.*

Proof. Planting a collision is done by drawing the two interacting agents from the appropriate urns and setting them to be updated which requires $\mathrm{O}(1)$ time. Sampling the length of a single collision-free run takes time $\mathrm{O}(\log n)$ (see section 4.6.1). For the final batch-processing step MultiBatched takes $\Theta(|Q|^2)$ time independently of the number of delayed agents. □

Theorem 6. *Let n be the number of agents and $|Q|$ the number of states. MultiBatched simulates N interactions in $\mathrm{O}(N|Q|/\sqrt{n/\log n})$ expected time if $|Q| = \omega(\sqrt{\log n})$ and $|Q| = \mathrm{o}(\sqrt{n \log n})$.*

Proof. Combining lemmas 8 and 9, we find a runtime of $\mathrm{O}(N(\rho \log n + |Q|^2)/\sqrt{\rho n})$. Setting $\rho = \Theta(|Q|^2 / \log n)$ balances the cost of sampling runs and planting collisions with the cost of batch processing, and thus does not increase the asymptotic cost per epoch. Higher values of ρ only increase the expected time complexity. □

MultiBatched has sub-constant work per interaction for $|Q| = \mathrm{o}\left(\sqrt{\frac{n}{\log n}}\right)$ and $N = \Theta(\mathrm{poly} n)$.

4.6 Heuristics and Implementation Details

Implementations of all discussed simulators (including scripts to reproduce figures and numbers included in this paper) are freely available. In the following, we highlight important aspects necessary to obtain simulators that are both fast in practice and highly customizable.

All simulators are implemented in C++ and use compile-time specializations to implement specific protocols and experimental setups.

In contrast to pure deterministic functions, non-deterministic transition functions (possibly with side-effects) have to be informed about every interaction carried out.[2] To allow for batch processing, we cannot use the natural invocation order. Instead, we inform the protocol how often a state pair will interact within an epoch. It is then expected to assign all participating agents to the appropriate states. See the full version of this paper [18] for more details and listings.

4.6.1 Sampling the Length of a Collision-Free Run

Recall that Batched and MultiBatched repeatedly sample the length ℓ of a collision-free run. In the following we discuss how this sampling can be implemented using the inverse sampling technique (Ist) [23]: let $\mathrm{cdf}(x)$ be the cumulative density function of a target distribution. Then, Ist draws a uniform variate U from $[0;1]$, solves $U = \mathrm{cdf}(x)$ for x, and returns it as the sample. We denote the CDF of $\mathsf{coll}\,(n,k)$ as $F_{n,k}(t)$. Lemma 4 yields:

$$1 - F_{n,k}(t) = \Pr[\ell{>}t] = \prod_{i}^{t} \frac{n-k-i}{n} = \frac{1}{n^t} \frac{(n-k)!}{(n-k-t-1)!} \overset{(x-1)!=\Gamma(x)}{=} n^{-t} \frac{\Gamma(n-k+1)}{\Gamma(n-k-t)}.$$

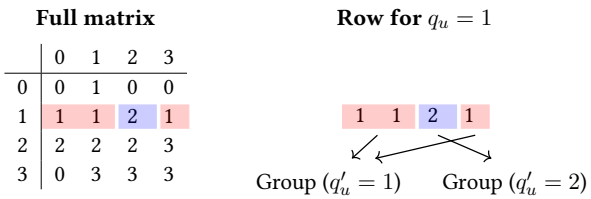
Since we are not aware of an inverse that can be evaluated fast, we numerically solve $U = F_{n,k}(t)$ for t. To avoid numerical instabilities, we rewrite the expression in terms of $\log\Gamma(x)$, which is available as the C standard function $\texttt{lgamma}(x)$:

$$U = 1{-}n^{-t}\frac{\Gamma(n-k+1)}{\Gamma(n-k-t)} \quad \Leftrightarrow \quad \log(1-U) = \log\Gamma(n-k+1){-}\log\Gamma(n-k-t){-}t\log n$$

Lacking a cheap derivative of the RHS, we rely on first-order numerical inversion methods only. In this context, an ad-hoc combination of binary search and regula-falsi gave most consistent results. We jump-start the search using a small look-up table containing lower and upper bounds on t for intervals of U and k.

While the method requires $\mathrm{O}(\log n)$ evaluations of $F_{n,k}(\cdot)$, we observe less than ten calls on average for $n = 2^{50}$. The resulting sampling algorithm has a practical runtime comparable to the sampling of hypergeometric random variates. Since the latter is sampled much more frequently, further optimizations will yield limited results to the total runtimes of Batched and MultiBatched.

[2] Observe that many protocols with non-deterministic transition functions have been derandomized, see, e.g., the notion of (biased) synthetic coins in [4, 17]. While this is supported by the simulator, we feel it is more convenient to offer the most expressive interface possible.



	0	1	2	3
0	0	1	0	0
1	1	1	2	1
2	2	2	2	3
3	0	3	3	3

Figure 4.3: Simplified transition matrix Δ' for a clock with period $m{=}4$. The initiator's phase (row) is circularly incremented only when matched with a suitable responder (column).

4.6.2 Heuristics

The frequent sampling of hypergeometric random variates, dominates MultiBatched's runtime. In the following, we discuss three heuristics to reduce this number.

Renaming In the *renaming heuristic* we exploit the observation that agents are typically not uniformly distributed over all states. Instead there are often sparsely populated states which are seldom hit when sampling an agent.

It can be beneficial to consider these states last. $\text{Seq}_{\text{Linear}}$'s linear search, for instance, stops as soon as the sampled state is found. The same is true when sampling Batched's and MultiBatched's interaction matrices: for row i, we draw D_i agents from a multivariate hypergeometric distribution. This is implemented by obtaining $|Q|{-}1$ properly parametrized hypergeometric variates; the process terminates early once all D_i agents have been sampled.

In both examples, we maximize the probability of early stopping by considering highly populated states first. To this end, we maintain a permutation $\pi\colon [|Q|] \to [|Q|]$ that sorts states decreasingly by their sizes. We then process states in the order indicated by π. If this permutation is updated once every $\Omega(|Q| \log |Q|)$ interactions, the sorting step becomes asymptotically negligible for sequential simulators. For Batched and MultiBatched, π can be updated once every $\Omega(\log q)$ batches.

Partitioning If δ is a deterministic function, we can model it as a matrix $\Delta \in (Q \times Q)^{|Q|\times|Q|}$. The matrix of a deterministic one-way protocol can be further simplified to $\Delta' \in Q^{|Q|\times|Q|}$ since the states of the responders remain unchanged.

For many meaningful protocols, the entries of Δ' are not random but exhibit some structure. As an example, consider the simplified[3] phase-clock transition matrix illustrated in fig. 4.3. Here, each row contains only two different output states. The *partitioning heuristic* uses this observation during the batch steps of Batched and MultiBatched. When sampling D_i responders for initiators in state q_i, we group together all entries in the i-th row of Δ' that assign the same new state to the initiating agents. It then suffices to draw one random hypergeometric variate per group.

[3]Formally the states of the phase-clock are pairs (x, b) where x represents the *phase*, and b marks an agent as *leader*. For the sake of simplicity we assume that all agents are followers.

Note that the heuristic does not reduce the runtime complexity of the algorithm — even if we precompute the partitioning. This is due the fact that we still need to compute the population sizes for each group which involves $\Theta(|Q|)$ additions per row. For pathological protocols, the number of hypergeometric random variates required for each row remains $\Omega(|Q|)$. A simple worst-case protocol is the transition function $\delta(q_u, q_v) = (q_v, q_v)$.

Skipping Generalizing *partitioning heuristic* to two-way protocols tends to be ineffective in practice: since initiator and responder may both update their states, the transition matrix is often more fragmented. The partitioning overhead then easily exceeds the potential savings. For such protocols MultiBatched uses a coarser partitioning, and only detects and skips transitions that preserve the configuration (i.e., $\delta(q_u, q_v) = (q_u, q_v)$ or $\delta(q_u, q_v) = (q_v, q_u)$).

4.6.3 Dynamic Epoch Lengths

Recall that MultiBatched is split into several epochs that each consist of multiple collision-free runs. In our implementation, we add runs to an epoch until the number of interactions exceeds a specified threshold T. The value of T has to be chosen as a trade-off between the cost of adding another run (i.e., sampling the run length and planting a collision) versus the diminishing return it yields (as later runs become shorter in expectation). This trade-off depends on the protocol and its configuration. For instance, the batch processing cost of a convergent protocol may become smaller compared to the initial costs (e.g., when most agents are in only a small fraction of the states).

As the trade-off is dynamic, we maximize the throughput using a control loop that dynamically optimizes the length of an epoch. Given the currently best value known for T, it increases (and later decreases) T to $1.1T$ and $0.9T$, respectively. For each of the three values, we measure the throughput, chose the T which maximizes it and repeat. Since the throughput response curve is single-peaked, the process will find a nearly optimal T.

4.7 Experimental Evaluation

In the following, we empirically evaluate the various simulation algorithms. The code is compiled using `g++-8.3` with flags `-O3 -march=native` and executed on the following system: 2× Intel Xeon Gold 6148 CPU @ 2.4 GHz (40 cores/80 hardware threads in total), 192 GiB DDR4 RAM @ 2666 MHz. Each data point is the median of at least five measurements (using different random seeds); error bars indicate their standard deviation.

$\text{Seq}_{\mathsf{BST}}$'s search tree is implemented as an array with breath-first-indexing (i.e., the weight of node $i \geq 1$ is stored at $A[i]$; its left child is at index $2i$, its right child at $2i+1$). In order to reduce pipeline stalls, we implement a branch-free traversal similarly to [37]. $\text{Seq}_{\mathsf{Array}}$ uses an array with 32 bit words to store states. We additionally consider $\text{Seq}^{\mathsf{prefetch}}_{\mathsf{Array}}$ which prefetches states for eight[4] interactions ahead of time as a latency hiding technique. If our Dynamic Alias Table implementation detects an imbalance necessitating rebuilding of the data structure, it will first attempt to quickly solve the problem by swapping the alias of the affected row with

[4] This is the optimal value measured for this CPU type and slightly varies between machines.

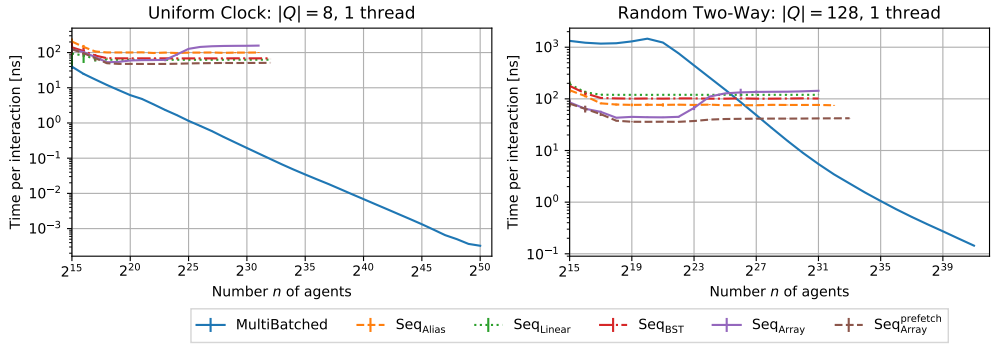


Figure 4.4: Processing time per interaction as function of the number of agents n. Each series ends with the largest value n for which the median of the total processing time is below 400 s.

up to five randomly selected partners; only if this fails a rebuild is issued. This heuristic does not change the asymptotic time complexity of the data structure.

The runtime of most simulators has non-trivial dependencies on the input parameters, protocol, and state distribution. Hence, we simulate a small number $N = n$ of interactions to prevent measuring artifacts caused by significant changes in the state distributions. MultiBatched typically simulates slightly more interactions due to the batching granularity. Since the runtime of all simulators is linear in N, we always report the time *per interaction* to ease extrapolation.
Three different protocols are used to highlight certain aspects of the algorithms.

- *Uniform Clock* and *Running Clock* implement the same deterministic one-way protocol phase-clock protocol inspired by [28]. In the *running* variant, all agents start in the first phase, and $\sqrt{n}$ of them are marked. Due to the choice of parameters, we expect only one out of $\Theta(\sqrt{n})$ interactions to change states. Thus, even at the end of the benchmark, the population is still highly concentrated in the lowest phase. The *uniform* variant, in contrast, evenly distributes marked and unmarked agents over all phases. This results in a constant update probability per interaction.

- The *Random Two-Way* protocol uses a deterministic transition function $\delta(q_i, q_j) = d_{ij}$ where each d_{ij} is initially drawn independently and uniformly at random from Q. Initially agents are evenly distributed over all states.

To ensure the correctness of our implementations, we rely on unit tests (e.g., using a family of protocols that count the number of interactions of each agent). Additionally, we cross-reference the results of various algorithms. Since we cannot expect two simulators to yield the same set of interactions, we compare their simulated dynamics. We, for instance, monitor the number of interactions required until a Uniform Clock starts running.

Number of Agents We begin our experimental study by investigating the dependencies on the problem size n. To this end, we search for the largest number n of agents that a simulator

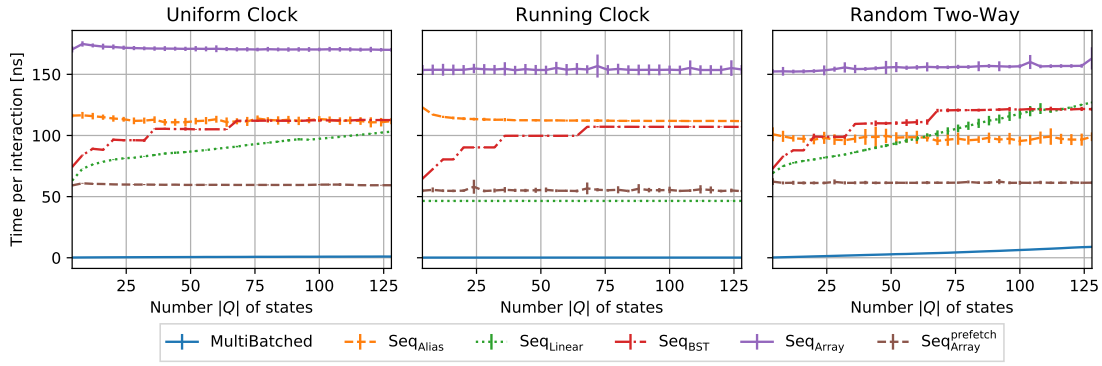


Figure 4.5: Processing time per interaction as function of the number of states $|Q|$ with $n = 2^{30}$.

can simulate within a fixed time budget of 400 s. Figure 4.4 reports such measurements for two different settings (see the full version of this paper [18] for the full set).

For the *Uniform Clock* protocol with $|Q| = 8$ states, the fastest Seq variant reaches $n = 2^{32}$ within the time budget. In the same time, MultiBatched simulates $N = n = 2^{50}$ interactions. For *Random Two-Way*, the ratio between the achievable population sizes is smaller but still exceeds three orders of magnitude. We attribute the different ratios mainly to the batching step. MultiBatched requires $\Theta(|Q|^2)$ hypergeometric variates per batching step for the latter protocol, since neither the partitioning nor the skipping heuristics (see section 4.6) are effective on a featureless transition matrix with uniformly random states. For the *Uniform Clock* protocol, the partitioning heuristic reduces the number of hypergeometric random variates to less than $2|Q|$ per batch.

Number of States As summarized in table 4.1, the number $|Q|$ of states crucially affects the algorithms' runtimes. To quantify the practical impact, we carry out scaling experiments for $4 \leq |Q| \leq 128$ while fixing all remaining parameters. Figure 4.5 visualizes the results.

MultiBatched performs best in all cases supporting our previous analysis. It shows almost no scaling behavior for both clock protocols. For *Random Two-Way*, the algorithm is almost one order of magnitude faster than its competitors despite a slow-down of 40 between the smallest and largest state sizes.

$\text{Seq}^{\text{prefetch}}_{\text{Array}}$ is the second fastest solution in almost all settings. In the *Running Clock* campaign, it is however outperformed by $\text{Seq}_{\text{Linear}}$. This can be explained by the fact that initially almost all agents are in state 0 which results in a constant time look-up despite the usage of a linear search. This behavior motivates the renaming heuristic.

We observe no systematic dependency on $|Q|$ for $\text{Seq}_{\text{Alias}}$ rendering it a good choice for very large state spaces. While it is up to a factor of 2.0 slower compared to $\text{Seq}^{\text{prefetch}}_{\text{Array}}$, the algorithm might be preferable in a parallel setting.

Memory Footprint and Parallelism Due to the stochastic nature of the protocols, we expect that in almost all applications several runs of the same protocol are required to derive statistically significant results. On modern machines with many processor cores, one should

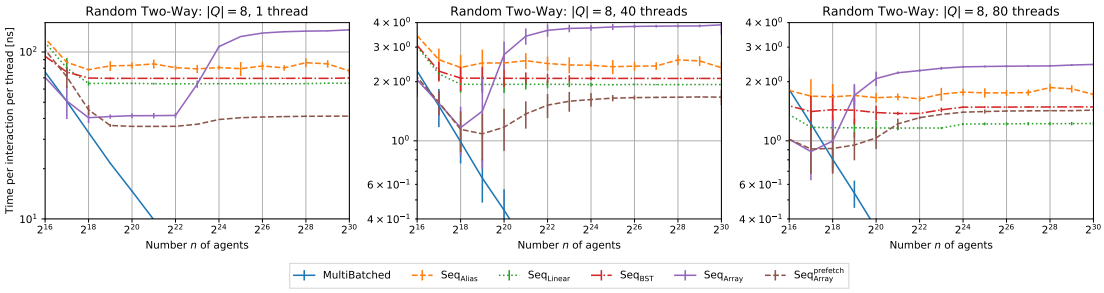


Figure 4.6: Effect of process parallelism on a machine with 40 CPU cores (plus HyperThreading).

be able to maximize the throughput by executing multiple independent simulations in parallel. As visualized in fig. 4.6, most simulators scale well with the number of threads and typically achieve a self-speedup of 40 to 50 times using 40 cores (plus HyperThreading) at $n = 2^{30}$. A notable exception is $\text{SEQ}^{\text{prefetch}}_{\text{Array}}$, which reaches only a speedup of 30 as it saturates the memory controllers of both CPU sockets.

Another aspect of parallel execution is the memory footprint. Since $\text{SEQ}_{\text{Array}}$ requires constant memory per agent, our implementations of $\text{SEQ}_{\text{Array}}$ and $\text{SEQ}^{\text{prefetch}}_{\text{Array}}$ allocate in excess of 320 GB main memory to execute 80 processes in parallel. Although, this number can be reduced by constant factors using a more efficient representation of states, it has to be contrasted to the competing algorithms with a state space of only a few kilobytes[5] for the same campaign.

4.8 Conclusions and Open Problems

We considered the simulation of large population protocols to allow the experimental investigation of slowly scaling observables in such systems. Two algorithm classes are discussed.

Sequential simulators carry out each interaction one after the other. We analyze the variants $\text{SEQ}_{\text{Array}}$, $\text{SEQ}_{\text{Linear}}$, SEQ_{BST}, and $\text{SEQ}_{\text{Alias}}$ which differ in the data structures maintaining the agents' states, and demonstrate substantial differences in their practical performances. As a by-product, we describe the Dynamic Alias Table which might be independently applicable.

Batched simulators coalesce interactions to achieve asymptotical speed-ups for protocols with a limited number of states. Our implementation then simulates more than 2^{50} interactions in 400 s which is several orders of magnitudes larger than the fastest sequential simulator.

Possible Extensions In some variants of the population model it is assumed that interactions are limited to some communication network. Since we store the configurations in our batched simulators as a multiset, it is not clear how to directly adapt our approach. We believe this might be an interesting extension of our work.

Further variants are concerned with the way how agents interact. It is straightforward to adapt our simulator software to a setting where, e.g., a random matching of agents interacts

[5] We report no exact numbers as the system's process overheads exceed the simulators' internal states.

in each time step. Furthermore, our approach could be generalized to a setting where more than two agents interact. In this case we are in need of good heuristics for partitioning the transition-tensor, since the work for updating a batch grows exponentially in the number of interacting agents. In general, sampling the interaction counters is a frequent and costly task. Therefore, any improved heuristic to sample from the $|Q| \times |Q|$ matrix using only $o(|Q|^2)$ variates would yield a measureable benefit for the total runtime of a simulation.

Bibliography

[1] Dan Alistarh, James Aspnes, and Rati Gelashvili. "Space-Optimal Majority in Population Protocols". In: *SODA*. SIAM, 2018. DOI: 10.1137/1.9781611975031.144.

[2] Dan Alistarh and Rati Gelashvili. "Polylogarithmic-Time Leader Election in Population Protocols". In: *ICALP (2)*. Vol. 9135. LNCS. Springer, 2015, pp. 479–491. DOI: 10.1007/978-3-662-47666-6_38.

[3] Dan Alistarh, Rati Gelashvili, and Milan Vojnovic. "Fast and Exact Majority in Population Protocols". In: *PODC*. ACM, 2015, pp. 47–56. DOI: 10.1145/2767386.2767429.

[4] Dan Alistarh et al. "Time-Space Trade-offs in Population Protocols". In: *SODA*. SIAM, 2017, pp. 2560–2579. DOI: 10.1137/1.9781611974782.169.

[5] Dana Angluin, James Aspnes, and David Eisenstat. "A simple population protocol for fast robust approximate majority". In: *Distributed Comput.* 21.2 (2008), pp. 87–102. DOI: 10.1007/s00446-008-0059-z.

[6] Dana Angluin, James Aspnes, and David Eisenstat. "Fast computation by population protocols with a leader". In: *Distributed Comput.* 21.3 (2008), pp. 183–199. DOI: 10.1007/s00446-008-0067-z.

[7] Dana Angluin, James Aspnes, and David Eisenstat. "Stably computable predicates are semilinear". In: *PODC*. ACM, 2006, pp. 292–299. DOI: 10.1145/1146381.1146425.

[8] Dana Angluin et al. "Computation in networks of passively mobile finite-state sensors". In: *Distributed Comput.* 18.4 (2006), pp. 235–253. DOI: 10.1007/s00446-005-0138-3.

[9] Dana Angluin et al. "The computational power of population protocols". In: *Distributed Comput.* 20.4 (2007), pp. 279–304. DOI: 10.1007/s00446-007-0040-2.

[10] Anish Arora, Shlomi Dolev, and Mohamed G. Gouda. "Maintaining Digital Clocks in Step". In: *Parallel Process. Lett.* 1 (1991), pp. 11–18.

[11] Richard Arratia, Skip Garibaldi, and Joe Kilian. "Asymptotic distribution for the birthday problem with multiple coincidences, via an embedding of the collision process". In: *Random Struct. Algorithms* 48.3 (2016), pp. 480–502. DOI: 10.1002/rsa.20591.

[12] James Aspnes and Eric Ruppert. "An Introduction to Population Protocols". In: *Bull. EATCS* 93 (2007), pp. 98–117.

[13] Stav Ben-Nun et al. "An $O(\log^{3/2} n)$ Parallel Time Population Protocol for Majority with $O(\log n)$ States". In: *PODC*. 2020, to appear.

[14] Petra Berenbrink, George Giakkoupis, and Peter Kling. "Optimal Time and Space Leader Election in Population Protocols". In: *STOC*. 2020, pp. 119–129.

[15] Petra Berenbrink et al. “A Population Protocol for Exact Majority with O(log5/3 n) Stabilization Time and Theta(log n) States”. In: *DISC*. Vol. 121. LIPIcs. Schloss Dagstuhl, 2018, 10:1–10:18. DOI: 10.4230/LIPIcs.DISC.2018.10.

[16] Petra Berenbrink et al. “Majority & Stabilization in Population Protocols”. In: *CoRR* abs/1805.04586 (2018). arXiv: 1805.04586.

[17] Petra Berenbrink et al. “Simple and Efficient Leader Election”. In: *SOSA@SODA*. Vol. 61. OASICS. Schloss Dagstuhl, 2018, 9:1–9:11. DOI: 10.4230/OASIcs.SOSA.2018.9.

[18] Petra Berenbrink et al. “Simulating Population Protocols in Sub-Constant Time per Interaction”. In: *CoRR* (2020). eprint: 2005.03584.

[19] Andreas Bilke et al. “Brief Announcement: Population Protocols for Leader Election and Exact Majority with $O(\log^2 n)$ States and $O(\log^2 n)$ Convergence Time”. In: *PODC*. ACM, 2017. DOI: 10.1145/3087801.3087858.

[20] Paul Bratley, Bennett L. Fox, and Linus Schrage. *A guide to simulation, 2nd Edition*. Springer, 1987.

[21] Luca Cardelli and Attila Csikász-Nagy. “The cell cycle switch computes approximate majority”. In: *Scientific reports* 2 (2012), p. 656. DOI: 10.1038/srep00656.

[22] Yuan-Jyue Chen et al. “Programmable chemical controllers made from DNA”. In: *Nature nanotechnology* 8.10 (2013), p. 755. DOI: 10.1038/nnano.2013.189.

[23] Luc Devroye. *Non-Uniform Random Variate Generation*. Springer, 1986. DOI: 10.1007/978-1-4613-8643-8.

[24] David Doty and David Soloveichik. “Stable Leader Election in Population Protocols Requires Linear Time”. In: *DISC*. Vol. 9363. LNCS. Springer, 2015, pp. 602–616. DOI: 10.1007/978-3-662-48653-5_40.

[25] Moez Draief and Milan Vojnovic. “Convergence Speed of Binary Interval Consensus”. In: *SIAM J. Control and Optimization* 50.3 (2012), pp. 1087–1109. DOI: 10.1137/110823018.

[26] Robert Elsässer and Tomasz Radzik. “Recent Results in Population Protocols for Exact Majority and Leader Election”. In: *Bull. EATCS* 126 (2018). URL: http://bulletin.eatcs.org/index.php/beatcs/article/view/549/546.

[27] Bennett L. Fox. “Generating Markov-Chain Transitions Quickly: I”. In: *INFORMS J. Comput.* 2.2 (1990), pp. 126–135.

[28] Leszek Gasieniec and Grzegorz Stachowiak. “Fast Space Optimal Leader Election in Population Protocols”. In: *SODA*. SIAM, 2018. DOI: 10.1137/1.9781611975031.169.

[29] Leszek Gasieniec, Grzegorz Stachowiak, and Przemyslaw Uznanski. “Almost Logarithmic-Time Space Optimal Leader Election in Population Protocols”. In: *SPAA*. ACM, 2019, pp. 93–102. DOI: 10.1145/3323165.3323178.

[30] Torben Hagerup, Kurt Mehlhorn, and J. Ian Munro. “Maintaining Discrete Probability Distributions Optimally”. In: *ICALP*. Vol. 700. LNCS. Springer, 1993, pp. 253–264.

[31] Lorenz Hübschle-Schneider and Peter Sanders. “Parallel Weighted Random Sampling”. In: *ESA*. Vol. 144. LIPIcs. Schloss Dagstuhl, 2019, 59:1–59:24.

[32] Fabian Kuhn and René Struik. "Random Walks Revisited: Extensions of Pollard's Rho Algorithm for Computing Multiple Discrete Logarithms". In: *Selected Areas in Cryptography*. Vol. 2259. LNCS. Springer, 2001, pp. 212–229. DOI: 10.1007/3-540-45537-X_17.

[33] Yossi Matias, Jeffrey Scott Vitter, and Wen-Chun Ni. "Dynamic Generation of Discrete Random Variates". In: *SODA*. ACM/SIAM, 1993, pp. 361–370.

[34] George B. Mertzios et al. "Determining Majority in Networks with Local Interactions and Very Small Local Memory". In: *ICALP (1)*. Vol. 8572. LNCS. Springer, 2014, pp. 871–882. DOI: 10.1007/978-3-662-43948-7_72.

[35] Yves Mocquard et al. "Counting with Population Protocols". In: *NCA*. IEEE Computer Society, 2015, pp. 35–42. DOI: 10.1109/NCA.2015.35.

[36] Sanguthevar Rajasekaran and Keith W. Ross. "Fast Algorithms for Generating Discrete Random Variates with Changing Distributions". In: *ACM Trans. Mod. Comp. Sim.* 3.1 (1993).

[37] Peter Sanders and Sebastian Winkel. "Super Scalar Sample Sort". In: *ESA*. Vol. 3221. LNCS. Springer, 2004, pp. 784–796. DOI: 10.1007/978-3-540-30140-0_69.

[38] David Soloveichik et al. "Computation with finite stochastic chemical reaction networks". In: *Nat. Comput.* 7.4 (2008), pp. 615–633. DOI: 10.1007/s11047-008-9067-y.

[39] Ernst Stadlober. "Ratio of uniforms as a convenient method for sampling from classical discrete distributions". In: *Winter Simulation Conference*. ACM Press, 1989, pp. 484–489. DOI: 10.1145/76738.76801.

[40] Michael D. Vose. "A Linear Algorithm For Generating Random Numbers With a Given Distribution". In: *IEEE Trans. Software Eng.* 17.9 (1991), pp. 972–975. DOI: 10.1109/32.92917.

[41] Alastair J. Walker. "An Efficient Method for Generating Discrete Random Variables with General Distributions". In: *ACM Math. Soft.* 3.3 (1977). DOI: 10.1145/355744.355749.

5 Fragile Complexity of Comparison-Based Algorithms

Peyman Afshani[*], Rolf Fagerberg[†], David Hammer[‡], Riko Jacob[§], Irina Kostitsyna[¶], Ulrich Meyer[‖], Manuel Penschuck[**], Nodari Sitchinava[††]

The version of the paper reproduced here is the short version presented at ESA 2019 (winning the best paper award). The full version of the paper [1] is available at `https://arxiv.org/abs/1901.02857`.

Abstract

We initiate a study of algorithms with a focus on the computational complexity of individual elements, and introduce the *fragile complexity* of comparison-based algorithms as the maximal number of comparisons any individual element takes part in. We give a number of upper and lower bounds on the fragile complexity for fundamental problems, including Minimum, Selection, Sorting and Heap Construction. The results include both deterministic and randomized upper and lower bounds, and demonstrate a separation between the two settings for a number of problems. The depth of a comparator network is a straight-forward upper bound on the worst case fragile complexity of the corresponding fragile algorithm. We prove that fragile complexity is a different and strictly easier property than the depth of comparator networks, in the sense that for some problems a fragile complexity equal to the best network depth can be achieved with less total work and that with randomization, even a lower fragile complexity is possible.

[*]Aarhus University, peyman@cs.au.dk

[†]University of Southern Denmark, rolf@imada.sdu.dk, Supported by the Independent Research Fund Denmark, Natural Sciences, grant DFF-7014-00041.

[‡]Goethe University Frankfurt and University of Southern Denmark, hammer@imada.sdu.dk, Supported by the Deutsche Forschungsgemeinschaft (DFG) under grants ME 2088/3-2 and ME 2088/4-2.

[§]IT University of Copenhagen, rikj@itu.dk

[¶]TU Eindhoven, i.kostitsyna@tue.nl

[‖]Goethe University Frankfurt, umeyer@ae.cs.uni-frankfurt.de, Supported by the Deutsche Forschungsgemeinschaft (DFG) under grants ME 2088/3-2 and ME 2088/4-2.

[**]Goethe University Frankfurt, mpenschuck@ae.cs.uni-frankfurt.de, Supported by the Deutsche Forschungsgemeinschaft (DFG) under grants ME 2088/3-2 and ME 2088/4-2.

[††]University of Hawaii at Manoa, nodari@hawaii.edu, Supported by the National Science Foundation under Grant No. CCF-1533823.

5.1 Introduction

Comparison-based algorithms is a classic and fundamental research area in computer science. Problems studied include minimum, median, sorting, searching, dictionaries, and priority queues, to name a few, and by now a huge body of work exists. The cost measure analyzed is almost always the total number of comparisons needed to solve the problem, either in the worst case or the expected case. Surprisingly, very little work has taken the viewpoint of the individual elements, asking the question: *how many comparisons must each element be subjected to?*

This question not only seems natural and theoretically fundamental, but is also practically well motivated: in many real world situations, comparisons involve some amount of destructive impact on the elements being compared, hence, keeping the maximum number of comparisons for each individual element low can be important. One example of such a situation is ranking of any type of consumable objects (wine, beer, food, produce), where each comparison reduces the available amount of the objects compared. Here, classical algorithms like QuickSort, which takes a single object and partitions the whole set with it, may use up this pivot element long before the algorithm completes. Another example is sports, where each comparison constitutes a match and takes a physical toll on the athletes involved. If a comparison scheme subjects one contestant to many more matches than others, both fairness to contestants and quality of result are impacted. The selection process could even contradict its own purpose—what is the use of finding a national boxing champion to represent a country at the Olympics if the person is injured in the process? Notice that in both examples above, quality of elements is difficult to measure objectively by a numerical value, hence one has to resort to relative ranking operations between opponents, i.e., comparisons. The detrimental impact of comparisons may also be of less directly physical nature, for instance if it involves a privacy risk for the elements compared, or if bias in the comparison process grows each time the same element is used.

Definition 2. *We say that a comparison-based algorithm $\mathcal{A}$ has* fragile complexity $f(n)$ *if each individual input element participates in at most $f(n)$ comparisons. We also say that $\mathcal{A}$ has* work $w(n)$ *if it performs at most $w(n)$ comparisons in total. We say that a particular element e has fragile complexity $f_e(n)$ in A if e participates in at most $f_e(n)$ comparisons.*

In this paper, we initiate the study of algorithms' fragile complexity—comparison-based complexity from the viewpoint of the individual elements—and present a number of upper and lower bounds on the fragile complexity for fundamental problems.

5.1.1 Previous work

One body of work relevant to what we study here is the study of sorting networks, propelled by the 1968 paper of Batcher [6]. In sorting networks, and more generally comparator networks, the notions of depth and size correspond to fragile complexity and standard worst case complexity,[1] respectively, since a network with depth $f(n)$ and size $w(n)$ easily can be converted into a comparison-based algorithm with fragile complexity $f(n)$ and work $w(n)$.

[1] For clarity, in the rest of the paper we call standard worst case complexity *work*.

Batcher gave sorting networks with $\mathcal{O}(\log^2 n)$ depth and $\mathcal{O}(n \log^2 n)$ size, based on clever variants of the MergeSort paradigm. A number of later constructions achieve the same bounds [10, 15, 16, 19], and for a long time it was an open question whether better results were possible. In the seminal result in 1983, Ajtai, Komlós, and Szemerédi [2, 4] answered this in the affirmative by constructing a sorting network of $\mathcal{O}(\log n)$ depth and $\mathcal{O}(n \log n)$ size. This construction is quite complex and involves expander graphs [20, 22], which can be viewed as objects encoding pseudorandomness, and which have many powerful applications in computer science and mathematics. The size of the constant factors in the asymptotic complexity of the AKS sorting network prevents it from being practical in any sense. It was later modified by others [8, 12, 17, 21], but finding a simple, optimal sorting network, in particular one not based on expander graphs, remains an open problem. Comparator networks for other problems, such as selection and heap construction have also been studied [5, 7, 14, 18, 23]. In all these problems the size of the network is super-linear.

As comparator networks of depth $f(n)$ and size $w(n)$ lead to comparison-based algorithms with $f(n)$ fragile complexity and $w(n)$ work, a natural question is, whether the two models are equivalent, or if there are problems for which comparison-based algorithms can achieve either asymptotically lower $f(n)$, or asymptotically lower $w(n)$ for the same $f(n)$.

One could also ask about the relationship between parallelism and fragile complexity. We note that parallel time in standard parallel models generally does not seem to capture fragile complexity. For example, even in the most restrictive exclusive read and exclusive write (EREW) PRAM model it is possible to create n copies of an element e in $\mathcal{O}(\log n)$ time and, thus, compare e to all the other input elements in $\mathcal{O}(\log n)$ time, resulting in $\mathcal{O}(\log n)$ parallel time but $\Omega(n)$ fragile complexity. Consequently, it is not clear whether Richard Cole's celebrated parallel merge sort algorithm [9] yields a comparison-based algorithm with low fragile complexity as it copies some elements.

5.1.2 Our contribution

In this paper we present algorithms and lower bounds for a number of classical problems, summarized in Table 5.1. In particular, we study finding the Minimum (Section 5.2), the Selection problem (Section 5.3), and Sorting (Section 5.4).

Minimum. The case of the deterministic algorithms is clear: using an adversary lower bound, we show that the minimum element needs to suffer $\Omega(\log n)$ comparisons and a tournament tree trivially achieves this bound (Subsection 5.2.1). The randomized case, however, is much more interesting. We obtain a simple algorithm where the probability of the minimum element suffering k comparisons is doubly exponentially low in k, roughly $1/2^{2^k}$ (see Subsection 5.2.2). As a result, the $\Theta(\log n)$ deterministic fragile complexity can be lowered to $O(1)$ expected or even $O(\log \log n)$ with high probability. We also show this latter high probability case is lower bounded by $\Omega(\log \log n)$ (Subsection 5.2.3). Furthermore, we can achieve a trade-off between the fragile complexity of the minimum element and the other elements. Here $\Delta = \Delta(n)$ is a parameter we can choose freely that basically upper bounds the fragile complexity of the non-minimal elements. We can find the minimum with $O(\log_\Delta n)$ expected fragile complexity while all the other elements suffer $O(\Delta + \log_\Delta n)$ comparisons (Subsec-

Problem		*Upper*		*Lower*
		$f(n)$	$w(n)$	$f(n)$
Minimum	Determ. (Sec. 5.2)	$\mathcal{O}(\log n)$ [T 7]	$\mathcal{O}(n)$	$f_{\min} = \Omega(\log n)$ [T 7]
	Rand. (Sec. 5.2)	$\langle \mathcal{O}(\log_\Delta n)^\dagger, \mathcal{O}(\Delta + \log_\Delta n)^\dagger \rangle$ [T 10] $\langle \mathcal{O}(1)^\dagger, \mathcal{O}(n^\varepsilon) \rangle$ (setting $\Delta = n^\varepsilon$) $O(\frac{\log n}{\log\log n})^\dagger$ [Cor 4]	$\mathcal{O}(n)$	$\langle \Omega(\log_\Delta n)^\dagger, \Delta \rangle$ [T 11] $\Omega(\frac{\log n}{\log\log n})^\dagger$ [Cor 4]
		$\langle O(\log_\Delta n \log\log \Delta)^\ddagger,$ $O(\Delta + \log_\Delta n \log\log \Delta)^\ddagger \rangle$ [T 10]	$\mathcal{O}(n)$ $\mathcal{O}(n)$	$f_{\min} =$ $= \Omega(\log\log n)^\ddagger$ [T 12]
Selection	Determ. (Sec. 5.3)	$\mathcal{O}(\log n)$ [T 13]	$\mathcal{O}(n)$ [T 13]	$\Omega(\log n)$ [Cor 3]
	Rand. (Sec. 5.3)	$\left\langle \mathcal{O}(\log\log n)^\dagger, \mathcal{O}\left(\sqrt{n}\right)^\dagger \right\rangle$ [T 14] $\left\langle \mathcal{O}\left(\frac{\log n}{\log\log n}\right)^\dagger, \mathcal{O}(\log^2 n)^\dagger \right\rangle$ [T 14]	$\mathcal{O}(n)^\dagger$	$\langle \Omega(\log_\Delta n)^\dagger, \Delta \rangle$ [T 11]
Merge	Determ. (Sec. 5.4)	$\mathcal{O}(\log n)$ [T 18]	$\mathcal{O}(n)$	$\Omega(\log n)$ [Lem 15]
Heap Constr.	Determ. (Sec. 5.5)	$\mathcal{O}(\log n)$ [Obs 19]	$\mathcal{O}(n)$	$\Omega(\log n)$ [T 7]

Table 5.1: Summary of presented results. Notation: $f(n)$ means fragile complexity; $w(n)$ means work; $\langle f_m(n), f_{rem}(n) \rangle$ means fragile complexity for the selected element (minimum/median) and for the remaining elements, respectively – except for lower bounds, where it means ⟨expected for the selected, limit for remaining⟩; † means holds in expectation; ‡ means holds with high probability $(1 - 1/n)$. $\varepsilon > 0$ is an arbitrary constant.

tion 5.2.3). Furthermore, this is tight: we show an $\Omega(\log_\Delta n)$ lower bound for the expected fragile complexity of the minimum element where the maximum fragile complexity of non-minimum elements is at most Δ.

Selection. Minimum finding is a special case of the selection problem where we are interested in finding an element of a given rank. As a result, all of our lower bounds apply to this problem as well. Regarding upper bounds, the deterministic case is trivial if we allow for $O(n \log n)$ work (via sorting). We show that this can be reduced to $O(n)$ time while keeping the fragile complexity of all the elements at $O(\log n)$ (Section 5.3). Once again, randomization offers a substantial improvement: e.g., we can find the median in $O(n)$ expected work and with $O(\log\log n)$ expected fragile complexity while non-median elements suffer $O(\sqrt{n})$ expected comparisons, or we can find the median in $O(n)$ expected work and with $O(\log n / \log\log n)$ expected fragile complexity while non-median elements suffer $O(\log^2 n)$ expected comparisons.

Sorting and other results. The deterministic selection, sorting, and heap construction fragile complexities follow directly from the classical results in comparator networks [4, 7].

However, we show a separation between comparator networks and comparison-based algorithms for the problem of Median (Section 5.3) and Heap Construction (Section 5.5), in the sense that depth/fragile complexity of $\mathcal{O}(\log n)$ can be achieved in $\mathcal{O}(n)$ work for comparison-based algorithms, but requires $\Omega(n \log n)$ [5] and $\Omega(n \log \log n)$ [7] sizes for comparator networks for the two problems, respectively. For sorting the two models achieve the same complexities: $\mathcal{O}(\log n)$ depth/fragile complexity and $\mathcal{O}(n \log n)$ size/work, which are the optimal bounds in both models due to the $\Omega(\log n)$ lower bound on fragile complexity for Minimum (Theorem 7) and the standard $\Omega(n \log n)$ lower bound on work for comparison-based sorting. However, it is an open problem whether these bounds can be achieved by simpler sorting algorithms than sorting networks, in particular whether expander graphs are necessary. One intriguing conjecture could be that any comparison-based sorting algorithm with $\mathcal{O}(\log n)$ fragile complexity and $\mathcal{O}(n \log n)$ work implies an expander graph. This would imply expanders, optimal sorting networks and fragile-optimal comparison-based sorting algorithms to be equivalent, in the sense that they all encode the same level of pseudorandomness.

We note that our lower bound of $\Omega(\log^2 n)$ on the fragile complexity of MergeSort (Theorem 15) implies the same lower bound on the depth of any sorting network based on binary merging, which explains why many of the existing simple sorting networks have $\Theta(\log^2 n)$ depth. Finally, our analysis of MergeSort on random inputs (Theorem 17) shows a separation between deterministic and randomized fragile complexity for such algorithms. In summary, we consider the main contributions of this paper to be:

- the introduction of the model of fragile complexity, which we find intrinsically interesting, practically relevant, and surprisingly overlooked
- the separations between this model and the model of comparator networks
- the separations between the deterministic and randomized setting within the model
- the lower bounds on randomized minimum finding

Due to space constraints, some proofs only appear in the full paper [1].

5.2 Finding the minimum

5.2.1 Deterministic Algorithms

As a starting point, we study deterministic algorithms that find the minimum among an input of n elements. Our results here are simple but they act as interesting points of comparison against the subsequent non-trivial results on randomized algorithms.

Theorem 7. *The fragile complexity of finding the minimum of n elements is $\lceil \log n \rceil$.*

Proof. The upper bound is achieved using a perfectly balanced tournament tree. The lower bound follows from a standard adversary argument. □

Observe that in addition to returning the minimum, the balanced tournament tree can also return the second smallest element, without any increase to the fragile complexity of the minimum. We refer to this deterministic algorithm that returns the smallest and the second smallest element of a set X as TournamentMinimum(X).

Corollary 3. *For any deterministic algorithm $\mathcal{A}$ that finds the median of n elements, the fragile complexity of the median element is at least $\lceil \log n \rceil - 1$.*

Proof. By a standard padding argument with $n - 1$ small elements. □

5.2.2 Randomized Algorithms for Finding the Minimum

We now show that finding the minimum is provably easier for randomized algorithms than for deterministic algorithms. We define f_{min} as the fragile complexity of the minimum and f_{rem} as the maximum fragile complexity of the remaining elements. For deterministic algorithms we have shown that $f_{\text{min}} \geq \log n$ regardless of f_{rem}. This is very different in the randomized setting. In particular, we first show that we can achieve $\mathbb{E}[f_{\text{min}}] = O(1)$ and $f_{\text{min}} = O(1) + \log \log n$ with high probability (we later show this high probability bound is also tight, Theorem 12).

```
1:  procedure SampleMinimum(X)   ▷ Returns the smallest and 2nd smallest element of X
2:      if |X| ≤ 8 return TournamentMinimum(X)
3:      Let A ⊂ X be a uniform random sample of X, with |A| = ⌈|X|/2⌉
4:      Let B ⊂ A be a uniform random sample of A, with |B| = ⌊|X|^{2/3}⌋
5:                  ▷ The minimum is either in (i) C ⊆ X \ A, (ii) D ⊆ A \ B or (iii) B
6:      (b1, b2) = SampleMinimum(B)          ▷ the minimum participates only in case (iii)
7:      Let D = {x ∈ A \ B | x < b2}          ▷ the minimum is compared once only in case (ii)
8:      Let (a'1, a'2) = SampleMinimum(D)                          ▷ only case (ii)
9:      Let (a1, a2) = TournamentMinimum(a'1, a'2, b1, b2)         ▷ case (ii) and (iii)
10:     Let C = {x ∈ X \ A | x < a2}                               ▷ only case (i)
11:     Let (c1, c2) = TournamentMinimum(C)                        ▷ only case (i)
12:     return TournamentMinimum(a1, a2, c1, c2)                   ▷ always
```

First, we show that this algorithm can actually find the minimum with expected constant number of comparisons. Later, we show that the probability that this algorithm performs t comparisons on the minimum drops roughly *doubly exponentially* on t.

We start with the simple worst-case analysis.

Lemma 10. *Algorithm SampleMinimum(X) achieves $f_{min} \leq 3 \log |X|$ in the worst case.*

Proof. First, observe that the smallest element in Lines 9 and 12 participates in at most one comparison because pairs of elements are already sorted. Then the fragile complexity of the minimum is defined by the maximum of the three cases:

1. One comparison each in Lines 10 and 12, plus (by Theorem 7) $\lceil \log |C| \rceil \leq \log |X|$ comparisons in Line 11.

2. One comparison each in Lines 7, 9, and 12, plus the recursive call in line 8.

3. One comparison each in Lines 6, 9, and 12, plus the recursive call in line 6.

The recursive calls in lines 8 and 6 are on at most $|X|/2$ elements because $B \subset A$, $D \subset A$, and $|A| = \lceil |X|/2 \rceil$. Consequently, the fragile complexity of the minimum is defined by the recurrence

$$T(n) \leq \begin{cases} \max\{3 + T(n/2), 2 + \log n\} & \text{if } n > 8 \\ 3 & \text{if } n \leq 8 \end{cases},$$

which solves to $T(n) \leq 3 \log n$.

□

Lemma 11. *Assume that in Algorithm SampleMinimum, the minimum y is in $X \setminus A$, i.e. we are in case (i). Then $\mathbf{Pr}[|C| = k \mid y \notin A] \leq \frac{k}{2^k}$ for any $k \geq 1$ and $n \geq 7$.*

Proof. There are $\binom{n-1}{\lceil n/2 \rceil}$ possible events of choosing a random subset $A \subset X$ of size $\lceil n/2 \rceil$ s.t. $y \notin A$. Let us count the number of the events $\{|C| = k \mid y \notin A\}$, which is equivalent to a_2, the second smallest element of A, being larger than exactly $k + 1$ elements of X.

For simplicity of exposition, consider the elements of $X = \{x_1, \ldots, x_n\}$ in sorted order. The minimum $y = x_1 \notin A$, therefore, a_1 (the smallest element of A) must be one of the k elements $\{x_2, \ldots, x_{k+1}\}$. By the above observation, $a_2 = x_{k+2}$. And the remaining $\lceil n/2 \rceil - 2$ elements of A are chosen from among $\{x_{k+3}, \ldots, x_n\}$. Therefore,

$$\mathbf{Pr}[|C| = k \mid y \notin A] = \frac{k \cdot \binom{n-(k+2)}{\lceil n/2 \rceil - 2}}{\binom{n-1}{\lceil n/2 \rceil}} = k \cdot \frac{(n-(k+2))!}{(\lfloor n/2 \rfloor - k)!(\lceil n/2 \rceil - 2)!} \cdot \frac{(\lceil n/2 \rceil)!(\lfloor n/2 \rfloor - 1)!}{(n-1)!}$$

Rearranging the terms, we get:

$$\mathbf{Pr}[|C| = k \mid y \notin A] = k \cdot \frac{(n-(k+2))!}{(n-1)!} \cdot \frac{(\lceil n/2 \rceil)!}{(\lceil n/2 \rceil - 2)!} \cdot \frac{(\lfloor n/2 \rfloor - 1)!}{(\lfloor n/2 \rfloor - k)!}$$

There are two cases to consider:

$$\begin{aligned} k = 1: \qquad \mathbf{Pr}[|C| = k \mid y \notin A] &= 1 \cdot \frac{1}{(n-1)(n-2)} \cdot \lceil n/2 \rceil \, (\lceil n/2 \rceil - 1) \cdot 1 \\ &\leq \frac{1}{(n-1)(n-2)} \cdot \frac{(n+1)}{2} \cdot \frac{(n-1)}{2} \\ &= \frac{n+1}{4 \cdot (n-2)} \leq \frac{1}{2} = \frac{k}{2^k} \qquad \text{for every } n \geq 5. \end{aligned}$$

$$
\begin{aligned}
k \geq 2: \qquad \mathbf{Pr}[|C| = k \mid y \notin A] &= k \cdot \frac{1}{\prod_{i=1}^{k+1}(n-i)} \cdot \lceil n/2 \rceil \left(\lceil n/2 \rceil - 1\right) \cdot \prod_{i=1}^{k-1} \left(\left\lfloor \frac{n}{2} \right\rfloor - i \right) \\
&\leq k \cdot \frac{1}{\prod_{i=1}^{k+1}(n-i)} \cdot \frac{n+1}{2} \cdot \frac{n-1}{2} \cdot \prod_{i=1}^{k-1} \frac{n-2i}{2} \\
&\leq \frac{k}{2^{k+1}} \cdot (n+1)(n-1) \cdot \frac{\prod_{i=1}^{k-1}(n-2i)}{\prod_{i=1}^{k+1}(n-i)} \\
&\leq \frac{k}{2^{k+1}} \cdot (n+1)(n-1) \cdot \frac{n-2}{(n-1)(n-2)(n-3)} \\
&= \frac{k}{2^{k+1}} \cdot \frac{n+1}{n-3} \leq \frac{k}{2^{k+1}} \cdot 2 = \frac{k}{2^k} \qquad \text{for every } n \geq 7.
\end{aligned}
$$

□

Theorem 8. *Algorithm* SampleMinimum *achieves* $\mathbb{E}[f_{min}] \leq 9$.

Proof. By induction on the size of X. In the base case $|X| \leq 8$, clearly $f_{\min} \leq 3$, implying the theorem.

Now assume that the calls in Line 8 and Line 6 have the property that $\mathbb{E}[f(b_1)] \leq 9$ and $\mathbb{E}[f(a'_1)] \leq 9$. Both in case (ii) and case (iii), the expected number of comparisons of the minimum is $\leq 9 + 3$. Case (i) happens with probability at least 1/2. In this case, the expected number of comparisons is 2 plus the ones from Line 11. By Lemma 11 we have $\mathbf{Pr}[|C| = k \mid \text{case (i)}] \leq k2^{-k}$. Because TournamentMinimum (actually any algorithm not repeating the same comparison) uses the minimum at most $k-1$ times, the expected number of comparisons in Line 11 is $\sum_{k=1}^{\lfloor n/2 \rfloor}(k-1)k2^{-k} \leq \sum_{k=1}^{\infty}(k-1)k2^{-k} \leq 4$. Combining the bounds we get $\mathbb{E}[f_{\min}] \leq \frac{9+3}{2} + \frac{2+4}{2} = 9$. □

Observe that the above proof did not use anything about the sampling of B, and also did not rely on TournamentMinimum.

Lemma 12. *For* $|X| > 2$ *and any* $\gamma > 1$: $\mathbf{Pr}\left[|D| \geq \gamma |X|^{1/3}\right] < |X| \exp(-\Theta(\gamma))$

Proof. Let $n = |X|$, $a = |A| = \lceil n/2 \rceil$ and $b = |B| = \left\lfloor n^{2/3} \right\rfloor$. The construction of the set B can be viewed as the following experiment. Consider drawing without replacement from an urn with b blue and $a - b$ red marbles. The i-th smallest element of A is chosen into B iff the i-th draw from the urn results in a blue marble. Then $|D| \geq \gamma |X|^{1/3} = \gamma n^{1/3}$ implies that this experiment results in at most one blue marble among the first $t = \gamma n^{1/3}$ draws. There are precisely $t + 1$ elementary events that make up the condition $|D| \geq t$, namely that the i-th draw is a blue marble, and where $i = 0$ stands for the event "all t marbles are red". Let us denote the probabilities of these elementary events as p_i.

Observe that each p_i can be expressed as a product of t factors, at least $t - 1$ of which stand for drawing a red marble, each upper bounded by $1 - \frac{b-1}{a}$. The remaining factor stands for

drawing the first blue marble (from the urn with $a-i$ marbles, b of which are blue), or another red marble. In any case we can bound

$$p_i \leq \left(1-\frac{b-1}{a}\right)^{t-1} \leq \left(1-\frac{b-1}{a}\right)^{\gamma n^{1/3}-1} = \exp\left(-\Theta\left(\frac{b\gamma n^{1/3}}{a}\right)\right).$$

Summing the $t+1$ terms, and observing $t+1<n$ if the event can happen at all, we get

$$\mathbf{Pr}[|D| \geq \gamma |X|^{1/3}] < n \cdot \exp\left(-\Theta\left(\frac{\gamma n^{1/3} n^{2/3}}{n/2}\right)\right) = n \cdot \exp\left(-\Theta(\gamma)\right).$$

□

Theorem 9. *There is a positive constant c, such that for any parameter $t \geq c$, the minimum in the Algorithm SampleMinimum(X) participates in at most $O(t+\log\log|X|)$ comparisons with probability at least $1-\exp(-2^t)2\log\log|X|$.*

Proof. Let $n=|X|$ and y be the minimum element. In each recursion step, we have one of three cases: (i) $y \in C \subseteq X \setminus A$, (ii) $y \in D \subseteq A \setminus B$ or (iii) $y \in B$. Since the three sets are disjoint, the minimum always participates in at most one recursive call. Tracing only the recursive calls that include the minimum, we use the superscript $X^{(i)}, A^{(i)}, B^{(i)}, C^{(i)}$, and $D^{(i)}$ to denote these sets at depth i of the recursion.

Let h be the first recursive level when $y \in C^{(h)}$, i.e., $y \notin A^{(h)}$. It follows that y will not be involved in the future recursive calls because it is in a single call to TournamentMinimum. Thus, at this level of recursion, the number of comparisons that y will accumulate is equal to $O(1)+\log|C^{(h)}|$. To bound this quantity, let $k=4\cdot 2^t$. Then, by Lemma 11, $\mathbf{Pr}[|C^{(h)}|>k] \leq k2^{-k} = 4\cdot 2^t \cdot 2^{-4\cdot 2^t} = 4\cdot 2^t \cdot 4^{-2^t} \cdot 4^{-2^t}$. Since $4x4^{-x} \leq 1$ for any $x \geq 1$, $\mathbf{Pr}[|C^{(h)}|>k] \leq 4^{-2^t}$ for any $t \geq 0$. I.e., the number of comparisons that y participates in at level h is at most $O(1)+\log k = O(1)+t$ with probability at least $1-4^{-2^t} \geq 1-\exp(-2^t)$.

Thus, it remains to bound the number of comparisons involving y at the recursive levels $i \in [1, h-1]$. In each of these recursive levels $y \notin C^{(i)}$, which only leaves the two cases: (ii) $y \in D^{(i)} \subseteq A^{(i)} \setminus B^{(i)}$ and (iii) $y \in B^{(i)}$. The element y is involved in at most $O(1)$ comparisons in lines 7, 9 and 12. The two remaining lines of the algorithm are lines 6 and 8 which are the recursive calls. We differentiate two types of recursive calls:

- Type 1: $|X^{(i)}| \leq 2^{4t}$. In this case, by Lemma 10, the algorithm will perform $O(t)$ comparisons at the recursive level i, as well as any subsequent recursive levels.
- Type 2: $|X^{(i)}| > 2^{4t}$. In this case, by Lemma 12 on the set $X^{(i)}$ and $\gamma = |X^{(i)}|^{1/3}$ we get:

  $$\mathbf{Pr}[|D^{(i)}| \geq \gamma |X^{(i)}|^{1/3}] < |X^{(i)}| \exp\left(-\Theta\left(|X^{(i)}|^{1/3}\right)\right) < \exp\left(-\Theta\left(|X^{(i)}|^{1/3}\right)\right)$$

  Note that since $|X^{(i)}|^{1/3} > 2^t$, by the definition of the Θ-notation, there exists a positive constant c, such that $\exp\left(-\Theta\left(|X^{(i)}|^{1/3}\right)\right) < \exp(-2^t)$. Thus, it follows that with probability $1-\exp(-2^t)$, we will recurse on a subproblem of size at most $\gamma|X^{(i)}|^{1/3} \leq |X^{(i)}|^{2/3}$. Let G_i be this (good) event, and thus $\mathbf{Pr}[G_i] \geq 1-\exp(-2^t)$.

Observe that the maximum number of times we can have good events of type 2 is very limited. With every such good event, the size of the subproblem decreases significantly and thus eventually we will arrive at a recursive call of type 1. Let j be this maximum number of "good" recursive levels of type 2. The problem size at the j-th such recursive level is at most $n^{(2/3)^{j-1}}$ and we must have that $n^{(2/3)^{j-1}} > 2^{4t}$ which reveals that we must have $j = O(\log\log n)$.

We are now almost done and we just need to use a union bound. Let G be the event that at the recursive level h, we perform at most $O(1) + t$ comparisons, and all the recursive levels of type 2 are good. G is the conjunction of at most $j + 1$ events and as we have shown, each such event holds with probability at least $1 - \exp(-2^t)$. Thus, it follows that G happens with probability $1-(j+1)\exp(-2^t) > 1-2\log\log n\exp(-2^t)$. Furthermore, our arguments show that if G happens, then the minimum will only particpate in $O(t+j) = O(t + \log\log n)$ comparisons. □

The major strengths of the above algorithm is the doubly exponential drop in probability of comparing the minimum with too many elements. Based on it, we can design another simple algorithm to provide a smooth trade-off between $f_{\min}$ and f_{rem}. Let $2 \le \Delta \le n$ be an integral parameter. We will design an algorithm that achieves $\mathbb{E}[f_{\min}] = O(\log_\Delta n)$ and $f_{\min} = O(\log_\Delta n \cdot \log\log\Delta)$ whp, and $f_{\text{rem}} = \Delta + O(\log_\Delta n \cdot \log\log\Delta)$ whp. For simplicity we assume n is a power of Δ. We build a fixed tournament tree T of degree Δ and of height $\log_\Delta n$ on X. For a node $v \in T$, let $X(v)$ be the set of values in the subtree rooted at v. The following code computes $m(v)$, the minimum value of $X(v)$, for every node v.

1: **procedure** TreeMinimum$_\Delta(X)$
2: For every leaf v, set $m(v)$ equal to the single element of $X(v)$.
3: For every internal node v with Δ children $u_1, \ldots, u_\Delta$ where the values $m(u_1), \ldots, m(u_\Delta)$ are known, compute $m(v)$ using SimpleMinimum algorithm on input $\{m(u_1), \ldots, m(u_\Delta)\}$.
4: Repeat the above step until the minimum of X is computed.

The correctness of TreeMinimum$_\Delta$ is trivial. So it remains to analyze its fragile complexity.

Theorem 10. *In TreeMinimum$_\Delta$, $\mathbb{E}[f_{min}] = O(\log_\Delta n)$ and $\mathbb{E}[f_{rem}] = \Delta + O(\log_\Delta n)$. Furthermore, with high probability, $f_{min} = O\left(\frac{\log n \log\log\Delta}{\log\Delta}\right)$ and $f_{rem} = O\left(\Delta + \frac{\log n \log\log\Delta}{\log\Delta}\right)$.*

Proof. First, observe that $\mathbb{E}[f_{\min}] = O(\log_\Delta n)$ is an easy consequence of Theorem 8. Now we focus on high probability bounds. Let $k = c \cdot h \log\ln\Delta$, and $h = \log_\Delta n$ for a large enough constant c. There are h levels in T. Let $\mathbf{f}_i$ be the random variable that counts the number of comparisons the minimum participates in at level i of T. Observe that these are independent random variables. Let $f_1, \ldots, f_h$ be integers such that $f_i \ge 1$ and $\sum_{i=1}^h f_i = k$, and let c' be the constant hidden in the big-O notation of Theorem 9. Use Theorem 9 h times (with n set to Δ, and $t = f_i$), and also bound $2\log\log\Delta < \Delta$ to get

$$\mathbf{Pr}\Big[\mathbf{f}_1 \ge c'(f_1 + \log\log\Delta) \vee \cdots \vee \mathbf{f}_h \ge c'(f_h + \log\log\Delta)\Big] \le \Delta^h e^{-\sum_i 2^{f_i}} \le \Delta^h e^{-h2^{k/h}}$$

where the last inequality follows from the inequality of arithmetic and geometric means (specifically, observe that $\sum_{i=1}^h 2^{f_i}$ is minimized when all f_i's are distributed evenly).

Now observe that the total number of different integral sequences $f_1, \ldots, f_h$ that sum up to k is bounded by $\binom{h+k}{h}$ (this is the classical problem of distributing k identical balls into h distinct bins). Thus, we have

$$\begin{aligned}\mathbf{Pr}[f_{\min} = O(k + h\log\log\Delta)] &\leq \binom{h+k}{h} \cdot \Delta^h \frac{1}{e^{h\cdot 2^{k/h}}} \leq \left(\frac{e(h+k)}{h}\right)^h \cdot \Delta^h \frac{1}{e^{h\cdot 2^{k/h}}} \\ &\leq \left(\frac{O\left(\frac{k}{h}\right)\cdot\Delta}{e^{2^{k/h}}}\right)^h = \left(\frac{O\left(\Delta^2\right)}{e^{2^{c\log\ln\Delta}}}\right)^h < \left(\frac{O(\Delta^2)}{e^{\ln^c\Delta}}\right)^h < \left(\frac{\Delta^3}{\Delta^{\ln^{c-1}\Delta}}\right)^h < \Delta^{-ch} = n^{-c}\end{aligned}$$

where in the last step we bound $(\ln\Delta)^{c-1} - 3 > c$ for large enough c and $\Delta \geq 3$. This is a high probability bound for $f_{\min}$. To bound f_{rem}, observe that for every non-minimum element x, there exists a lowest node v such that x is not $m(v)$. If x is not passed to the ancestors of v, x suffers at most Δ comparisons in v, and below v x behaves like the minimum element, which means that the above analysis applies. This yields that whp we have $f_{\text{rem}} = \Delta + O\left(\frac{\log n\log\log\Delta}{\log\Delta}\right)$. □

5.2.3 Randomized Lower Bounds for Finding the Minimum

Expected Lower Bound for the Fragile Complexity of the Minimum.

The following theorem is our main result.

Theorem 11. *In any randomized minimum finding algorithm with fragile complexity of at most Δ for any element, the expected fragile complexity of the minimum is at least $\Omega(\log_\Delta n)$.*

Note that this theorem implies the fragile complexity of finding the minimum:

Corollary 4. *Let $f(n)$ be the expected fragile complexity of finding the minimum (i.e. the smallest function such that some algorithm achieves $f(n)$ fragile complexity for all elements (minimum and the rest) in expectation). Then $f(n) = \Theta(\frac{\log n}{\log\log n})$.*

Proof. Use Theorem 10 as the upper bound and Theorem 11, both with $\Delta = \frac{\log n}{\log\log n}$, observing that if $f(n)$ is an upper bound that holds with high probability, it is also an upper bound on the expectation. □

To prove Theorem 11 we give a lower bound for a *deterministic* algorithm $\mathcal{A}$ on a random input of n values, $x_1, \ldots, x_n$ where each x_i is chosen iid and uniformly in $(0, 1)$. By Yao's minimax principle, the lower bound on the expected fragile complexity of the minimum when running $\mathcal{A}$ also holds for any randomized algorithm.

We prove our lower bound in a model that we call "comparisons with additional information (CAI)": if the algorithm $\mathcal{A}$ compares two elements x_i and x_j and it turns out that $x_i < x_j$, then the value x_j is revealed to the algorithm. Clearly, the algorithm can only do better with this extra information. The heart of the proof is the following lemma which also acts as the "base case" of our proof.

Lemma 13. *Let Δ be an upper bound on f_{rem}. Consider T values $x_1, \ldots, x_T$ chosen iid and uniformly in $(0, b)$. Consider a deterministic algorithm $\mathcal{A}$ in CAI model that finds the minimum value y among $x_1, \ldots, x_T$. If $T > 1000\Delta$, then with probability at least $\frac{7}{10}$ $\mathcal{A}$ will compare y against an element x such that $x \geq b/(100\Delta)$.*

Proof. By simple scaling, we can assume $b = 1$. Let p be the probability that $\mathcal{A}$ compares y against a value larger than $1/(100\Delta)$. Let I_{small} be the set of indices i such that $x_i < 1/(100\Delta)$. Let $\mathcal{A}'$ be a deterministic algorithm in CAI model such that:

- $\mathcal{A}'$ is given all the indices in I_{small} (and their corresponding values) except for the index of the minimum. We call these the known values.
- $\mathcal{A}'$ minimizes the probability p' of comparing the y against a value larger than $1/(100\Delta)$.
- $\mathcal{A}'$ finds the minimum value among the unknown values.

Since $p' \leq p$, it suffices to bound p' from below. We do this in the remainder of the proof.

Observe that the expected number of values x_i such that $x_i < 1/(100\Delta)$ is $T/(100\Delta)$. Thus, by Markov's inequality, $\mathbf{Pr}[|I_{\text{small}}| \leq T/(10\Delta)] \geq \frac{9}{10}$. Let's call the event $|I_{\text{small}}| \leq T/(10\Delta)$ the good event. For algorithm $\mathcal{A}'$ all values smaller than $1/(100\Delta)$ except for the minimum are known. Let U be the set of indices of the unknown values. Observe that a value x_i for $i \in U$ is either the minimum or larger than $1/(100\Delta)$, and that $|U| = T - |I_{\text{small}}| + 1 > \frac{9}{10}T$ (using $\Delta \geq 1$) in the good event. Because $\mathcal{A}'$ is a deterministic algorithm, the set U is split into set F of elements that have their first comparison against a known element, and set W of those that are first compared with another element with index in U. Because of the global bound Δ on the fragile complexity of the known elements, we know $|F| < \Delta \cdot |I_{\text{small}}| \leq \Delta T/(10\Delta) = T/10$. Combining this with the probability of the good event, by union bound, the probability of the minimum being compared with a value greater than $1/(100\Delta)$ is at least $1-(1-\frac{9}{10})-(1-\frac{8}{9}) \geq 7/10$. □

Based on the above lemma, our proof idea is the following. Let $G = 100\Delta$. We would like to prove that on average $\mathcal{A}$ cannot avoid comparing the minimum to a lot of elements. In particular, we show that, with constant probability, the minimum will be compared against some value in the range $[G^{-i}, G^{-i+1}]$ for every integer i, $1 \leq i \leq \frac{\log_G n}{2}$. Our lower bound then follows by an easy application of the linearity of expectations. Proving this, however, is a little bit tricky. However, observe that Lemma 13 already proves this for $i = 1$. Next, we use the following lemma to apply Lemma 13 over all values of i, $1 \leq i \leq \frac{\log_G n}{2}$.

Lemma 14. *For a value b with $0 < b < 1$, define $p_k = \binom{n}{k} b^i (1-b)^{n-k}$, for $0 \leq k \leq n$. Choosing $x_1, \ldots, x_n$ iid and uniformly in $(0, 1)$ is equivalent to the following: with probability p_k, uniformly sample a set I of k distinct indices in $\{1, \ldots, n\}$ among all the subsets of size k. For each $i \in I$, pick x_i iid and uniformly in $(0, b)$. For each $i \notin I$, pick x_i iid and uniformly in $(b, 1)$.*

Proof. It is easy to see that choosing $x_1, \ldots, x_n$ iid uniformly in $(0, 1)$ is equivalent to choosing a point X uniformly at random inside an n dimensional unit cube $(0, 1)^n$. Therefore, we will

prove the equivalence between (i) the distribution defined in the lemma, and (ii) choosing such point X.

Let Q be the n-dimensional unit cube. Subdivide Q into 2^n rectangular region defined by the Cartesian product of intervals $(0, b)$ and $(b, 1)$, i.e., $\{(0, b), (b, 1)\}^n$ (or alternatively, bisect Q with n hyperplanes, with the i-th hyperplane perpendicular to the i-th axis and intersecting it at coordinate equal to b).

Consider the set R_k of rectangles in $\{(0, b), (b, 1)\}^n$ with exactly k sides of length b and $n-k$ sides of length $1-b$. Observe that for every choice of k (distinct) indices $i_1, \ldots, i_k$ out of $\{1, \ldots, n\}$, there exists exactly one rectangle r in R_k such that r has side length b at dimensions $i_1, \ldots, i_k$, and all the other sides of r has length $n - k$. As a result, we know that the number of rectangles in R_k is $\binom{n}{k}$ and the volume of each rectangle in R_k is $b^k(1 - b)^k$. Thus, if we choose a point X randomly inside Q, with probability p_k it will fall inside a rectangle r in R_k; furthermore, conditioned on this event, the dimensions $i_1, \ldots, i_k$ where r has side length b is a uniform subset of k distinct indices from $\{1, \ldots, n\}$. □

Remember that our goal was to prove that with constant probability, the minimum will be compared against some value in the range $[G^{-i}, G^{-i+1}]$ for every integer i, $1 \leq i \leq \frac{\log_G n}{2}$. We can pick $b = G^{-i+1}$ and apply Lemma 14. We then observe that it is very likely that the set of indices I that we are sampling in Lemma 14 will contain many indices. For every element x_i, $i \in I$, we are sampling x_i independently and uniformly in $(0, b)$ which opens the door for us to apply Lemma 13. Then we argue that Lemma 13 would imply that with constant probability the minimum will be compared against a value in the range $(b/G, b) = (G^{-i}, G^{-i+1})$. The lower bound claim of Theorem 11 then follows by invoking the linearity of expectations.

We are ready to prove that the minimum element will have $\Omega(\log_\Delta n)$ comparisons on average.

Proof of Theorem 11. First, observe that we can assume $n \geq (100,000\Delta)^2$ as otherwise we are aiming for a trivial bound of $\Omega(1)$.

We create an input set of n values $x_1, \ldots, x_n$ where each x_i is chosen iid and uniformly in $(0, 1)$. Let $G = 100\Delta$. Consider an integer i such that $1 \leq i < \frac{\log_G n}{2}$. We are going to prove that with constant probability, the minimum will be compared against a value in the range (G^{-i}, G^{-i+1}), which, by linearity of expectation, shows the stated $\Omega(\log_\Delta n)$ lower bound for the fragile complexity of the minimum.

Consider a fixed value of i. Let S be the set of indices with values that are smaller than G^{-i+1}. Let p be the probability that $\mathcal{A}$ compares the minimum against an x_j with $j \in S$ such that $x_j \geq G^{-i}$. To prove the theorem, it suffices to prove that p is lower bounded by a constant. Now consider an algorithm $\mathcal{A}'$ that finds the minimum but for whom all the values other than those in S have been revealed and furthermore, assume $\mathcal{A}'$ minimizes the probability of comparing the minimum against an element $x \geq G^{-i}$ (in other words, we pick the algorithm which minimizes this probability, among all the algorithms). Clearly, $p' \leq p$. In the rest of the proof we will give a lower bound for p'.

Observe that $|S|$ is a random variable with binomial distribution. Hence $\mathbb{E}[|S|] = nG^{-i+1} > \sqrt{n}$ where the latter follows from $i < \frac{\log_G n}{2}$. By the properties of the binomial distribution we

have that $\mathbf{Pr}\left[|S| < \frac{\mathbb{E}[|S|]}{100}\right] < \frac{1}{10}$. Thus, with probability at least $\frac{9}{10}$, we will have the "good" event that $|S| \geq \frac{\mathbb{E}[|S|]}{100} \geq \frac{\sqrt{n}}{100}$.

In case of the good event, Lemma 14 implying that conditioned on S being the set of values smaller than G^{-i+1}, each value x_j with $j \in S$ is distributed independently and uniformly in the range $(0, G^{-i+1})$. As a result, we can now invoke Lemma 13 on the set S with $T = |S|$. Since $n \geq (100,000\Delta)^2$ we have $T = |S| \geq \frac{\sqrt{n}}{100} \geq \frac{100{,}000\Delta}{100}$. By Lemma 13, with probability at least $\frac{7}{10}$, the minimum will be compared against a value that is larger than G^{-i}.

Thus, by law of total probability, it follows that in case of a good event, with probability $\frac{7}{10}$ the minimum will be compared to a value in the range (G^{-i}, G^{-i+1}). However, as the good event happens with probability $\frac{9}{10}$, it follows that with probability at least $1-(1-\frac{7}{10})-(1-\frac{9}{10}) = \frac{6}{10}$, the minimum will be compared against a value in the range (G^{-i}, G^{-i+1}). □

Lower bound for the fragile complexity of the minimum whp.

With Theorem 9 in Subsection 5.2.2, we show in particular that SampleMinimum guarantees that the fragile complexity of the minimum is at most $\mathcal{O}(\log\log n)$ with probability at least $1 - 1/n^c$ for any $c > 1$. (By setting $t = 2\log\log n$).

Here we show that this is optimal up to constant factors in the fragile complexity.

Theorem 12. *For any constant $\varepsilon > 0$, there exists a value of n_0 such that the following holds for any randomized algorithm $\mathcal{A}$ and for any $n > n_0$: there exists an input of size n such that with probability at least $n^{-\varepsilon}$, $\mathcal{A}$ performs $\geq \frac{1}{2}\log\log n$ comparisons with the minimum.*

Proof. We use (again) Yao's principle and consider a fixed deterministic algorithm $\mathcal{A}$ working on the uniform input distribution, i.e., all input permutations have probability $1/n!$. Let $f = \frac{1}{2}\log\log n$ be the upper bound on the fragile complexity of the minimum. Let $k = 2^f = \sqrt{\log n}$ and let S be the set of the k smallest input values. Let π be a uniform permutation (the input) and $\pi(S)$ be the permutation of the elements of S in π. Observe that $\pi(S)$ is a uniform permutation of the elements of S. We reveal the elements not in S to $\mathcal{A}$. So, $\mathcal{A}$ only needs to find the minimum in $\pi(S)$. By Theorem 7 there is at least one "bad" permutation of S which forces algorithm $\mathcal{A}$ to do $\log k = f$ comparisons on the smallest element. Observe $\log k! < \log k^k = k\log k = \sqrt{\log n}\frac{1}{2}\log\log n$. Observe that there exists a value of n_0 such that for $n > n_0$ the right hand side is upper bounded by $\varepsilon\log n$, so $k! \leq n^\varepsilon$, for $n > n_0$. Hence, the probability of a "bad" permutation is at least $1/k! > n^{-\varepsilon}$. □

5.3 Selection and median

The (n,t)-*selection problem* asks to find the t-th smallest element among n elements of the input. The simplest solution to the (n,t)-selection problem is to sort the input. Therefore, it can be solved in $\mathcal{O}(\log n)$ fragile complexity and $\mathcal{O}(n\log n)$ work by using the AKS sorting network [2]. For comparator networks, both of these bounds are optimal: the former is shown by Theorem 7 (and in fact it applies also to any algorithm) and the latter is shown in the full version of this paper [1].

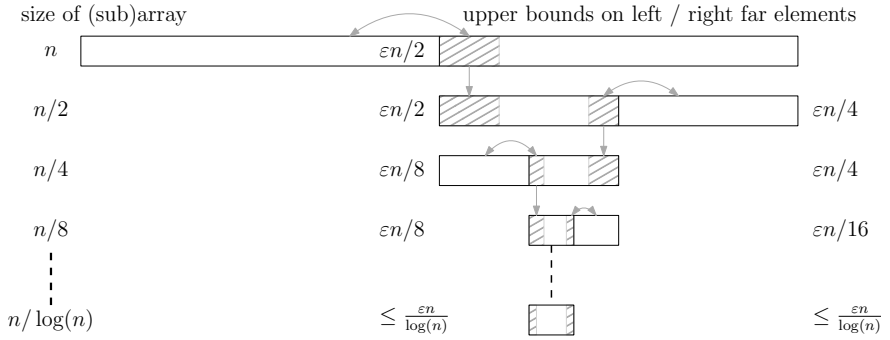


Figure 5.1: Illustration of the alternating division process using ε-halvers.

In contrast, in this section we show that comparison-based algorithms can do better: we can solve Selection deterministically in $\Theta(n)$ work and $\Theta(\log n)$ fragile complexity, thus, showing a separation between the two models. However, to do that, we resort to constructions that are based on expander graphs. Avoiding usage of the expander graphs or finding a simpler optimal deterministic solution is an interesting open problem (see Section 5.6). Moreover, in Subsection 5.3.2 we show that we can do even better by using randomization.

5.3.1 Deterministic selection

Theorem 13. *detMed There is a deterministic algorithm for Selection which performs $\mathcal{O}(n)$ work and has $\mathcal{O}(\log n)$ fragile complexity.*

Proof sketch. It suffices to just find the median since by simple padding we can generalize the solution for the (n, t)-selection problem.

We use *ε-halvers* that are the central building blocks of the AKS sorting network. An ε-halver approximately performs a partitioning of an array of size n into the smallest half and the largest half of the elements. More precisely, for any $m \leq n/2$, at most εn of the m smallest elements will end up in the right half of the array, and at most εn of the m largest elements will end up in the left half of the array. Using expander graphs, a comparator network implementing an ε-halver of constant depth can be built [2, 3]. We use the corresponding comparison-based algorithm of constant fragile complexity.

The idea is to use ε-halvers to find roughly $\frac{n}{\log n}$ elements with rank between $(1+\alpha)\frac{n}{2}$ and $(2-\alpha)\frac{n}{2}$ and also roughly $\frac{n}{\log n}$ elements with rank between $\alpha\frac{n}{2}$ and $(1-\alpha)\frac{n}{2}$, for some constant $0 < \alpha < 1$. This is done by repeatedly using ε-halvers but alternating between selecting the left half and the right half (Figure 5.1). Using these, we filter the remaining elements and discard a constant fraction of them. Then we recurse on the remaining elements. The details are a bit involved, as we have to guarantee that no element accumulates too many comparisons throughout the recursions. We have to do some bookkeeping as well as some additional ideas to provide this guarantee. Details can be found in the full version of the paper [1]. □

Corollary 5. *There is a deterministic algorithm for partition which performs $\mathcal{O}(n)$ work and has $\mathcal{O}(\log n)$ fragile complexity.*

Proof. At the end of the SELECTION algorithm, the set of elements smaller (larger) than the median is the union of the respective filtered sets (sets $\mathcal{L}$ and $\mathcal{R}$ in the proof in the full version of the paper [1]) and the first (last) half of the sorted set in the base case of the recursion. Again, simple padding generalizes this to (n, t)-partition for arbitrary $t \neq \frac{n}{2}$. □

5.3.2 Randomized selection

In the full paper [1], we present the details of an expected work-optimal selection algorithm with a trade-off between the expected fragile complexity $f_{\text{med}}(n)$ of the selected element and the maximum expected fragile complexity $f_{\text{rem}}(n)$ of the remaining elements. In particular, we obtain the following combinations:

Theorem 14. *Randomized selection is possible in expected linear work, while achieving expected fragile complexity of the median* $\mathbb{E}[f_{med}(n)] = \mathcal{O}(\log \log n)$ *and of the remaining elements* $\mathbb{E}[f_{rem}(n)] = \mathcal{O}(\sqrt{n})$*, or* $\mathbb{E}[f_{med}(n)] = \mathcal{O}\left(\frac{\log n}{\log \log n}\right)$ *and* $\mathbb{E}[f_{rem}(n)] = \mathcal{O}(\log^2 n)$.

5.4 Sorting

Recall from Section 5.1 that the few existing sorting networks with depth $\mathcal{O}(\log n)$ are all based on expanders, while a number of $\mathcal{O}(\log^2 n)$ depth networks have been developed based on binary merging. Here, we study the power of the mergesort paradigm with respect to fragile complexity. We first prove that any sorting algorithm based on binary merging must have a worst-case fragile complexity of $\Omega(\log^2 n)$. This provides an explanation why all existing sorting networks based on merging have a depth no better than this. We also prove that the standard mergesort algorithm on random input has fragile complexity $\mathcal{O}(\log n)$ with high probability, thereby showing a separation between the deterministic and the randomized situation for binary mergesorts. Finally, we demonstrate that the standard mergesort algorithm has a worst-case fragile complexity of $\Theta(n)$, but that this can be improved to $\mathcal{O}(\log^2 n)$ by changing the merging algorithm to use exponential search. The omitted proofs can be found in the full paper [1].

Lemma 15. *lemMergeLB Merging of two sorted sequences* A *and* B *has fragile complexity at least* $\lfloor \log_2 |A| \rfloor + 1$.

Theorem 15. *Any binary mergesort has fragile complexity* $\Omega(\log^2 n)$.

Proof. The adversary is the same as in the proof of Lemma 16, except that as scapegoat element for a merge of A and B it always chooses the scapegoat from the *larger* of A and B. We claim that for this adversary, there is a constant $c > 0$ such that for any node v in the mergetree, its scapegoat element has participated in at least $c \log^2 n$ comparisons in the subtree of v, where n is the number of elements merged by v. This implies the theorem.

We prove the claim by induction on n. The base case is $n = \mathcal{O}(1)$, where the claim is true for small enough c, as the scapegoat by Lemma 15 will have participated in at least one comparison. For the induction step, assume v merges two sequences of sizes n_1 and n_2, with

$n_1 \geq n_2$. By the base case, we can assume $n_1 \geq 3$. Using Lemma 15, we would like to prove for the induction step

$$c \log^2 n_1 + \lfloor \log n_2 \rfloor + 1 \geq c \log^2 (n_1 + n_2). \tag{5.1}$$

This will follow if we can prove

$$\log^2 n_1 + \frac{\log n_2}{c} \geq \log^2 (n_1 + n_2)\,. \tag{5.2}$$

The function $f(x) = \log^2 x$ has first derivative $2(\log x)/x$ and second derivative $2(1-\log x)/x^2$, which is negative for $x > e = 2.71\ldots$. Hence, $f(x)$ is concave for $x > e$, which means that first order Taylor expansion (alias the tangent) lies above f, i.e., $f(x_0)+f'(x_0)(x-x_0) \geq f(x)$ for $x_0, x > e$. Using $x_0 = n_1$ and $x = n_1+n_2$ and substituting the first order Taylor expansion into the right side of (5.2), we see that (5.2) will follow if we can prove

$$\frac{\log n_2}{c} \geq 2\frac{\log n_1}{n_1} n_2\,,$$

which is equivalent to

$$\frac{\log n_2}{n_2} \geq 2c\frac{\log n_1}{n_1}\,. \tag{5.3}$$

Since $n_1 \geq n_2$ and $(\log x)/x$ is decreasing for $x \geq e$, we see that (5.3) is true for $n_2 \geq 3$ and c small enough. Since $\log(3)/3 = 0.366\ldots$ and $\log 2/2 = 0.346\ldots$, it is also true for $n_2 = 2$ and c small enough. For the final case of $n_2 = 1$, the original inequality (5.1) reduces to

$$\log^2 n_1 + \frac{1}{c} \geq \log^2 (n_1 + 1)\,. \tag{5.4}$$

Here we can again use concavity and first order Taylor approximation with $x_0 = n_1$ and $x = n_1 + 1$ to argue that (5.4) follows from

$$\frac{1}{c} \geq 2\frac{\log n_1}{n_1}\,.$$

which is true for c small enough, as $n_1 \geq 3$ and $(\log x)/x$ is decreasing for $x \geq e$. □

Theorem 16. *Standard MergeSort with linear merging has a worst-case fragile complexity of* $\Theta(n)$.

Lemma 16. *Standard MergeSort has fragile complexity* $\Omega(\log^2 n)$.

Proof. In MergeSort, when merging two sorted sequences A and B, no comparisons between elements of A and B have taken place before the merge. Also, the sorted order of $A \cup B$ has to be decided by the algorithm after the merge. We can therefore run the adversary argument from the proof of Lemma 15 in all nodes of the mergetree of MergeSort. If the adversary reuses scapegoat elements in a bottom-up fashion—that is, as scapegoat for a merge of A and

B chooses one of the two scapegoats from the two merges producing A and B—then the scapegoat at the root of the mergetree has participated in

$$\Omega(\sum_{i=0}^{\log n} \log 2^i) = \Omega(\sum_{i=0}^{\log n} i) = \Omega(\log^2 n)$$

comparisons, by Lemma 15 and the fact that a node at height i in the mergetree of standard MergeSort operates on sequences of length $\Theta(2^i)$. □

Observation 2. *Consider two sorted sequences $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$. In linear merging, the fragile complexity of element a_i is at most $\ell + 1$ where ℓ is the largest number of elements from B that are placed directly in front of a_i (i.e. $b_j < \ldots < b_{j+\ell-1} < a_i$).*

Lemma 17. *Let $X = \{x_1, \ldots, x_{2k}\}$ be a finite set of distinct elements, and consider a random bipartition $X_L, X_R \subset X$ with $|X_L| = |X_R| = k$ and $X_L \cap X_R = \varnothing$, such that $\mathbf{Pr}[x_i \in X_L] = 1/2$. Consider an arbitrary ordered set $Y = \{y_1, \ldots, y_m\} \subset X$ with $m \leq k$. Then $\mathbf{Pr}[Y \subseteq X_L \vee Y \subseteq X_R] < 2^{1-m}$.*

Proof.

$$\mathbf{Pr}[Y \subseteq X_L \vee Y \subseteq X_R] = 2\prod_{i=1}^{m} \mathbf{Pr}[\, y_i \in X_L \,|\, y_1, \ldots y_{i-1} \in X_L \,] = 2\frac{(2k)^{-m}k!}{(k-m)!} \leq 2 \cdot 2^{-m}.$$

□

Theorem 17. *Standard MergeSort with linear merging on a randomized input permutation has a fragile complexity of $\mathcal{O}(\log n)$ with high probability.*

Proof. Let $Y = (y_1, \ldots, y_n)$ be the input-sequence, π^{-1} be the permutation that sorts Y and $X = (x_1, \ldots, x_n)$ with $x_i = y_{\pi^{-1}(i)}$ be the sorted sequence. Wlog we assume that all elements are unique[2], that any input permutation π is equally likely[3], and that n is a power of two.

Merging in one layer. Consider any merging-step in the mergetree. Since both input sequences are sorted, the only information still observable from the initial permutation is the bi-partitioning of elements into the two subproblems. Given π, we can uniquely retrace the mergetree (and vice-versa): we identify each node in the recursion tree with the set of elements it considers. Then, any node with elements $X_P = \{y_\ell, \ldots, y_{\ell+2k-1}\}$ has children

$$X_L = \left\{ x_{\pi(i)} \;\middle|\; \ell \leq \pi(i) \leq \ell + k - 1 \right\} = \{y_\ell, \ldots, y_{\ell+k-1}\},$$
$$X_R = \left\{ x_{\pi(i)} \;\middle|\; \ell + k \leq \pi(i) \leq \ell + 2k - 1 \right\} = \{y_{\ell+k}, \ldots, y_{\ell+2k-1}\}.$$

Hence, locally our input permutation corresponds to an stochastic experiment in which we randomly draw exactly half of the parent's elements for the left child, while the remainder goes to right.

This is exactly the situation in Lemma 17. Let N_i be a random variable denoting the number of comparisons of element y_i in the merging step. Then, from Observation 2 and Lemma 17 it

[2] If this is not the case, use input sequence $Y' = ((y_1, 1), \ldots, (y_n, n))$ and lexicographical compares.
[3] If not shuffle it before sorting in linear time and no fragile comparisons.

follows that $\mathbf{Pr}[N_i = m{+}1] \leq 2^{-m}$. Therefore N_i is stochastically dominated by $N_i \preceq 1{+}Y_i$ where Y_i is a geometric random variable with success probability $p = 1/2$.

Merging in all layers. Let $N_{j,i}$ be the number of times element y_i is compared in the j-th recursion layer and define $Y_{j,i}$ analogously. Due to the recursive partitioning argument, $N_{j,i}$ and $Y_{j,i}$ are iid in j. Let N_i^T be the total number of comparisons of element i, i.e. $N_i^T \preceq \log_2 n + \sum_{j=1}^{\log_2 n} Y_{j,i}$. Then a tail bound on the sum of geometric variables (Theorem 2.1 in [13]) yields:

$$\mathbf{Pr}\left[\sum\nolimits_{j=1}^{\log_2 n} Y_{j,i} \geq \lambda \mathbb{E}\left[\sum\nolimits_{j=1}^{\log_2 n} Y_{j,i}\right] = 2\lambda \log_2 n\right] \overset{[13]}{\leq} \exp\left(-\frac{1}{2}\frac{2\ln n}{\ln 2}[\lambda{-}1{-}\log\lambda]\right) = n^{-2},$$

where we set $\lambda \approx 3.69$ in the last step solving $\lambda - \log\lambda = 2\log 2$. Thus, we bound the probability $\mathbf{Pr}\left[N_i^T \geq (1{+}2\lambda)\log_2 n\right] \leq n^{-2}$.

Fragile complexity. It remains to show that with high probability no element exceeds the claimed fragile complexity. We use a union bound on N_i^T for all i:

$$\mathbf{Pr}\left[\max_i\{N_i^T\} = \omega(\log n)\right] \leq n\mathbf{Pr}\left[N_i^T = \omega(\log n)\right] \leq 1/n\,. \qquad \square$$

Theorem 18. *thmMergeExponential Exponential merging of two sequences $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$ has a worst-case fragile complexity of $\mathcal{O}(\log n)$.*

Corollary 6. *corMergesortExponential Applying Theorem 18 to standard MergeSort with exponential merging yields a fragile complexity of $\mathcal{O}(\log^2 n)$ in the worst-case.*

5.5 Constructing binary heaps

Theorem 19. *The fragile complexity of the standard binary heap construction algorithm of Floyd [11] is $\mathcal{O}(\log n)$.*

The above observation is easy to verify (shown in the full paper [1]). We note that this fragile complexity is optimal by Theorem 7, since Heap Construction is stronger than Minimum. Brodal and Pinotti [7] showed how to construct a binary heap using a comparator network in $\Theta(n \log\log n)$ size and $\mathcal{O}(\log n)$ depth. They also proved a matching lower bound on the size of the comparator network for this problem. This, together with Observation 19 and the fact that Floyd's algorithm has work $\mathcal{O}(n)$, gives a separation between work of fragility-optimal comparison-based algorithms and size of depth-optimal comparator networks for Heap Construction.

5.6 Conclusions

In this paper we introduced the notion of fragile complexity of comparison-based algorithms and we argued that the concept is well-motivated because of connections both to real world situations (e.g., sporting events), as well as other fundamental theoretical concepts (e.g., sorting networks). We studied the fragile complexity of some of the fundamental problems and

revealed interesting behavior such as the large gap between the performance of deterministic and randomized algorithms for finding the minimum. We believe there are still plenty of interesting and fundamental problems left open. Below, we briefly review a few of them.

- The area of comparison-based algorithms is much larger than what we have studied. In particular, it would be interesting to study "geometric orthogonal problems" such as finding the maxima of a set of points, detecting intersections between vertical and horizontal line segments, kd-trees, axis-aligned point location and so on. All of these problems can be solved using algorithms that simply compare the coordinates of points.
- Is it possible to avoid using expander graphs to obtain simple deterministic algorithms to find the median or to sort?
- Is it possible to obtain a randomized algorithm that finds the median where the median suffers $O(1)$ comparisons on average? Or alternatively, is it possible to prove a lower bound? If one cannot show a $\omega(1)$ lower bound for the fragile complexity of the median, can we show it for some other similar problem?

Bibliography

[1] Peyman Afshani et al. "Fragile Complexity of Comparison-Based Algorithms". In: *CoRR* abs/1901.02857 (2019). arXiv: 1901.02857. URL: http://arxiv.org/abs/1901.02857.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. "An $O(n \log n)$ Sorting Network". In: *Proceedings of the 15th Symposium on Theory of Computation*. STOC '83. ACM, 1983, pp. 1–9. ISBN: 0-89791-099-0. URL: http://doi.acm.org/10.1145/800061.808726.

[3] M. Ajtai, J. Komlós, and E. Szemerédi. "Halvers and Expanders". In: *FOCS'92*. Pittsburgh, PN: IEEE Computer Society Press, Oct. 1992, pp. 686–692. ISBN: 0-8186-2900-2.

[4] M. Ajtai, J. Komlós, and E. Szemerédi. "Sorting in $c \log n$ parallel steps". In: *Combinatorica* 3.1 (Mar. 1983), pp. 1–19. ISSN: 1439-6912. URL: https://doi.org/10.1007/BF02579338.

[5] V. E. Alekseev. "Sorting Algorithms with minimum memory". In: *Kibernetika* 5.5 (1969), pp. 99–103.

[6] K. E. Batcher. "Sorting Networks and their Applications". English. In: *Proceedings of AFIPS Spring Joint Computer Conference* (1968), pp. 307–314.

[7] G. Stølting Brodal and M. C. Pinotti. "Comparator Networks for Binary Heap Construction". In: *Proc. 6th Scandinavian Workshop on Algorithm Theory*. Vol. 1432. LNCS. Springer Verlag, Berlin, 1998, pp. 158–168. ISBN: 978-3-540-64682-2. DOI: 10.1007/BFb0054364.

[8] V. Chvátal. *Lecture notes on the new AKS sorting network*. Tech. rep. DCS-TR-294. Department of Computer Science, Rutgers University, New Brunswick, NJ, Oct. 1992.

[9] R. Cole. "Parallel Merge Sort". In: *SIAM Journal on Computing* 17.4 (1988), pp. 770–785.

[10] M. Dowd et al. "The periodic balanced sorting network". In: *J. ACM* 36.4 (Oct. 1989), pp. 738–757.

[11] R. W. Floyd. "Algorithm 245: Treesort". In: *Commun. ACM* 7.12 (Dec. 1964), p. 701. ISSN: 0001-0782. URL: http://doi.acm.org/10.1145/355588.365103.

[12] M. T. Goodrich. "Zig-zag sort: a simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time". In: *STOC'14*. ACM, 2014, pp. 684–693. ISBN: 978-1-4503-2710-7. URL: http://dl.acm.org/citation.cfm?id=2591796.

[13] S. Janson. "Tail bounds for sums of geometric and exponential variables". In: *Statistics & Probability Letters* 135 (2018), pp. 1–6. ISSN: 0167-7152. DOI: https://doi.org/10.1016/j.spl.2017.11.017.

[14] S. Jimbo and A. Maruoka. "A Method of Constructing Selection Networks with $O(\log n)$ Depth". In: *SIAM Journal on Computing* 25.4 (1996), pp. 709–739.

[15] I. Parberry. "The pairwise sorting network". In: *Parallel Processing Letters* 2.2-3 (1992), pp. 205–211.

[16] B. Parker and I. Parberry. "Constructing sorting networks from k-sorters". In: *Information Processing Letters* 33.3 (Nov. 1989), pp. 157–162.

[17] M. S. Paterson. "Improved sorting networks with $O(\log N)$ depth". In: *Algorithmica* 5.1 (1990), pp. 75–92.

[18] N. Pippenger. "Selection Networks". In: *SIAM Journal on Computing* 20.5 (1991), pp. 878–887.

[19] V. R. Pratt. *Shellsort and Sorting Networks*. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York, 1972. ISBN: 0-8240-4406-1.

[20] S.Hoory, N. Linial, and A. Wigderson. "Expander Graphs and Their Applications". In: *BAMS: Bulletin of the American Mathematical Society* 43 (2006), pp. 439–561.

[21] J. I. Seiferas. "Sorting Networks of Logarithmic Depth, Further Simplified". In: *Algorithmica* 53.3 (2009), pp. 374–384.

[22] S. P. Vadhan. "Pseudorandomness". In: *Foundations and Trends in Theoretical Computer Science* 7.1-3 (2012), pp. 1–336.

[23] A. Yao and F. F. Yao. "Lower Bounds on Merging Networks". In: *J. ACM* 23.3 (1976), pp. 566–571.