High-Performance Graph Algorithms

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Moritz Graf von Looz-Corswarem

aus Köln

Tag der mündlichen Prüfung:	23.01.2019
Erster Gutachter:	Prof. Dr. Henning Meyerhenke
Zweiter Gutachter:	Prof. Dr. Peter Sanders

Ich versichere, diese Dissertation selbstständig angefertigt und alle benutzten Hilfsmittel vollständig angegeben zu haben. Was ich aus Arbeiten anderer und eigener Veröffentlichungen unverändert oder mit Änderungen entnahm, habe ich kenntlich gemacht. Die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis habe ich beachtet.

Acknowledgements

I thank my advisor Henning Meyerhenke, who took me on as a PhD student and guided me through the last years. He was never too busy to give advice and support, and to closely collaborate on papers we wrote together.

I also want to thank all those I was fortunate to collaborate with. Charilaos Tzovas, I will always remember our common struggle to finally get our partitioner working and scaling. Sören Laue and Mustafa Özdayi, thank you for the very fruitful collaboration in generating random hyperbolic graphs. I'm grateful to Thomas Brandes from SCAI, who never hesitated to support another unorthodox request of what I'd like to do with their library. Thank you also Andreas Longva, our collaboration was brief but very enjoyable.

Mario Wolter and Vadym Aizinger, thank you for collaborating with us on applications in need of graph partitioning. Working on applied problems in computer science is much better with actual applications. Thank you also Thomas Bohlen, Gerald Eisenberg, Eliakim Schünemann, Thomas Soddemann, Tilman Metz and Rüdiger Zehn, I enjoyed our collaboration within the WAVE project.

Thank you Manuel Penschuck, Sebastian Lamm, Christian Schulz and Peter Sanders for our joint work in combining different techniques for faster graph generation.

For advice on the not always intuitive world of non-Euclidean geometry, I thank Roman Prutkin. For advice on graph partitioning, parallel scalability and profiling, I thank Michael Axtmann and (again) Christian Schulz.

A big thank you to our group: Elisabetta Bergamini, Roland Glantz and Christian Staudt, who accompanied me in the beginning of my PhD and Maria Predari, Alexander van der Grinten and Eugenio Angriman, who did so at the end.

Ralf Kölmel, thank you for making sure all our infrastructure was running so smoothly that we could keep believing the abstractions computer scientists like.

I thank Alex Meier, Patrick Lahr, Maria Predari and Charlotte Mertz for proofreading. You saved me from more than one blunder in probability theory, graph partitioning and basic grammar.

For tremendous support, both logistical and emotional, I thank my parents and my brother Felix.

Zusammenfassung

Verschiedenste Phänomene, in denen die Struktur von Beziehungen im Vordergrund steht, lassen sich sinnvoll als Graphen darstellen. Bekannte Beispiel sind soziale Netzwerke, die Infrastruktur des Internets, Warenkataloge und Kundenvorlieben, Straßennetzwerke und Nahrungsnetze in Ökosystemen. In wissenschaftlichen Simulationen werden oft Teilaufgaben als Knoten modelliert und ihre Datenabhängigkeiten als Kanten. Mit Methoden aus der Graphentheorie lässt sich dann die parallele Kommunikation optimieren.

Viele Graphen haben eine zugrundeliegende Geometrie. Diese ist explizit bei Graphen, die ein geometrisches Phänomen modellieren, zum Beispiel Gitternetze aus numerischen Simulationen, oder Sensornetzwerke mit geographischen Koordinaten. Manch andere Graphen ohne geometrische Informationen folgen jedoch einer *versteckten* Geometrie und in einer geeigneten Einbettung würden Graphdistanzen und geometrische Distanzen zusammenfallen.

Zunehmende Aufmerksamkeit erfahren *komplexe Netzwerke*, Graphen mit üblicherweise kleinem Durchmesser, einer heterogenen Struktur und einer Gradverteilung, in der es verhältnismäßig viele Knoten mit überdurchschnittlich vielen Nachbarn gibt. Solche Graphen werden auch als *skalenfrei* oder *Kleine-Welt-Netzwerke* bezeichnet. Die bekanntesten Beispiele für komplexe Netzwerke sind soziale Netzwerke, das Netz der über Hyperlinks verbundenen Webseiten sowie Kommunikationsnetzwerke. Viele davon sind im Bereich von hunderten Millionen bis Milliarden von Knoten und Milliarden von Kanten. Generative Modelle für komplexe Netzwerke sind hierbei nicht nur relevant dafür, Erkenntnisse über Entwicklung realer Netzwerke zu gewinnen, sondern auch für die Erzeugung von Testdaten für die Entwicklung neuer Analysealgorithmen. Auch wenn reale Testdaten zur Verfügung stehen, sind sie oft durch ihre Größe umständlich in der Handhabung. Ein generatives Modell, aus dem Zufallsgraphen effizient gezogen werden können, kann Entwicklungszyklen deutlich beschleunigen.

Während z.B. Straßennetzwerke der sphärischen Geometrie folgen, in der sie gebaut wurden, gibt es einige Hinweise darauf, dass die passende Geometrie für komplexe Netzwerke weder euklidisch noch sphärisch ist, sondern hyperbolisch. Basierend darauf haben Krioukov et al. *hyperbolische Zufallsgraphen* definiert, ein generatives Graphenmodell für komplexe Netzwerke. Ein Graph darin wird generiert, in dem Knoten zufällig in einer Kreisscheibe in der hyperbolischen Ebene verteilt werden, die Wahrscheinlichkeit für eine Kante $\{u, v\}$ wird durch die hyperbolische Distanz zwischen u und v bestimmt. Das Modell wird durch die Verteilungen der Knotenpositionen und Kantenwahrscheinlichkeiten parametrisiert. Eine bedeutende Unterklasse sind *hyperbolische Schwellwertgraphen*, in denen zwei Knoten genau dann verbunden sind, wenn ihre Distanz geringer ist als ein Schwellwert. In diesem Modell ergeben sich einige gewünschte Eigenschaften von komplexen Netzwerken direkt aus der Geometrie, was eine theoretische Analyse erleichtert.

Leider hat ein naiver Generierungsalgorithmus eine quadratische Laufzeit, was die Generierung von Netzwerken interessanter Größe stark erschwert.

Im ersten Teil meiner Dissertation stelle ich vier neue Generierungsalgorithmen für hyperbolische Zufallsgraphen vor. Durch die Projektion der hyperbolischen Ebene in die euklidische Geometrie der Poincaré-Kreisscheibe können wir existierende geometrische Datenstrukturen adaptieren. Unser erster Algorithmus basiert auf einem polaren Quadtree, durch den wir überflüssige Distanzberechnungen vermeiden und eine Laufzeit von $\mathcal{O}((n^{3/2}+m)\log n)$ (mit hoher Wahrscheinlichkeit) erreichen können. Dies war der erste Generierungsalgorithmus für hyperbolische Schwellwertgraphen mit subquadratischer Laufzeit. In Experimenten war unsere Implementierung etwa drei Größenordnungen schneller als eine Referenzimplementierung des quadratischen Algorithmus.

Für den zweiten Algorithmus haben wir die Quadtreestruktur erweitert, um auch generelle hyperbolische Zufallsgraphen zu erzeugen, in denen alle Kanten probabilistisch sind. Da alle Kanten eine positive Wahrscheinlichkeit haben, hätte die explizite Berechnung jeder Kantenwahrscheinlichkeit wieder eine quadratische Komplexität. Wir betrachten stattdessen für jeden Knoten die Lücken zwischen seinen Nachbarn in einer Liste aller Knoten und ziehen die Länge dieser Lücken zufällig. Da die Wahrscheinlichkeit für jede Kante von der Distanz ihrer Knoten abhängt, benutzen wir die Quadtreestruktur um Schranken für die Abstände und damit Kantenwahrscheinlichkeiten zu bestimmen. Um die Balance zwischen Qualität der Schranken und dem Aufwand ihrer Berechnung zu halten, aggregieren wir Teilbäume in *virtuelle Blattzellen*, wenn in ihnen nur wenige Nachbarn erwartet werden. Damit können wir sowohl die Anzahl der Distanzberechnungen als auch die Anzahl der besuchten Quadtreezellen beschränken und erhalten mit $\mathcal{O}((n^{3/2}+m)\log n)$ (mhW) die gleiche asymptotische Zeitkomplexität auch für generelle hyperbolische Zufallsgraphen.

Das Szenario, in dem eine zufällige Teilmenge eines geometrischen Datensatzes gewünscht ist und die Wahrscheinlichkeit jedes Punktes, in der Ergebnismenge enthalten zu sein, von der Distanz zu einem Anfragepunkt abhängt, tritt nicht nur bei der Erzeugung hyperbolischer Zufallsgraphen auf. Wir definieren *probabilistische Nachbarschaftsanfragen* und erweitern unseren Algorithmus für euklidische Datensätze.

Unser zweiter Algorithmus zur Erzeugung von hyperbolischen Schwellwertgraphen teilt die verwendete Kreisscheibe in der hyperbolischen Ebene in ringförmige Bänder auf. Für jeden Punkt können für jedes Band die Winkelkoordinaten möglicher Nachbarn mittels des hyperbolischen Kosinussatzes beschränkt werden, was die Anzahl der zu prüfenden Kandidaten stark reduziert. Das führt zu einer Komplexität von $\mathcal{O}(n\log^2 n + m)$ und einer Verringerung der experimentellen Laufzeiten um eine weitere Größenordnung.

Schließlich haben wir auch diese Datenstruktur auf generelle hyperbolische Zufallsgraphen erweitert und einen Generierungsalgorithmus mit Zeitkomplexität $\mathcal{O}(n\log^2 n + m)$ angegeben.

Der zweite Teil meiner Dissertation betrachtet in vielen Aspekten das Gegenteil des ersten. Anstatt Graphen mit latenter Geometrie sind es welche mit expliziten Knotenkoordinaten. Anstatt Graphen zu generieren, zerteile ich sie. Das *Graphpartitionierungsproblem* besteht darin, zu einem gegebenen Graphen und einer Zielfunktion eine Partition der Knotenmenge zu finden, so dass alle Teilmengen ähnlich groß sind und die Zielfunktion optimiert wird. Zu den vielen Anwendungen gehört die Parallelisierung von Simulationen im wissenschaftlichen Rechnen bei gleichzeitiger Balancierung der Last und Minimierung der Kommunikation. Andere Anwendungen sind die Aufteilung von Graphen zur Parallelisierung von Netzwerkanalysealgorithmen, Vorberechnungen von Datenstrukturen zur Routenplanung sowie viele weitere. Da das Graphpartitionierungsproblem NP-vollständig und schwer zu approximieren ist, werden in der Praxis Heuristiken verwendet.

Unseren ersten Graphpartitionierer haben wir für eine Anwendung in der Quantenchemie entwickelt. Elektronendichten zu berechnen ist nötig um Interaktionen zwischen Proteinen korrekt vorherzusagen, aber skaliert quadratisch mit der Größe des Proteins. Diese Skalierung, zusammen mit hohen konstanten Faktoren, limitiert die exakte Berechnung effektiv auf Proteine mit höchstens einigen hundert Aminosäuren. Diese Einschränkung kann durch Aufteilung größerer Proteine in Teilstücke umgangen werden, diese Ansätze sind als *Subsystemmethoden* bekannt. Dabei aber werden Interaktionen von Aminosäuren in verschiedenen Teilstücken vernachlässigt, diese Vernachlässigung führt zu einem Fehler im Ergebnis. Wir modellieren das Szenario als Partitionierungsproblem: Im zu partitionierenden *Proteingraphen* repräsentiert jeder Knoten eine Aminosäure, jede Kante eine Interaktion zwischen zwei Aminosäuren und jedes Kantengewicht den zu erwartenden Fehler, der durch die Vernachlässigung dieser Interaktion verursacht würde. Eine Menge aus Subsystemen mit minimalem Fehler ist dann äquivalent zu einer Partition des Proteingraphen mit minimalem Kantenschnitt. Hierbei wird durch die Hauptkette eines Proteins bereits eine Reihenfolge und implizite Geometrie definiert. Weiterhin muss, um chemisch sinnvoll anwendbar zu sein, eine Partition weitere Bedingungen erfüllen. Wir haben die Multi-Level-Heuristik zusammen mit lokaler Suche nach Fiduccia und Mattheyses so angepasst, dass sie die zusätzlichen Bedingungen erfüllt. Für ein eingeschränktes Szenario mit kleinerem Lösungsraum bieten wir einen optimalen Algorithmus basierend auf dynamischer Programmierung. Wir erreichen insgesamt einen um im Durchschnitt 13.5% geringeren Kantenschnitt gegenüber der naiven Lösung, die bisher in der Praxis eingesetzt wurde.

Unser zweiter Graphpartitionierer ist für geometrische Meshes aus numerischen Simulationen. Die Größe der Simulationen, bestehend aus Milliarden von Gitterpunkten, erfordert eine Parallelisierung nicht nur der Anwendung, sondern auch der Partitionierung selbst. Die Multi-Level-Heuristik erzielt zwar im Allgemeinen Partitionen mit hoher Qualität, aber bei geringer Skalierbarkeit. Aufgrund der großen Halos in unserer Zielanwendung spielt auch die Form der partitionierten Blöcke eine Rolle, konvexe Formen erfordern weniger Kommunikation. Wir erweitern deshalb Lloyds Algorithmus für das bekannte k-Means-Problem, welches konvexe Cluster erzeugt. Für den Einsatz zur Graphpartitionierung fügen wir einen Balancierungsschritt hinzu und adaptieren mehrere geometrische Optimierungen um eine schnelle Laufzeit und parallele Skalierbarkeit zu erreichen. Lloyds Algorithmus in seiner klassischen Form ist anfällig gegenüber lokalen Minima, das Ergebnis hängt stark von den initialen Zentren ab; Verfahren zur Garantie guter Zentren haben eine für unseren Fall ungeeignet hohe Laufzeit. Wir sortieren deshalb die Eingabepunkte

entlang einer raumfüllenden Kurve, und selektieren anschließend initiale Zentren äquidistant unter den sortierten Punkten. Unsere Implementierung skaliert zu zehntausenden Prozessen und Milliarden von Knoten und berechnet eine Partition innerhalb von Sekunden. Im Vergleich zum Stand der Technik anderer schneller geometrischer Partitionierer berechnet unsere Methode Partitionen mit einem im Schnitt 10-15% geringeren Kommunikationsvolumen. Das zeigt sich auch in kürzerer Kommunikationszeit in Benchmarks zu verteilter Matrix-Vektor-Multiplikation.

Abstract

Ever since Euler took a stroll across the bridges of a city then called Königsberg, relationships of entities have been modeled as graphs. Being a useful abstraction when the structure of relationships is the significant aspect of a problem, popular uses of graphs include the modeling of social networks, supply chain dependencies, the internet, customer preferences, street networks or the who-eats-whom (aka food network) in an ecosystem.

In parallel computing, the assignment of sub-tasks to processes can massively influence the performance, since data dependencies between processes are significantly more expensive than within them. This scenario has been profitably modeled as a graph problem, with sub-tasks as vertices and their communication dependencies as edges.

Many graphs are governed by an underlying geometry. Some are derived directly from a geometric object, such as street networks or meshes from spatial simulations. Others have a *hidden* geometry, in which no explicit geometry information is known, but the graph structure follows an underlying geometry. A suitable embedding into this geometry would then show a close relationship between graph distances and geometric distances.

A subclass of graphs enjoying significant attention are *complex networks*. Though hard to define exactly, they are commonly characterized by a low diameter, heterogeneous structure, and a skewed degree distribution, often following a power law. The most famous examples include social networks, the hyperlink network and communication networks. Development of new analysis algorithms for complex networks is ongoing. Especially since the instances of interest are often in the size of billions of vertices, fast analysis algorithms and approximations are a natural focus of development. To accurately test and benchmark new developments, as well as to gain theoretical insight about network formation, generative graph models are required: A mathematical model describing a family of random graphs, from which instances can be sampled efficiently. Even if real test data is available, interesting instances are often in the size of terabytes, making storage and transmission inconvenient.

While the underlying geometry of street networks is the spherical geometry they were built in, there is some evidence that the geometry best fitting to complex networks is not Euclidean or spherical, but hyperbolic. Based on this notion, Krioukov et al. proposed a generative graph model for complex networks, called *Random Hyperbolic Graphs*. They are created by setting vertices randomly within a disk in the hyperbolic plane and connecting pairs of vertices with a probability depending on their distance. An important subclass of this model, called *Threshold Random Hyperbolic Graphs* connects vertices exactly if the distances between vertices is below a threshold. This model has pleasant properties and has received considerable attention from theoreticians. Unfortunately, a straightforward generation algorithm has a complexity quadratic in the number of nodes, which renders it infeasible for instances of more than a few million vertices.

We developed four faster generation algorithms for random hyperbolic graphs: By projecting hyperbolic geometry in the Euclidean geometry of the Poincaré disk model, we are able to use adapted versions of existing geometric data structures. Our first algorithm uses a polar quadtree to avoid distance calculations and achieves a time complexity of $\mathcal{O}((n^{3/2} + m) \log n)$ whp. – the first subquadratic generation algorithm for threshold random hyperbolic graphs. Empirically, our implementation achieves an improvement of three orders of magnitude over a reference implementation of the straightforward algorithm.

We extend this quadtree data structure further for the generation of general random hyperbolic graphs, in which all edges are probabilistic. Since each edge has a non-zero probability of existing, sampling them by throwing a biased coin for each would again cost quadratic time complexity. We address this issue by sampling jumping widths within leaf cells and aggregating subtrees to *virtual leaf cells* when the expected number of neighbors in them is less than a threshold. With this tradeoff, we bound both the number of distance calculations and the number of examined quadtree cells per edge, resulting in the same time complexity of $\mathcal{O}((n^{3/2} + m) \log n)$ also for general random hyperbolic graphs.

We generalize this sampling scenario and define *Probabilistic Neighborhood Queries*, in which a random sample of a geometric point set is desired, with the probability of inclusion depending on the distance to a query point. Usable to simulate probabilistic spatial spreading, we show a significant speedup on a proof of concept disease simulation.

Our second algorithm for threshold random hyperbolic graphs uses a data structure of concentric annuli in the hyperbolic plane. For each given vertex, the positions of possible neighbors in each band can be restricted with the hyperbolic law of cosines, leading to a much reduced number of candidates that need to be checked. This yields a reduced time complexity of $\mathcal{O}(n \log^2 n + m)$ and a further order of magnitude in practice for graphs of a few million vertices.

Finally, we extend also this data structure to general random hyperbolic graphs, with the same time complexity for constant parameters.

The second part of my thesis is in many aspects the opposite of the first. Instead of a hidden geometry, I consider graphs whose geometric information is explicit. Instead of using it to generate graphs, I use their geometric information to decide how to cut them into pieces. Given a graph, the *Graph Partitioning Problem* asks for a disjoint partition of the vertex set so that each subset has a similar number of vertices and some objective function is optimized. Its many applications include parallelizing a computing task while balancing load and minimizing communication, dividing a graph into blocks as preparation for graph analysis tasks or finding natural cuts in street networks for efficient route planning. Since the graph partitioning problem is NP-complete and hard to approximate, heuristics are used in practice.

Our first graph partitioner is designed for an application in quantum chemistry. Computing electron density fields is necessary to accurately predict protein interactions, but the required time scales quadratically with the protein's size, with punitive constant factors. This effectively restricts these density-based methods to proteins of at most a few hundred

amino acids. It is possible to circumvent this limitation by computing only parts of the target protein at a time, an approach known as *subsystem quantum chemistry*. However, the interactions between amino acids in different parts are then neglected; this neglect causes errors in the solution. We model this problem as partitioning a *protein graph*: Each vertex represents one amino acid, each edge an interaction between them and each edge weight the expected error caused by neglecting this interaction. Finding a set of subsets with minimum error is then equivalent to finding a partition of the protein graph with minimum edge cut. The requirements of the chemical simulations cause additional constraints on this partition, as well as an implied geometry by the protein structure. We provide an implementation of the well-known multilevel heuristic together with the local search algorithm by Fiduccia and Mattheyses, both adapted to respect these new constraints. We also provide an optimal dynamic programming algorithm for a restricted scenario with a smaller solution space. In terms of edge cut, we achieve an average improvement of 13.5% against the naive solution, which was previously used by domain scientists.

Our second graph partitioner targets geometric meshes from numerical simulations in the scale of billions of grid points, parallelized to tens of thousands of processes. In general purpose graph partitioning, the multilevel heuristic computes high-quality partitions, but its scalability is limited. Due to the large halos in our targeted application, the shape of the partitioned blocks is also important, with convex shapes leading to less communication. We adapt the well-known k-means algorithm, which yields convex shapes, to the partitioning of geometric meshes by including a balancing scheme. We further extend several existing geometric optimizations to the balanced version to achieve fast running times and parallel scalability. The classic k-means algorithm is highly dependent on the choice of initial centers. We select initial centers among the input points along a space-filling curve, thus guaranteeing a similar distribution as the input points. The resulting implementation scales to tens of thousands of processes and billions of vertices, partitioning them in seconds. Compared to previous fast geometric partitioners, our method provides partitions with a 10-15% lower communication volume and also a corresponding smaller communication time in a distributed SpMV benchmark.

Contents

1. Introduction

Surprisingly many phenomena in computer science, engineering and the natural sciences can sensibly be modeled as graphs. The first known example is Euler's analysis of the bridges of Königsberg, abstracting away the geometry to focus on the connections. The best-known current examples are probably social networks, in which vertices represent people (or bots, nowadays) and edges represent interactions and relationships. Assembling a genome sequence from scattered fragments can be solved as an instance of the *longest path problem* [133]. Other common areas include operations research (power grids, logistics networks) or biology (protein interaction networks) and the internet (router connections, PageRank).

Numerical simulations in engineering or climate research often discretize the simulation domain into a geometric mesh, either to model phenomena as partial differential equations or for use in explicit time-stepping models.

In several graph problems, geometric information is helpful if present or desirable if absent. In the router network forming the internet, artificial coordinates from a *graph embedding* can help in routing, reducing the need for explicit routing tables. In early approaches to route planning in street networks, Dijkstra's algorithm to find shortest paths was accelerated by using coordinates to steer it into the right direction. Finally, in graph drawing, planar coordinates for a given graph are desired to give an intuitive visual impression of its structure.

This thesis considers two algorithmic problems concerning geometric graphs: Following the notion that many complex networks have a *hidden hyperbolic geometry*, how to quickly *generate* random graphs from this geometry. Second, given a graph with *explicit geometry*, how to partition it into a number of disjoint, equal-sized blocks so that some performance metric (usually the edge cut) is minimized. This is commonly called the graph partitioning problem.

1.1 Motivation

1.1.1 Generating Random Hyperbolic Graphs

Generative network models play a central role in many complex network studies for several reasons: Real data often contains confidential information; it is then desirable to work on similar synthetic networks instead. Quick testing of algorithms requires small test cases, while benchmarks and scalability studies need bigger graphs. Graph generators can provide data at different user-defined scales for this purpose. Also, transmitting and storing a generative model and its parameters is much easier than doing the same with a gigabyte-sized network. A central goal for generative models is to produce networks which replicate relevant structural features of real-world networks [35]. Finally, generative models are an important theoretical part of network science, as they can improve our understanding of network formation.

Random hyperbolic graphs (RHGs), introduced by Krioukov et al. [93], are a very promising graph family in this context: They yield a provably high clustering coefficient (a measure for the frequency of triangles) [63], small diameter [24] and a power-law degree distribution with adjustable exponent. This family of graphs has been analyzed well theoretically [23, 63, 24, 89] and Krioukov et al. [93] show that complex networks have a natural embedding in hyperbolic geometry. A graph is generated by randomly placing vertices in the hyperbolic plane and connecting each pair with a probability depending on their distance. An important special case are *threshold random hyperbolic graphs*, in which a pair is connected if their distance is below a threshold. Unfortunately, a straightforward generation algorithm has a quadratic time complexity, rendering it infeasible for large graphs.

1.1.2 Partitioning Geometric Graphs

Numerical simulations in scientific computing require parallelization due to their increasing size. Since communication between processes is often several orders of magnitude slower than computation, minimizing communication while balancing the computational load is a frequent necessity for good performance. This is a common application for graph partitioning, with each vertex representing a subtask and each edge representing a communication dependency between subtasks.

Frequently, though, applications have additional constraints for permissible partitions, requiring new algorithmic developments. We consider two application scenarios: 1. Subsystem quantum chemistry, for which protein graphs are partitioned to yield feasible running times, and the edge weights model the expected approximation error. Several new constraints induced by the chemical simulations require adaptations of existing graph partitioning algorithms or new developments. 2. Parallel scientific computing using geometric meshes. In this scenario the edge weights model data flow between neighboring grid points, resulting in inter-process communication if the edge is cut. Due to the input size, also the partitioning process itself must happen in parallel and scale to high numbers of processes.

1.2 Outline and Contribution

After this introduction (Chapter 1) follows a chapter on basic notions of graphs, notation and hyperbolic geometry (Chapter 2). The two main algorithmic problems I consider in this thesis – generation of hyperbolic random graphs and partitioning of geometric graphs – are connected by the common theme of graphs and geometry, but sufficiently different that little of the related literature is related to both. Thus, the discussion of related work is included in the respective chapters.

We provide several generation algorithms for hyperbolic random graphs in Chapter 3. Our main data structure is a polar quadtree on the Poincaré disk (Section 3.3). The first subquadratic algorithm (Section 3.4) uses this polar quadtree and is in practice three orders of magnitude faster than the existing implementation available at the time. We prove a time complexity of $\mathcal{O}(n^{3/2} \log n)$ with high probability. (We say that a statement holds *with high probability* (whp.) if it has a probability of at least $1-1/n$ for n sufficiently large.) The algorithm was presented at ISAAC'15 [164] and is restricted to the common special case of *threshold random hyperbolic graphs*.

We remove this restriction with our second algorithm (Section 3.5), also using a polar quadtree and nested probabilistic sampling methods. By aggregating subtrees of the quadtree to *virtual leaf cells*, we bound both the number of distance calculations and quadtree visits, and prove the same time complexity of $\mathcal{O}(n^{3/2} \log n)$ whp. We generalize this sampling method also for Euclidean spatial datasets, where it can be used for faster simulations of probabilistic spreading processes.This work was first presented at IWOCA'16 [161].

Our third algorithm for hyperbolic random graphs deals with the faster generation of threshold random hyperbolic graphs (Section 3.6). It divides the disk area in the hyperbolic plane, from which vertex positions are sampled, into concentric annuli. This results in an (empirical) running time of $\mathcal{O}(n \log^2 n)$ and another order of magnitude improvement in practice, presented at HPEC'16 [163].

We finally present a fourth algorithm (Section 3.7) combining the concentric annuli and probabilistic sampling methods, achieving a time complexity of $\mathcal{O}(n \log^2 n)$ also for general random hyperbolic graphs and also a significant improvement in practice. This extension has not yet been published elsewhere.

In addition to algorithms for the generation of static graphs, we also define a model for dynamic random hyperbolic graphs with gradual node change. Since the quadtree data structure can handle node updates efficiently, the time complexity of updating the neighborhood of a node v is $\mathcal{O}(\sqrt{n} \log n + \deg v)$. This combination, along with the theoretical background to the nested probabilistic sampling models, was published 2018 in the Journal of Experimental Algorithms [162].

We present our partitioning algorithms and results in Chapter 4, together with a comparison to related graph partitioners. The first partitioning application targets protein graphs for electron density calculations (Section 4.3). For the special case of *main chain partitioning*, in which each block must be continuous on the main chain of the protein, we provide a dynamic programming algorithm and prove its optimality. Compared to naively cutting the protein graph at every X amino acids, our methods achieve an average reduction of 13% in terms of edge cut. This result was presented at SEA'16 [166].

For the parallel partitioning of large, geometric meshes (Section 4.4), we present and implement a scalable, balanced version of Lloyd's algorithm for the k-means problem. With several geometric optimizations and careful seeding of initial centers, our implementation converges in seconds for tens of thousands of blocks and billions of input points. It also scales to tens of thousands of processes, independently of the number of blocks. The resulting partitions have a $10 - 15\%$ lower communication volume than other current geometric partitioners. This algorithm and experimental results were presented at ICPP'18 [165]. For faster redistribution of application data, we reorganized the communication patterns in the LAMA library, leading to a reduction of redistribution time by two orders of magnitude. This was not a new algorithmic development, but resulted from a careful analysis of the previous communication flow.

We conclude with Chapter 5.

2. Preliminaries

2.1 Graphs

A graph $G = (V, E)$ in its basic form consists of a set of vertices V and a set of edges $E \subseteq V \times V$.

Common extensions include *node weights* and *edge weights*, defined as functions $\gamma : V \rightarrow \mathbb{R}$ and $\omega : E \rightarrow \mathbb{R}$. A graph is called *undirected* if for every edge (u, v), an opposite edge (v, u) (with the same weight) exists. We use the set notation $\{u, v\}$ for undirected graphs. Two vertices u and v are called *adjacent* if an edge (u, v) exists between them.

The *neighborhood* of a vertex u is the set of all vertices adjacent to u:

$$\Gamma(u) = \{v | \{u, v\} \in E\}$$

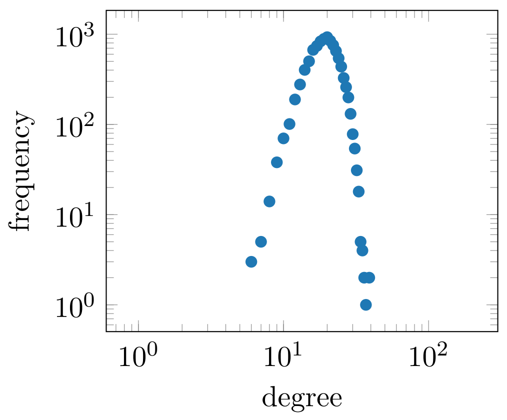
The degree $\deg(u)$ of a vertex u is the cardinality of its neighborhood $\Gamma(u)$. For directed graphs, the *in-degree* of a vertex u is the number of incoming edges (v, u) and the *out-degree* is the number of outgoing edges (u, v). However, all further graphs occurring in this thesis are undirected.

The names "node" and "vertex" are sometimes used interchangeably. In the context of graphs, we only use "vertex" and reserve the name "node" for computing units in parallel processing.

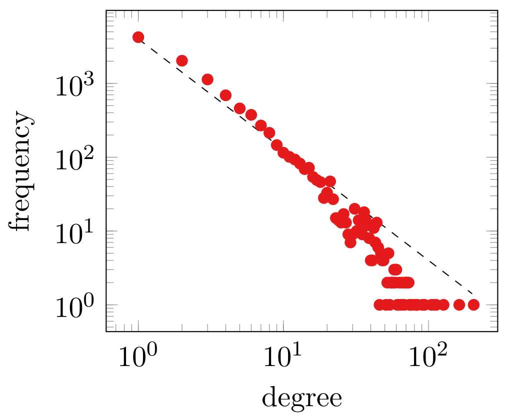
A similar proliferation of names exists with the terms "graph" and "network". Sometimes the distinction is made that "graph" is used for the mathematical entity and "network" for representations of real phenomena. We use "graph", except for when discussing "complex networks", a term which has been established in its own right.

Subgraphs

Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the graph G_2 is called a subgraph of G_1 if it only contains vertices and edges also present in G_1: $V_2 \subseteq V_1$ and $E_2 \subseteq E_1 \cap (V_2 \times V_2)$.



(a) Degree distribution of an Erdős-Rényi random graph, in which the existence of each edge is sampled independently at random. The observed frequencies follow a binomial distribution, with sharply falling tails after the peak.



(b) Degree distribution of the largest component of the PGP web of trust. Vertices represent keys, edges represent signatures. No "typical" degree exists, as many vertices have far more than average.

Figure 2.1: Comparison of degree distributions.

Degree Distribution

The distribution of degrees of a graph is called its *degree distribution* and can give insight into its nature. In regular meshes, vertices have very similar degrees. In a graph where the endpoints of each edge are sampled uniformly at random, the degrees follow a binomial distribution. It is common in social networks that many hubs exists with widely more connections than average. Thus, the degree distribution is one of many properties commonly used to classify graphs [17]. Figure 2.1 compares the degree distributions of a simple random graph and a small social network, the web of trust of PGP signatures. The distribution in Figure 2.1b follows a *power-law*: The number of vertices with degree k is proportional to $k^{-\gamma}$ for a fixed exponent γ:

$$\#\{v | \deg(v) = k\} \propto k^{-\gamma}. \tag{2.1}$$

For the degrees of the PGP web of trust in Figure 2.1b, $\gamma := 1.5$ gives a reasonable fit.

Barabási and Réka [16] propose *preferential attachment* processes as possible explanations of power-law degree distributions. If new vertices in a growing graph attach to existing vertices with a probability proportional to their existing degree (a "rich-get-richer dynamic"), the degrees of the resulting graph will follow a power-law distribution. Newman shows that scientific collaboration networks follow such a *preferential attachment* dynamic empirically [121].

The degree distribution has effects on network processes. In disease simulations, for example, networks with power-law degree distributions are harder to vaccinate. In networks with binomially distributed degrees, vaccinating a certain fraction of vertices uniformly at random stops epidemics in expectation. This strategy does not work for networks with power-law degree distributions of similar densities, as the large number of well-connected hubs enables infections of the remaining parts should any of them be left unvaccinated [59].

Other natural phenomena such as volcano eruptions or personal wealth also have effect sizes distributed according to power-laws [34, 152].

Paths and Distances

A distinct sequence of vertices $v_1, v_2, \dots v_k \subseteq V$ is called a *path* of length k if for each pair v_i, v_{i+1} in the sequence, an edge $\{v_i, v_{i+1}\}$ exists in E. A *shortest* path between v_1 and v_k is a path so that no other path from v_1 to v_k with a shorter length exists. The *distance* between two vertices s and t is the length of the shortest path connecting them. If no such path exists, their distance is considered to be infinite. To distinguish it from geometric distances, the length of a shortest path between vertices s and t is also called the *graph-theoretic distance* of s to t, we denote it with $\mathrm{dist}_G(s,t)$. The *diameter* of a graph is the largest distance occurring in it:

$$\mathrm{diam}(G) = \max_{s,t \in V} \mathrm{dist}_G(s,t) \tag{2.2}$$

The diameter of a graph thus also determines how long it takes for an information or epidemic to spread through it.

Cuts and Partitions

A *cut* of a graph $G = (V, E)$ is a division of the vertex set into two subsets S and $V \setminus S$. Edges with one endpoint in S and the other in $V \setminus S$ are called *cut edges.* The *weight* of a cut is the sum of weights of cut edges.

A *partition* is a generalized cut: A k-partition of a graph is a division of the vertex set into k disjoint subsets. The *edge cut* of a partition is the sum of all weights of edges whose endpoints are in different subsets. A vertex subset in a partition is also called a *block.*

A *cluster* in a graph is also a subset of its vertices and a *clustering* seeks to divide a given graph into a set of clusters that reflect its structure. The main difference to a partitioning problem is that the number of desired subsets is often not given in advance, but part of the exploratory analysis performed with a clustering.

Given a graph $G = (V, E)$ and a partition Π, the corresponding *block graph* G_Π contains one vertex for each subset in Π. In the block graph, an edge (v_i, v_j) between the vertices representing the blocks V_i and V_j exists if an edge connects the blocks V_i and V_j in the original graph.

Communities

Similarly, a *community* in a graph is loosely defined as a group of vertices which has many internal edges and few external edges. Newman and Girvan [119] quantify this idea by introducing *modularity*, a measure that compares the actual number of internal edges with the expected number in a random graph of equal degree distribution. Finding a partition of a graph that maximizes modularity is $\mathcal{NP}$-hard [29].

Geometric Embedding

An *embedding* of a graph into a geometry $\mathbb{B}$ assigns coordinates to each vertex v. The *distortion* of an embedding is the difference between the geometric and graph-theoretic distances between its vertices. Graphs with a small diameter tend to be harder to embed. A *graph drawing* has a related, but different objective: The graph layout should give an intuitive visual representation of its structure, mostly in two or three dimensions. Often, edge crossings should be avoided.

2.1.1 Complex Networks

A network with non-trivial structure is called a *complex network*[168, 87]. Different definitions exist, they agree in that social networks, web graphs and most infrastructure networks are examples of complex networks, while regular meshes or simple random graphs are not. Commonly named criteria [120] are a skewed degree distribution, often following a power-law, a small diameter (commonly called the small-world-phenomenon [6, 168]), high clustering and a hierarchical structure [35].

Travers and Milgram experimentally measured the average path lengths of random messages in US acquaintanceship networks [158]. Due to the surprisingly small result – six – these experiments are now known as the *small-world experiments*.

2.2 Hyperbolic Geometry

The most well-known and commonly used geometry is Euclidean, but it is not the only possible one.[1]

On the surface of a sphere, spherical geometry applies, which is curved *positively*. When drawing triangles on a globe, they have an interior angle sum of more than 180° and appear "fat". A projection from a globe to a (Euclidean) map always leads to distortions. This is not due to a different number of dimensions, as both a sphere surface and a map have two dimensions. Different map projections preserve angles, distances or areas, but never all three.[2]

Given a line l and a point p not on l in Euclidean geometry, there is exactly one line l' that contains p and is parallel to l [53]. This is called the *parallel postulate* and geometers tried for centuries to derive it from the other Euclidean axioms. After unsuccessful attempts of a proof by negation[3], Bolyai [25] defined a consistent hyperbolic geometry, in which the parallel postulate does not hold.

[1] Lovecraft, a horror author from the early 20th century, used "non-Euclidean" as synonym for alien and frightening. [102]

[2] The Mercator projection, the most common projection for world maps, preserves angles but distorts distances and areas.

[3] "You must not attempt this approach to parallels. I know this way to the very end. I have traversed this bottomless night, which extinguished all light and joy in my life. I entreat you, leave the science of parallels alone...Learn from my example" – in a letter from Farkas Bolyai to his son János Bolyai. Fortunately for us, he didn't listen.

Property	Euclidean	spherical	hyperbolic
Curvature $k = -\zeta^2$	0	> 0	< 0
Triangle angle sum	$180°$	$> 180°$	$< 180°$
Circle Length	πr	-	$2\pi\zeta \sinh(r/\zeta) \approx e^r$
Circle Area	πr^2	-	$2\pi\zeta^2 (\cosh(r/\zeta) - 1) \approx e^r$

Table 2.1: Properties of Euclidean, Spherical and Hyperbolic space. These are also called the three *isotropic* spaces, as distances in them are independent of absolute positions and thus invariant under translation.

Hyperbolic geometry has *negative* curvature: Triangles have an angle sum of less than 180° and appear "thin". Even more than spherical geometry, hyperbolic space cannot be projected to Euclidean space without distortions. Also in contrast to spherical geometry, a hyperbolic space of low dimension d cannot easily be embedded into a Euclidean space with dimension $d+1$[99, 71]. An overview of properties is shown in Table 2.1. Among them is that in Euclidean geometry, the circumference of a circle grows linearly with its radius, while the area grows quadratically. In hyperbolic geometry, both grow exponentially with the radius. For a curvature K, the term $\frac{1}{-\sqrt{K}}$ occurs in many geometric equations, thus we define $\zeta = \frac{1}{-\sqrt{K}}$ as a shorthand.

In a tree of constant degree b, the number of vertices with distance r to the root also grows exponentially with r, as $(b+1)b^{r-1}$. The number of vertices with distance at most r scales with $[(b+1)b^r - 2]/(b-1)$. When drawing a tree of constant degree b into Euclidean space, the space per vertex thus decreases with increasing distance to the root, as the area to draw them increases only quadratically. This is not the case for hyperbolic space, where circle area increases exponentially as well; when setting $\zeta = \ln b$, it even increases exactly with b^r. This has led to the idea of hyperbolic space as "continuous trees".

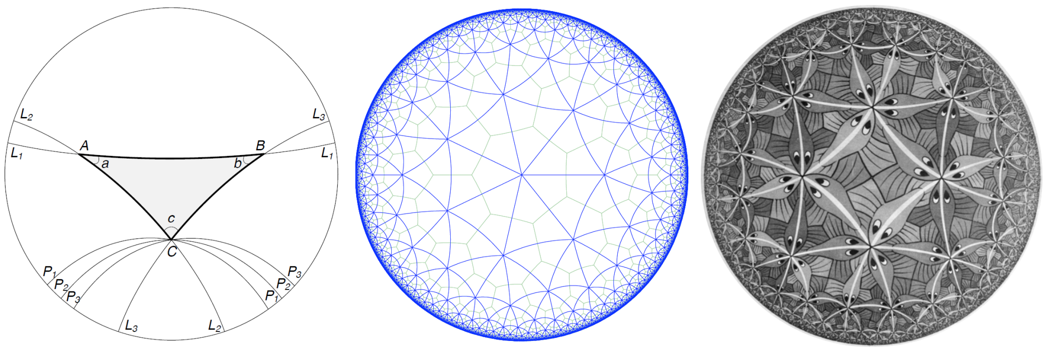
Complex networks, whose hierarchical structure resembles trees, are empirically easier to embed into hyperbolic space than Euclidean space [94], leading to lower distortions.

2.2.1 Models

Several models for hyperbolic geometry were derived by the French mathematician Poincaré, among them the Poincaré disk model, which projects the hyperbolic plane into the unit disk. Angles are preserved, but since an infinite plane is projected into a finite disk, distances and areas are necessarily distorted.

Figure 2.2 shows examples of the Poincaré disk model. Shortest paths between points are circle arcs, intersecting the boundary circle at right angles. Figure 2.2a shows parallels in hyperbolic geometry: The lines P_1, P_2 and P_3 all contain C and are parallel to line L_1, which is impossible within Euclidean geometry – and thus not visible in this Euclidean projection. To distinguish the notation of geometries, we use $\mathrm{dist}_{\mathcal{H}}(.,.)$ for hyperbolic distances and $||\cdot||$ for Euclidean ones.

The true hyperbolic distance between two projected points p_E and q_E in the Poincaré disk



(a) Lines in the hyperbolic plane are projected to circle arcs, hitting the unit circle at a right angle.

(b) Arcs of equal length are drawn smaller with increasing distance to the center

(c) Circle Limit III, M.C. Escher, 1959 [38]. Infinitely many fish in the hyperbolic plane.

Figure 2.2: Visualizations of hyperbolic geometry in the Poincaré disk model. Figures a and b are from [93, Figure 1]. We thank Krioukov et al., who generously gave permission.

is given by the Poincaré metric[4]:

$$\text{dist}_{\mathcal{H}}(p_E, q_E) = \text{acosh}\left(1 + 2\frac{||p_E - q_E||^2}{(1 - ||p_E||^2)(1 - ||q_E||^2)}\right). \tag{2.3}$$

Note that $(1 - ||p_E||^2)$ and $(1 - ||q_E||^2)$ are in the denominator, thus the distance between p and q can be arbitrarily large when they approach the border of the disk. The unit circle itself at radius 1 is not part of the model – points on it would have an infinity hyperbolic distance to all others.

Figure 2.2b shows this effect: The connecting lines all have the same hyperbolic length, but are drawn ever smaller as the distance from the circle center increases.

M. C. Escher used the effect of projecting an infinite plane into a finite area for his *Circle Limit* series: Figure 2.2c shows infinitely many fish, projected into the unit disk.

2.2.2 Coordinate Systems

A point $p = (x, y)$ in the plane can also be defined as its distance to the origin $((0, 0))$ and the angle towards the x-axis. In this form (r_p, θ_p), called *polar coordinates*, the component r_p is the distance to the origin and called the *radial coordinate* of p, while θ_p is called the *angular coordinate.*

A point in polar coordinates can be converted into classical (Cartesian) coordinates with $x := r \cos\theta$ and $y := r \sin\theta$. In the opposite direction, the radial coordinates r is $\sqrt{x^2 + y^2}$

[4]Note that hyperbolic space is a metric space and the properties following from it, for example the triangle inequality, also hold.

and the angular coordinate is found through a case distinction:

$$\theta = \begin{cases} \arctan \frac{y}{x} + \pi & \text{for } x < 0 \\ \arctan \frac{y}{x} & \text{for } x > 0 \\ \frac{\pi}{2} & \text{for } x = 0 \text{ and } y > 0 \\ -\frac{\pi}{2} & \text{for } x = 0 \text{ and } y < 0 \\ \text{undefined} & \text{for } x = y = 0. \end{cases}$$

Native Coordinates

In the Poincaré model, the hyperbolic distance $\mathrm{dist}_{\mathcal{H}}(p_E, (0,0))$ of a point p to the origin can be derived from Equation 2.3:

$$\mathrm{dist}_{\mathcal{H}}(p_E, (0,0)) = \mathrm{acosh}\left(1 + 2\frac{||p_E||^2}{(1 - ||p_E||^2)}\right). \tag{2.4}$$

Krioukov et al. [93] define a coordinate system using polar coordinates, in which the angular coordinate of each point is the same as in the Poincaré disk model, but its radial coordinate is the hyperbolic distance to the origin. They call these *native coordinates.*

Instead of the Poincaré metric, a different method is used to compute distances between points in native coordinates: Two points p and q, along with the origin, define a triangle. In native coordinates, the radial coordinates of p and q are the lengths of two sides of the triangle, while the distance between them is the length of the third side. This distance can then be calculated using the hyperbolic law of cosines:

$$\cosh \mathrm{dist}_{\mathcal{H}}(p,q) = \cosh r_p \cosh r_q - \sinh r_p \sinh r_q \cos \Delta\theta \tag{2.5}$$

$$\mathrm{dist}_{\mathcal{H}}(p,q) = \mathrm{acosh}\left(\cosh r_p \cosh r_q - \sinh r_p \sinh r_q \cos \Delta\theta\right) \tag{2.6}$$

In this, $\Delta\theta = \pi - |\pi - |\theta_p - \theta_q||$ is the angle between p and q at the origin.

This coordinate representation avoids numerical precision issues that may occur due to the small numerical range in the unit disk model. We will mostly use native coordinates in further explorations into hyperbolic space. When the choice of geometry is clearly implied by the context, we omit the subscript $\mathcal{H}$ for brevity.

A mapping $g : \mathcal{H}^2 \to D_1(0)$ from the native representation to the Poincaré disc model needs to preserve the hyperbolic distance to the origin across models. Given the native radial coordinate $r_{\mathcal{H}}$, its corresponding radial coordinate r_e in the Poincaré disc model is then:

$$g(r_{\mathcal{H}}) = \sqrt{\frac{\cosh(r_{\mathcal{H}}) - 1}{\cosh(r_{\mathcal{H}}) + 1}}. \tag{2.7}$$

This mapping gives the correct hyperbolic distance, as can be seen:

$$\begin{aligned}
\mathrm{dist}_{\mathcal{H}}((\phi_{\mathcal{H}}, g(r_{\mathcal{H}})), (0,0)) &= \mathrm{acosh}\left(1 + 2\frac{||(\phi_{\mathcal{H}}, g(r_{\mathcal{H}})) - (0,0)||^2}{(1 - ||(\phi_{\mathcal{H}}, g(r_{\mathcal{H}}))||^2)(1 - ||(0,0)||^2)}\right) \\
&= \mathrm{acosh}\left(1 + 2\frac{||(\phi_{\mathcal{H}}, g(r_{\mathcal{H}}))||^2}{(1 - ||(\phi_{\mathcal{H}}, g(r_{\mathcal{H}}))||^2)(1)}\right) \\
&= \mathrm{acosh}\left(1 + 2\frac{\left|\left|\left(\phi_{\mathcal{H}}, \sqrt{\frac{\cosh(r_{\mathcal{H}})-1}{\cosh(r_{\mathcal{H}})+1}}\right)\right|\right|^2}{\left(1 - \left|\left|\left(\phi_{\mathcal{H}}, \sqrt{\frac{\cosh(r_{\mathcal{H}})-1}{\cosh(r_{\mathcal{H}})+1}}\right)\right|\right|^2\right)}\right) \\
&= \mathrm{acosh}\left(1 + 2\frac{\left(\sqrt{\frac{\cosh(r_{\mathcal{H}})-1}{\cosh(r_{\mathcal{H}})+1}}\right)^2}{\left(1 - \left(\sqrt{\frac{\cosh(r_{\mathcal{H}})-1}{\cosh(r_{\mathcal{H}})+1}}\right)^2\right)}\right) \\
&= \mathrm{acosh}\left(1 + 2\frac{\left(\frac{\cosh(r_{\mathcal{H}})-1}{\cosh(r_{\mathcal{H}})+1}\right)}{1 - \left(\frac{\cosh(r_{\mathcal{H}})-1}{\cosh(r_{\mathcal{H}})+1}\right)}\right) \\
&= \mathrm{acosh}\left(1 + 2\frac{(\cosh(r_{\mathcal{H}}) - 1)}{\cosh(r_{\mathcal{H}}) + 1 - (\cosh(r_{\mathcal{H}}) - 1)}\right) \\
&= \mathrm{acosh}\left(1 + 2\frac{(\cosh(r_{\mathcal{H}}) - 1)}{2}\right) \\
&= \mathrm{acosh}\left((\cosh(r_{\mathcal{H}})\right) = r_{\mathcal{H}}.
\end{aligned}$$

2.2.3 Applications

Krioukov et al. suggest that many network structures can be explained by a hidden geometry, and that complex networks correspond to hyperbolic geometry. As stated above, an intuitive understanding is that the hierarchy common in complex networks corresponds to the tree-like fashion of the hyperbolic expansion of space. Indeed, Kleinberg [90] found that ad-hoc wireless systems and sensor-nets can be projected into hyperbolic space in such a way to enable greedy geographic routing.

Among others, Shavitt and Tankel [149] embed a network of internet routers with measured shortest paths into hyperbolic space, constructing a distance oracle for graph distances. This is possible in hyperbolic space with significantly less distortion than in Euclidean space.

A generative model called *Random Hyperbolic Graphs* (RHG) suggested by Krioukov et al. uses this connection to sample random graphs with some realistic properties. It is considered in detail in Section 3.1.1.

Independent of networks, hyperbolic geometry is connected to Minkowski spacetime, a geometric interpretation of special relativity[160].

Embeddings of symbolic data into the Poincaré ball have been used to learn hierarchical representations [122].

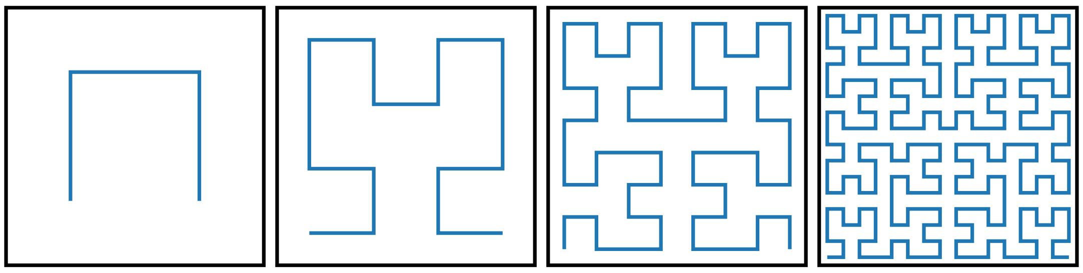


Figure 2.3: The first four iterations of the Hilbert curve in the unit square.

2.3 Space-Filling Curves

A space-filling curve is a function from a one-dimensional interval that surjectively fills a higher-dimensional interval. For 2 dimensions, let h be such a curve. Without loss of generality, it fills the unit square:

$$h : \mathcal{I}[0,1] \to [0,1]^2. \tag{2.8}$$

Hilbert [70] presented a construction in which a curve is refined recursively and reaches the whole unit square in the limit. The first four iterations of the *Hilbert curve* in the unit square are shown in Figure 2.3

While Hilbert was not the first to define a space-filling curve, the Hilbert curves show good *spatial locality*. Two points whose indices on the curve are close, will also be close in the projection: $||h_r(x+\epsilon) - h_r(x)||_2 \leq 2^r\epsilon$, where r is the recursion level of the curve.

Space-filling curves are a way to relate higher-dimensional geometric problems to one-dimensional geometric problems. The popular SLURM workload manager, for example, uses Hilbert curves to linearize the computer topology before assigning jobs, ensuring a higher locality [128].

For an in-depth discussion including the many applications, see the introductory book by Bader [15].

3. Random Hyperbolic Graphs and Probabilistic Queries

This chapter presents several efficient generation algorithms for *Random Hyperbolic Graphs*, a generative graph model proposed by Krioukov et al [93]. After discussing the model itself (Section 3.1.1) and briefly touching on other generation algorithms for complex networks, we present our contributions, starting in Section 3.2. These include a model extension for dynamic graphs (Section 3.2) and four generation algorithms. A polar quadtree on the Poincaré disk (Section 3.3) is the basis of the first subquadratic generation algorithm for threshold random hyperbolic graphs (Section 3.4), a common special case. The algorithm was presented at ISAAC'15 [164]; an extension for general random hyperbolic graphs (Section 3.5) was presented at IWOCA'16 [161] and appeared in the Journal of Experimental Algorithms (JEA), then used with an application to dynamic graph updates [162].

A different data structure based on dividing the Poincaré disk into ring-shaped slabs enables a more efficient algorithm for threshold random hyperbolic graphs, presented in Section 3.6. The algorithm and its implementation were first presented at IEEE HPEC'16 [163] and are reprinted with permission. I'm grateful to Sören Laue, who had the initial algorithmic idea and to Mustafa Özdayi, who did the first implementation in NetworKit, which I later refined. An extension to general random hyperbolic graphs is shown in Section 3.7, it is not published elsewhere yet.

Since the field of generating random hyperbolic graphs has piqued the interest of several research groups, other generation algorithms have been published in the meantime, partly building on our work. We discuss these in more detail in the conclusion, Section 3.9.

3.1 Related Work

3.1.1 Random Hyperbolic Graphs

Krioukov et al. [93] relate complex networks with hierarchical structures to hyperbolic geometry and introduce the family of *random hyperbolic graphs* (RHG). In the RHG model, vertices are generated as points in polar coordinates (ϕ, r) on a disk of radius R (denoted as $\mathbb{D}_R$) in the hyperbolic plane with curvature $-\zeta^2$. The angular coordinate ϕ is drawn from a uniform distribution over $[0, 2\pi]$, while the probability density for the radial coordinate r is given by [93, Eq. (17)] in native coordinates and controlled by a dispersion parameter α:

$$f(r) = \frac{\alpha \sinh(\alpha r)}{\cosh(\alpha R) - 1}, r \in [0, R). \tag{3.1}$$

For $\alpha/\zeta = 1$, this yields a uniform distribution on the hyperbolic plane within $\mathbb{D}_R$. For lower values of α/ζ, vertices are more likely to be in the center, for higher values more likely at the border of $\mathbb{D}_R$.

We denote the hyperbolic distance between two points p_1 and p_2 with $\text{dist}_{\mathcal{H}}(p_1, p_2)$. In the model, any two vertices u and v are connected by an edge with a probability depending on their distance, given by the following equation and parametrized by a temperature T:

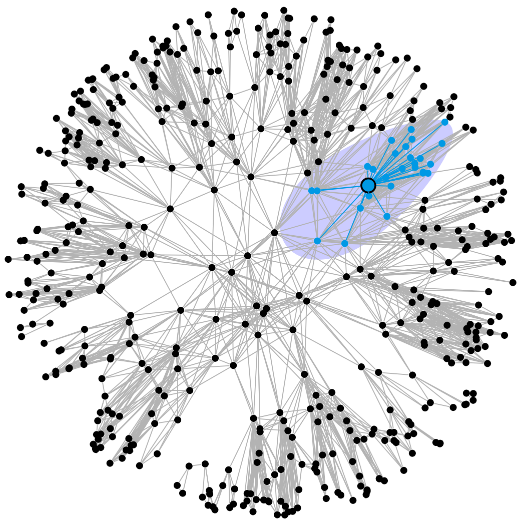
$$p(\{u, v\} \in E) = \left(1 + e^{(1/T)\cdot(\text{dist}_{\mathcal{H}}(u,v)-R)/2}\right)^{-1}. \tag{3.2}$$

For the limiting case of $T = 0$, the neighborhood of a point consists of exactly those points within a hyperbolic circle of radius R, giving rise to the name *threshold random hyperbolic graphs*.[5] In the other extreme of $T = \infty$, the geometry's influence vanishes and the resulting model resembles the Erdős-Rényi-model with a binomial degree distribution.

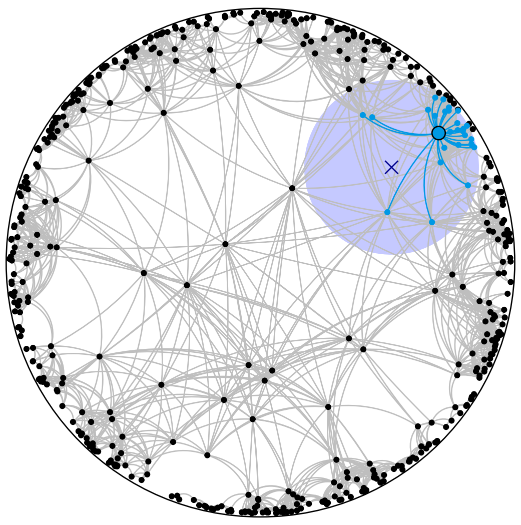
Several works have analyzed the properties of the resulting graphs theoretically. Krioukov et al. [93, Eq. (29)] show that for $\alpha/\zeta \geq \frac{1}{2}$, the degree distribution follows a power law with exponent $\gamma := 2 \cdot \alpha/\zeta + 1$. Gugelmann et al. [63] prove non-vanishing clustering and a low variation of the clustering coefficient. Bode et al. [24] show a phase transition at $\zeta/\alpha = 1$: For $\zeta/\alpha < 1$, the size of the largest component is almost surely sublinear, for $\zeta/\alpha > 1$, it is linear [23]. They also show that the curvature parameter ζ can be fixed while retaining all degrees of freedom, leading us to assume $\zeta = 1$ from now on without loss of generality. For $2 < \gamma < 3$, Kiwi and Mitsche [89] bound the diameter asymptotically almost surely to $\mathcal{O}((\log n)^{32/((5-\gamma)(3-\gamma))})$. Friedrich and Krohmer [56] present a tighter bound of $\mathcal{O}((\log n)^{2/(3-\gamma)})$. The average degree $\overline{k}$ of a random hyperbolic graph is controlled with the radius R, using an approximation given by [93, Eq. (22)]. This radius is commonly set to $R = 2 \log n + C$ with a user-defined constant C, leading to a stable average degree with changing graph size. An example of a threshold random hyperbolic graph with 500 vertices, $R \approx 5.08$, $T = 0$ and $\alpha = 0.8$ in this representation is shown in Figure 3.1a. For the purpose of illustration in the figure, we choose a vertex u (the bold blue vertex) and an artificially small[6] example neighborhood, adding edges (u, v) for all vertices v where $\text{dist}_{\mathcal{H}}(u, v) \leq 0.2 \cdot R$. The neighborhood of u then consists of vertices within a hyperbolic circle (marked in blue). Figure 3.1b shows the same graph with coordinates in the Poincaré disk model. Note that geodesics in this model are circle arcs, meeting the boundary circle

[5] Also called *hyperbolic unit-disk graphs*, *step model of random hyperbolic graphs* or (slightly confusingly) just random hyperbolic graphs. While we consider *hyperbolic unit-disk graphs* to be more intuitive, we stick with *threshold random hyperbolic graphs* to avoid name proliferation.

[6] Depicting neighborhoods as in the actual model of RHGs would result in a graph too dense to be useful in a visualization.



(a) Native representation as specified in [93].



(b) Poincaré disk model.

Figure 3.1: Threshold random hyperbolic graph in native coordinates and the Poincaré disk model. Neighbors of the bold blue vertex are in the hyperbolic respective Euclidean circle. In this visualization, an edge (u, v) is only added if $\mathrm{dist}_{\mathcal{H}}(u, v) \leq 0.2R$. The center of the Euclidean circle is marked with $\times$.

at a right angle.

Papadopoulos et al. [126] consider random hyperbolic graphs as a preferential attachment model. They argue that, when given in polar coordinates, the angular and radial coordinates model popularity and similarity, respectively, of vertices in a network. An edge is more likely if one of the vertices is close to the center of the disk (thus having higher popularity) or when vertices have a small angular distance, modeling their similarity.

3.1.2 Other Graph Generators

Numerous other generative graph models exist, including many designed for complex networks and not based on geometry. For a short overview, see [153]. All these models cover different aspects of network formation and the graphs generated by them have systematically different properties. No model is widely accepted as covering the majority of use cases.

Most ancient of all, the Erdős-Rényi (ER) model [52] draws a random graph uniformly among those with a specified number of vertices and edges. This model is easy to analyze and implement, but lacking in structure and not a suitable fit for modeling complex networks.

The Barabasi-Albert model [4] is a preferential attachment model, designed to replicate the growth of real complex networks. The probability that a new vertex attaches to an existing vertex v is proportional to v's degree, which results in a power-law degree distribution. While the distribution's exponent is constant for the basic model, generalizations for arbitrary exponents exist (see e. g. [120, Chap. 14]). Preferential attachment processes can be implemented with a running time in $\mathcal{O}(n + m)$ [18].

The Dorogovtsev-Mendes model [47] is designed to model network growth by attaching new nodes to the end points of a randomly chosen edge. It is very fast in theory ($\Theta(n)$) and practice and yields a power-law degree distribution. However, it accepts only the vertex count as parameter and has thus a fixed average degree.

The Recursive Matrix (R-MAT) model [36] was proposed to recreate properties of complex networks including a power-law degree distribution, the small-world property and self-similarity. Design goals also include few parameters and high generation speed. The R-MAT generator recursively subdivides the initially empty adjacency matrix into quadrants and drops edges into it according to given probabilities. It has $\Theta(m \log n)$ asymptotic complexity and is fast in practice. However, at least the R-MAT parameters used by the Graph500 system benchmark [118] lead to an insignificant community structure compared to real-world graphs and also small clustering coefficients.

Given a degree sequence *seq*, the Chung-Lu (CL) model [3] adds each edge (u, v) with a probability of $p(u, v) = \frac{seq(u)seq(v)}{\sum_k seq(k)}$. The expected degree of each vertex is then as specified in *seq*. The model can be conceived as a weighted version of the well-known Erdős-Rényi model and has similar capabilities as the R-MAT model [148]. Implementations exist with $\Theta(n + m)$ time complexity [114]. The LFR benchmark generator [97] was designed by Lancichinetti, Fortunato and Radicchi to evaluate community detection algorithms. Typically, the user specifies vertex degrees and community sizes. Each vertex is then assigned to a community and connects a $(1 - \mu)$-fraction of its edges to other vertices in the same community, the remaining edges lead to vertices in other communities. The mixing parameter μ controls the hardness of the benchmark instance.

The *block two-level ER model* [91] (BTER) uses the standard ER model to form relatively dense subgraphs and thus distinct communities. Afterwards, the Chung-Lu model is used to add edges, matching the desired degree distribution in expectation [147]. This is done in $\Theta(n + m \log d_{\max})$, where $d_{\max}$ is the maximum vertex degree. The BTER model has been extended by Bridges et al. [30] and El-daghar et al. [50] to more faithfully model community structures.

Mahadevan et al. [104] define and discuss dK-distributions, the vertex degree correlation among subgraphs of size d. When used to replicate input graphs, an increasing value of d causes the replica to be increasingly faithful to the original.

Staudt et al. [153] use several graph generators to create scaled replicas of real-world graphs and compare their similarity to the originals.

3.1.3 Related Range Queries

The probabilistic neighborhood queries used in the generation of general hyperbolic random graphs can also be used independently. We thus discuss related range queries on geometric datasets.

Fast Deterministic Range Queries

A good overview of spatial queries on geometric datasets is given in Samet's book [141], mostly concerning trees and their variations. See the survey of Arge and Larsen [7] for

I/O efficient worst case analysis in an external memory model. A focus on average-case performance led to the rise of R-trees e. g. [80] in applied settings.

Deterministic queries in balanced quadtrees and *kd*-trees of dimension d require $\mathcal{O}(d \cdot n^{1-1/d})$ time [141, Ch. 1.4], thus $\mathcal{O}(\sqrt{n})$ in the planar case. Our algorithm for PNQs matches this query complexity up to a logarithmic factor. Yet note that, since for general probability functions f and distance metrics in our scenario all points in the set P could be neighbors, data structures for deterministic queries cannot solve a PNQ efficiently without adaptations.

Hu et al. [74] give a query sampling algorithm for one-dimensional data that, given a set P of n points in $\mathbb{R}$, an interval $q = [x, y]$ and an integer $t \geq 1$, returns t elements uniformly sampled from $P \cap q$. They describe a structure of $\mathcal{O}(n)$ space that answers a query in $\mathcal{O}(\log n + t)$ time and supports updates in $\mathcal{O}(\log n)$ time. While also offering query sampling, PNQs differ from the problem considered by Hu et al. in two aspects: We consider not only the 1-dimensional case, and our sampling probabilities (user-defined with a distance-dependent function) are not necessarily uniform.

Range Queries on Uncertain Data

During the previous decade probabilistic queries *different* from PNQs have become popular. The main scenarios can be put into two categories [130]: (i) Probabilistic databases contain entries that come with a specified confidence (e. g. sensor data whose accuracy is uncertain) and (ii) objects with an uncertain location, i. e. the location is specified by a probability distribution. Both scenarios differ under typical and reasonable assumptions from ours: Queries for uncertain data are usually formulated to return *all* points in the neighborhood whose confidence/probability exceeds a certain threshold [92], or computing points that are possibly nearest neighbors [2].

In our model, the choice of inclusion of a point p is a random choice for every different p. In particular, depending on the probability distribution, *all* vertices in the plane can have positive probability to be part of some other's neighborhood. In the related scenarios this would only be true with extremely small confidence values or extremely large query circles.

3.2 Dynamic Model

Many phenomena modeled by graphs change over time.[7] *Dynamic graphs* model both a structure and its changes. For the case of random hyperbolic graphs, Papadopoulos et al. [127] examine greedy routing in changing networks and for this purpose propose a dynamic model. Network growth is modeled as newly arriving vertices having higher radial coordinates than existing ones, thus join the network at the border of a growing disk. Departure of vertices happens uniformly at random. While this is a suitable dynamic behavior for modeling internet infrastructure with sudden site failures or additions, change

[7]Even the famous solution to the problem of crossing all bridges of Königsberg exactly once – it no longer applies. Some of the bridges in now-Kaliningrad have been torn down, others constructed.

in e. g., social networks happens more gradually; people rarely leave society completely and rejoin it at a random position.

To model such a gradual change in networks, we design and implement a dynamic version with vertex movement. Such a model should fulfill several objectives: First, it should be *consistent*: After moving a vertex, the network may change, but properties should stay the same *in expectation.* Since the properties emerge from the vertex positions, the probability distribution of vertex positions needs to be preserved. Second, the movement should be *directed*: If the movement direction of a vertex at time t is completely independent from the direction at $t+1$, the same links would vanish and reappear repeatedly.

We attain the first objective by scaling the movements along the radial axis, as given by Theorem 1. The second objective is fulfilled by initially setting step values τ_ϕ and τ_r for each vertex and using them in each movement step. As a result, if a vertex i moves in a certain direction at time t, it will move in the same direction at $t+1$, except if the new position would be outside the hyperbolic disk $\mathbb{D}_R$. In this case, the movement is inverted and the vertex “bounces” off the boundary. The different probability densities in the center of the disk and the outer regions are translated into movement speed: A vertex is less likely to be in the center; thus it needs to spend less time there while traversing it, resulting in a higher speed. In the interpretation of popularity and similarity, this could be compared to participants having their “15 minutes of fame”.

We implement this movement in two phases and with discrete time steps: In the initialization, step values τ_ϕ and τ_r are assigned to each vertex according to the desired movement. Each movement step of a vertex then consists of a rotation and a radial movement. This step is described in Algorithm 1; a visualization of the radial movement is shown in Figure 3.2.

Algorithm 1: move(ϕ, r) – Movement step in dynamic model

```
Input: $\phi, r, \tau_\phi, \tau_r, R, \alpha$.
Output: $\phi_{\text{new}}$, $r_{\text{new}}$
    1. x = $\sinh(r \cdot \alpha)$;
    2. y = x+$\tau_r$;
    3. z = asinh(y)/$\alpha$;
    4. $\phi_{\text{new}} = (\phi + \tau_\phi) \bmod 2\pi$
    5. Return ($\phi_{\text{new}}$, z)
```

If the new vertex position would be outside the boundary ($r > R$) or below the origin ($r < 0$), the movement is reflected and τ_r set to $-\tau_r$.

Theorem 1. *Let $f_{r,\phi}((p_r, p_\phi))$ be the probability density of point positions, given in polar coordinates. Let* move$((p_r, p_\phi))$ *(Algorithm 1) be a movement step. Then, the vertex movement preserves the distribution of angular and radial distributions: $f_{r,\phi}(\text{move}((p_r, p_\phi))) = f_{r,\phi}((p_r, p_\phi))$.*

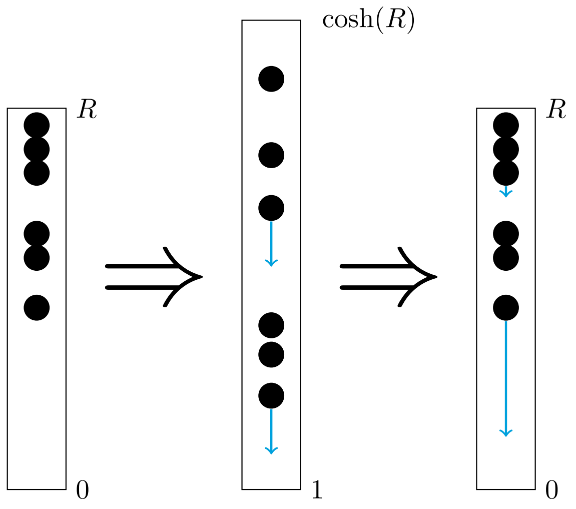


Figure 3.2: For each movement step, radial coordinates are mapped into the interval $[1, \cosh(\alpha R))$, where the coordinate distribution is uniform. Adding τ_r and transforming the coordinates back results in correctly scaled movements.

Proof. Since the distributions of angular and radial coordinates are independent, we consider them separately: $f_{r,\phi}(p_r, p_\phi) = f_r(p_r) \cdot f_\phi(p_\phi)$.

As introduced in Eq. (3.1), the radial coordinate r is sampled from a distribution with density $\alpha \sinh(\alpha r)/(\cosh(\alpha R) - 1)$. We introduce random variables X, Y, Z for each step concerning the radial coordinates in Algorithm 1, each is denoted with the upper case letter of its equivalent. An additional random variable Q denotes the pre-movement radial coordinate. The other variables are defined as $X = \sinh(Q \cdot \alpha)$, $Y = X + \tau_r$ and $Z = \operatorname{asinh}(Y)/\alpha$.

Let f_Q, f_X, f_Y and f_Z denote the density functions of these variables:

$$\begin{aligned}
f_Q(r) &= \frac{\alpha \sinh(\alpha r)}{\cosh(\alpha R) - 1}, \\
f_X(r) &= f_Q\left(\frac{\operatorname{asinh}(r)}{\alpha}\right) = \frac{\alpha r}{\cosh(\alpha R) - 1}, \\
f_Y(r) &= f_X(r - \tau_r) = \frac{\alpha r - \tau_r}{\cosh(\alpha R) - 1}, \\
f_Z(r) &= f_Y(\sinh(r \cdot \alpha)) = \frac{\alpha \sinh(\alpha r) - \tau_r}{\cosh(\alpha R) - 1} = f_Q(r) - \frac{\tau_r}{\cosh(\alpha R) - 1}.
\end{aligned}$$

The distributions of Q and Z only differ in the constant addition of $\tau_r/(\cosh(\alpha R) - 1)$. Every $(\cosh(\alpha R) - 1)/\tau_r$ steps, the radial movement reaches a limit (0 or R) and is reflected, causing τ_r to be multiplied with -1. On average, τ_r is thus zero and $F_Q(r) = F_Z(r)$.

A similar argument works for the rotational step: While the rotational direction is unchanged, the change in coordinates is balanced by the addition or subtraction of 2π whenever the interval $[0, 2\pi)$ is left, leading to an average of zero in terms of change. □

3.3 Polar Quadtrees on the Poincaré disk

Efficient geometric algorithms need suitable geometric data structures. Our first algorithm for sublinear neighborhood queries (and thus subquadratic generation) of random hyperbolic graphs uses a *polar quadtree* on the Poincaré disk representation of the hyperbolic plane, introduced in Section 2.2.1. Since in the Poincaré disk model, the unit disk represents the entire infinite hyperbolic plane, we construct quadtrees only on a *subset* of the unit disk.

Quadtrees are geometric data structures for the plane. Each node in the tree covers a (usually) rectangular region and either has four children or is a leaf node. Leaf nodes directly contain points or other objects, for each inner node C, the regions of its four children are a disjoint partition of the region of C.

While quadtrees are less suited to higher dimensions as for example k-d-trees, the complexity is comparable in the plane. For the (circular) range queries we discuss, quadtrees have the significant advantage of a bounded aspect ratio: A cell in a k-d-tree might extend arbitrarily far in one direction, rendering theoretical guarantees about the area affected by the query circle difficult. In contrast, the region covered by a quadtree cell is determined by its position and level. Euclidean quadtrees are common [141], but we are not aware of previous adaptations to hyperbolic space.

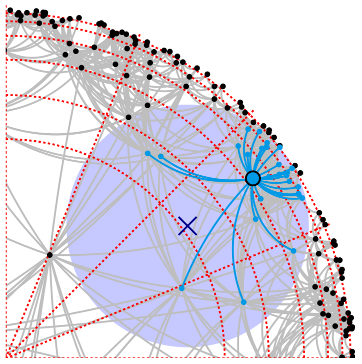


Figure 3.3: Polar quadtree

A node in the quadtree T is defined as a tuple $(\min_\phi, \max_\phi, \min_r, \max_r)$ with $\min_\phi \leq \max_\phi$ and $\min_r \leq \max_r$. It is responsible for a point $p = (\phi_p, r_p) \in D_1(0)$ iff $(\min_\phi \leq \phi_p < \max_\phi)$ and $(\min_r \leq r_p < \max_r)$. Figure 3.3 shows a section of a polar quadtree, where quadtree nodes are marked by dotted red lines. We call the geometric region corresponding to a quadtree node its *quadtree cell.* The root cell of Q has the boundaries $\min_\phi := 0, \max_\phi := 2\pi, \min_r := 0$ and $\max_r := R_T$, the radius of the disk that should be covered by T. When a point is to be inserted into an already full leaf node, the node is split into four children, once in the angular and once in the radial direction. Splitting in the angular direction is straightforward as the angle range is halved: $\text{mid}_\phi := \frac{\max_\phi + \min_\phi}{2}$. For the radial direction, we choose the splitting radius to result in an equal division of probability mass. For this, the radial probability distribution must be integrable and have a support of exactly $[0, R_T)$. This is the case for random hyperbolic graphs (see Eq. 3.1), the quadtree radius R_T is then the Poincaré projection of the native disk radius R.

Let $j(r)$ denote the radial probability distribution and $J(r) := \int_0^{R_T} j(s)ds$ the indefinite integral of $j(r)$. We normalize $J(r)$ so that $J(R_T) = 1$. The value $J(r)$ then gives the fraction of probability mass inside radius r.

The total probability mass in a ring delimited by $\min_r$ and $\max_r$ is then $J(\max_r) - J(\min_r)$. Since $j(r)$ is positive for r between 0 and R_T, the restricted function $J|_{[0,R_T]}$ is a bijection. The inverse $(J|_{[0,R_T]})^{-1}$ thus exists and we set the splitting radius mid_r to $(J|_{[0,R_T]})^{-1}\left(\frac{J(\max_r)+J(\min_r)}{2}\right)$. In the context of random hyperbolic graphs, this leads to:

$$\mathrm{mid}_{r\mathcal{H}} := \operatorname{acosh}\left(\frac{\cosh(\alpha \max_{r\mathcal{H}}) + \cosh(\alpha \min_{r\mathcal{H}})}{2}\right)/\alpha. \tag{3.3}$$

(Note that Eq. (3.3) uses radial coordinates in the native representation, which are converted back to coordinates in the Poincaré disk.) This leads to three lemmas useful for establishing the time complexity of the main quadtree operations:

Lemma 1. *Let $\mathbb{D}_{R_T}$ be a disk of radius R_T, $j(r)$ be a probability distribution with the support $[0, R_T)$, T be a polar quadtree on $\mathbb{D}_{R_Q}$ constructed according to j, and let p be a point in $\mathbb{D}_{R_T}$, drawn with a uniformly distributed angular coordinate and a radial coordinate distributed as j. Let C be a quadtree cell at depth i. Then, the probability that p is in C is 4^{-i}.*

Proof. Let C be a quadtree cell at level k, delimited by $\min_r$, $\max_r$, $\min_\phi$ and $\max_\phi$. The probability that a point p is in C is then given by

$$\Pr(p \in C) = \frac{\max_\phi - \min_\phi}{2\pi} \cdot (J(\max_r) - J(\min_r)). \tag{3.4}$$

The boundaries of the children of C are given by the splitting rules of Equation 3.3.

$$\mathrm{mid}_\phi := \frac{\max_\phi + \min_\phi}{2} \tag{3.5}$$

$$\mathrm{mid}_r := (J|_{[0,R]})^{-1}\left(\frac{J(\max_r) + J(\min_r)}{2}\right) \tag{3.6}$$

We proceed with induction over the depth i of C. Start of induction ($i = 0$): At depth 0, only the root cell exists and covers the whole disk. Since $C = \mathbb{D}_R$, $\Pr(p \in C) = 1 = 4^{-0}$.

Inductive step ($i \to i + 1$): Let C_i be a node at depth i. C_i is delimited by the radial boundaries $\min_r$ and $\max_r$, as well as the angular boundaries $\min_\phi$ and $\max_\phi$. It has four children at depth $i + 1$, separated by mid_r and mid_ϕ. Let SW be the south west child of C_i. With Eq. (3.4), the probability of $p \in SW$ is:

$$\Pr(p \in SW) = \frac{\mathrm{mid}_\phi - \min_\phi}{2\pi} \cdot (J(\mathrm{mid}_r) - J(\min_r))$$

.

Using Equations (3.5) and (3.6), this results in a probability of

$$\begin{aligned}
\Pr(p \in SW) &= \frac{\frac{\max_\phi + \min_\phi}{2} - \min_\phi}{2\pi} \cdot \left(J\left((J|_{[0,R_T]})^{-1} \left(\frac{J(\max_r) + J(\min_r)}{2} \right) \right) - J(\min_r) \right) \\
&= \frac{\frac{\max_\phi + \min_\phi}{2} - \min_\phi}{2\pi} \cdot \left(\frac{J(\max_r) + J(\min_r)}{2} - J(\min_r) \right) \\
&= \frac{\frac{\max_\phi - \min_\phi}{2}}{2\pi} \cdot \left(\frac{J(\max_r) - J(\min_r)}{2} \right) \\
&= \frac{1}{4} \frac{\max_\phi - \min_\phi}{2\pi} \cdot (J(\max_r) - J(\min_r))
\end{aligned}$$

As per the induction hypothesis, $\Pr(p \in C_i)$ is 4^{-i} and $\Pr(p \in SW)$ is thus $\frac{1}{4} \cdot 4^{-i} = 4^{-(i+1)}$. Due to symmetry when selecting mid_ϕ, the same holds for the south east child of C_i. Together, they contain half of the probability mass of C_i. Again due to symmetry, the same proof then holds for the northern children as well. □

Lemma 2. *When n balls are thrown independently and uniformly at random into n^3 bins, the probability that any bin receives more than one ball is less than $1/2n$.*

Proof. Each ball has a probability of $1/\left(n^3\right)$ to land in a given bin. Since balls are thrown independently, each pair of balls has a probability of $1/\left(n^3\right)^2$ to land in a given bin. Among n balls, $\binom{n}{2}$ pairs exist. With Boole's inequality, it follows that the probability of a given bin receiving at least two balls is:

$$\binom{n}{2}\left(\frac{1}{n^3}\right)^2.$$

With n^3 bins, it follows again from Boole's inequality that the probability that any of the bins receives at least two balls is at most

$$n^3\binom{n}{2}\left(\frac{1}{n^3}\right)^2 = \binom{n}{2}\left(\frac{1}{n^3}\right).$$

This is smaller than $1/n$:

$$\binom{n}{2}\left(\frac{1}{n^3}\right) = \frac{n(n-1)}{2} \cdot \frac{1}{n^3} < \frac{n^2}{2} \cdot \frac{1}{n^3} = \frac{1}{2n}.$$

□

Lemma 3. *Let R_T and $\mathbb{D}_{R_T}$ be as in Lemma 1. Let T be a polar quadtree on $\mathbb{D}_{R_T}$ containing n points whose positions follow the probability distributions for which the quadtree was constructed. Then, for n sufficiently large,* $\text{height}(T) \in \mathcal{O}(\log n)$ *whp.*

Proof. In a complete quadtree, 4^i cells exist at depth i. For analysis purposes only, we construct such a complete but initially empty quadtree of height $k = 3 \cdot \lceil \log_4(n) \rceil$, which has at least n^3 leaf cells. As seen in Lemma 1, a given point has an equal chance to land in each leaf cell. Hence, we can apply Lemma 2 with each leaf cell being a bin and a point being a ball. (The fact that we can have more than n^3 leaf cells only helps in reducing the average load.) From this we can conclude that, for n sufficiently large, no leaf cell of the current tree contains more than 1 point with high probability (whp). Consequently, the total quadtree height does not exceed $k = 3 \cdot \lceil \log_4(n) \rceil \in \mathcal{O}(\log n)$ whp.

Let T' be the quadtree as constructed in the previous paragraph, starting with a complete quadtree of height k and splitting leaves when their capacity is exceeded. Let T be the quadtree created in our algorithm, starting with a root node, inserting points and also splitting leaves when necessary, growing the tree downward.

Since both trees grow downward as necessary to accommodate all points, but T does not start with a complete quadtree of height k, the set of quadtree nodes in T is a subset of the quadtree nodes in T'. Consequently, the height of T is bounded by $\mathcal{O}(\log n)$ whp as well. □

Lemma 4. *Inserting n points into an initially empty quadtree T has a time complexity of $\mathcal{O}(n \log n)$ whp.*

Proof. Let $h(T)$ be the final height of T. Since the height never shrinks when inserting a element, it is also the maximum height during construction. For each element insertion, at most $h(T)$ nodes are visited to find the correct leaf for insertion. When a node is split, all affected elements are moved down one level. Thus, each element is moved at most $h(T)$ times during the insertion of other elements and the necessary node splits. The time complexity to construct a quadtree T with n vertices is thus $\mathcal{O}(n \cdot 2 \cdot h(T)) = \mathcal{O}(n \cdot h(T))$, which is $\mathcal{O}(n \log n)$ whp due to Lemma 3. □

3.3.1 Distance between Quadtree Cell and Point

For range queries, both deterministic and probabilistic, we need a lower bound for the distance between the query point q and any point in a given quadtree cell. Since the quadtree cells are polar, the distance calculations might be unfamiliar and we show and prove them explicitly. For hyperbolic geometry, we use native coordinates, the resulting distance calculations are shown in Algorithm 2 and proven in Lemma 5. To enable a more general use also in Euclidean geometry (we've heard it is slightly more popular for some reason), we show the distance calculations for Euclidean geometry in Algorithm 3 and prove them in Lemma 6.

Lemma 5. *Let C be a quadtree cell and q a point in the hyperbolic plane, both given in native coordinates. The first value returned by Algorithm 2 is the distance of C to q.*

Proof. When q is in C, the distance is trivially zero. Otherwise, the distance between q and C can be reduced to the distance between q and the boundary of C, ∂C:

$$\mathrm{dist}_{\mathcal{H}}(C, q) = \mathrm{dist}_{\mathcal{H}}(\partial C, q) = \inf_{p \in \partial C} \mathrm{dist}_{\mathcal{H}}(p, q) \tag{3.7}$$

Algorithm 2: Infimum and supremum of distance in a hyperbolic polar quadtree

```
Input: quadtree cell $C = (\min_r, \max_r, \min_\phi, \max_\phi)$, query point $q = (\phi_q, r_q)$
Output: infimum and supremum of hyperbolic distances $q$ to points in $C$
/* start with corners of cell as possible extrema                    */
cornerSet = $\{(\min_\phi, \min_r), (\min_\phi, \max_r), (\max_\phi, \min_r), (\max_\phi, \max_r)\}$;
a = $\cosh(r_q)$;
b = $\sinh r_q \cdot \cos(\phi_q - \min_\phi)$;
/* Left/Right boundaries                                              */
leftExtremum = $\frac{1}{2} \ln \left(\frac{a+b}{a-b}\right)$;
if $min_r <$ leftExtremum $< max_r$ then
    add $(\min_\phi, \text{leftExtremum})$ to cornerSet;
b = $\sinh r_q \cdot \cos(\phi_q - \max_\phi)$;
rightExtremum = $\frac{1}{2} \ln \left(\frac{a+b}{a-b}\right)$;
if $min_r <$ rightExtremum $< max_r$ then
    add $(\max_\phi, \text{rightExtremum})$ to cornerSet;
/* Top/bottom boundaries                                              */
if $min_\phi < \phi_q < max_\phi$ then
    add $(\phi_q, \min_r)$ and $(\phi_q, \max_r)$ to cornerSet;
$\phi_{\text{mirrored}} = \phi_q + \pi \mod 2\pi$;
if $min_\phi < \phi_{mirrored} < max_\phi$ then
    add $(\phi_{\text{mirrored}}, \min_r)$ and $(\phi_{\text{mirrored}}, \max_r)$ to cornerSet;
/* If point is in cell, distance is zero:                             */
if $min_\phi \leq \phi_q < max_\phi$ and $min_r \leq r_q < max_r$ then
    infimum = 0;
else
    infimum = $\min_{e \in \text{cornerSet}} \text{dist}_{\mathcal{H}}(q, e)$;
supremum = $\max_{e \in \text{cornerSet}} \text{dist}_{\mathcal{H}}(q, e)$;
return infimum, supremum;
```

Algorithm 3: Infimum and supremum of distance in a Euclidean polar quadtree

```
Input: quadtree cell $C = (\min_r, \max_r, \min_\phi, \max_\phi)$, query point $q = (\phi_q, r_q)$
Output: infimum and supremum of Euclidean distances $q$ to interior of $C$
/* start with corners of cell as possible extrema                              */
1  cornerSet = $\{(\min_\phi, \min_r), (\min_\phi, \max_r), (\max_\phi, \min_r), (\max_\phi, \max_r)\}$;
   /* Left/Right boundaries                                                    */
2  leftExtremum= $r_q \cdot \cos(\min_\phi - \phi_q)$;
3  if $min_r <$ leftExtremum $< max_r$ then
4      add $(\min_\phi, \text{leftExtremum})$ to cornerSet;
5  rightExtremum= $r_q \cdot \cos(\max_\phi - \phi_q)$;
6  if $min_r <$ rightExtremum $< max_r$ then
7      add $(\max_\phi, \text{rightExtremum})$ to cornerSet;
   /* Top/bottom boundaries                                                    */
8  if $min_\phi < \phi_q < max_\phi$ then
9      add $(\phi_q, \min_r)$ and $(\phi_q, \max_r)$ to cornerSet;
10 $\phi_{\text{mirrored}} = \phi_q + \pi \mod 2\pi$;
11 if $min_\phi < \phi_{mirrored} < max_\phi$ then
12     add $(\phi_{\text{mirrored}}, \min_r)$ and $(\phi_{\text{mirrored}}, \max_r)$ to cornerSet;
   /* If point is in cell, distance is zero:                                   */
13 if $min_\phi \leq \phi_q < max_\phi$ AND $min_r \leq r_q < max_r$ then
14     infimum = 0;
15 else
16     infimum = $\min_{e \in \text{cornerSet}} \text{dist}_\mathcal{H}(q, e)$;
17 supremum = $\max_{e \in \text{cornerSet}} \text{dist}_\mathcal{H}(q, e)$;
18 return infimum, supremum;
```

Since the boundary is closed and bounded, this infimum is actually a minimum:

$$\mathrm{dist}_{\mathcal{H}}(C,q) = \inf_{p\in\partial C} \mathrm{dist}_{\mathcal{H}}(p,q) = \min_{p\in\partial C} \mathrm{dist}_{\mathcal{H}}(p,q) \tag{3.8}$$

The boundary of a quadtree cell consists of four closed curves:

- left: $\{(\min_\phi, r) | \min_r \leq r \leq \max_r\}$
- right: $\{(\max_\phi, r) | \min_r \leq r \leq \max_r\}$
- lower: $\{(\phi, \min_r) | \min_\phi \leq \phi \leq \max_\phi\}$
- upper: $\{(\phi, \max_r) | \min_\phi \leq \phi \leq \max_\phi\}$

We write the distance to the whole boundary as a minimum over the distances to its parts:

$$\mathrm{dist}_{\mathcal{H}}(\partial C,q) = \min_{A\in\{\text{left, right, lower, upper}\}} \mathrm{dist}_{\mathcal{H}}(A,q) \tag{3.9}$$

Angular Boundaries

All points on an angular boundary curve A have the same angular coordinate ϕ_A. The distance of a point q in the hyperbolic plane and a point $q' \in A$ thus only depends on the radial coordinate of q'. Let $d_{A,q}(r) = \mathrm{acosh}(\cosh(r)\cosh(r_q) - \sinh(r)\sinh(r_q)\cos(\phi_q - \phi_A))$ for a fixed point q. The distance $\mathrm{dist}_{\mathcal{H}}(A,q)$ can then be reduced to:

$$\mathrm{dist}_{\mathcal{H}}(A,q) = \min_{\min_r \leq r \leq \max_r} d_{A,q}(r).$$

The minimum of $d_{A,q}$ on A is the minimum of $d_{A,q}(\min_r)$, $d_{A,q}(\max_r)$ and the value at possible extrema. To find the extrema, we define a function $g(r) = \cosh(r)\cosh(r_q) - \sinh(r)\sinh(r_q)\cos(\phi_q - \phi_A)$. Since acosh is strictly monotone, $g(r)$ has the same extrema as $d_{A,q}(r)$.

The factors $\cosh(r_q)$ and $\sinh(r_q)\cos(\phi_q - \phi_A)$ do not depend on r, to increase readability we substitute them with the constants a and b:

$$\begin{aligned}
a &= \cosh(r_q)\\
b &= \sinh(r_q)\cos(\phi_q - \phi_A)\\
d_{A,q}(r) &= \mathrm{acosh}(\cosh(r)\cdot a - \sinh(r)\cdot b)\\
g(r) &= \cosh(r)\cdot a - \sinh(r)\cdot b
\end{aligned}$$

The derivative of g is thus:

$$g'(r) = \sinh(r)\cdot a - \cosh(r)\cdot b = \frac{e^r - e^{-r}}{2}\cdot a - \frac{e^r + e^{-r}}{2}\cdot b$$

With some transformations, we get the roots of $g'(r)$:

Case $a = b$**:**

$$
\begin{aligned}
g'(r) = 0 &\iff \frac{e^r - e^{-r}}{2} \cdot a = \frac{e^r + e^{-r}}{2} \cdot a \iff \\
e^r - e^{-r} = e^r + e^{-r} &\iff -e^{-r} = e^{-r} \iff e^{-r} = 0
\end{aligned}
$$

For $a = b$, $d_{A,q}$ has no extrema in $\mathbb{R}$.

Case $a \neq b$**:**

$$
\begin{aligned}
g'(r) = 0 &\iff \frac{e^r - e^{-r}}{2} \cdot a = \frac{e^r + e^{-r}}{2} \cdot b &&\iff \\
ae^r - ae^{-r} = be^r + be^{-r} &\iff (a-b)e^r - (a+b)e^{-r} = 0 &&\iff \\
(a-b)e^r = (a+b)e^{-r} &\iff e^r = \frac{a+b}{a-b}e^{-r} &&\iff \\
e^{2r} = \frac{a+b}{a-b} &\iff 2r = \ln\left(\frac{a+b}{a-b}\right) &&\iff \\
r &= \frac{1}{2}\ln\left(\frac{a+b}{a-b}\right)
\end{aligned}
$$

For $a \neq b$, $d_{A,q}$ has a single extremum at $\frac{1}{2}\ln\left(\frac{a+b}{a-b}\right)$. This extremum is calculated for both angular boundaries in Lines 4 and 8 of Algorithm 2.

If $d(r)$ has an extremum x in A, the minimum of $d_{A,q}(r)$ on A is $\min\{d_{A,q}(\min_r), d_{A,q}(\max_r), d_{A,q}(x)\}$, otherwise it is $\min\{d_{A,q}(\min_r), d_{A,q}(\max_r)\}$.

Radial Boundaries

A similar approach works for the radial boundary curves. Let B be a radial boundary curve at radius r_B and angular bounds $\min_\phi$ and $\max_\phi$. Let $d_{B,q}(\phi)$ be the distance to q restricted to radius r_B.

$$
\begin{aligned}
d_{B,q} &: [0, 2\pi] \to \mathbb{R} \\
d_{B,q}(\phi) &= \operatorname{acosh}(\cosh(r_B)\cosh(r_q) - \sinh(r_B)\sinh(r_q)\cos(\phi_q - \phi))
\end{aligned}
$$

Similarly to the angular boundaries, we define some constants and a function $g(\phi)$ with the same extrema as $d_{B,q}$:

$$
\begin{aligned}
a &= \cosh(r_B)\cosh(r_q) \\
b &= \sinh(r_B)\sinh(r_q) \\
g(\phi) &= a - b\cos(\phi_q - \phi)
\end{aligned}
$$

Case: $b = 0$:

$$b = \sinh(r_B)\sinh(r_q) = 0 \iff g(\phi) = a$$

Since g is constant, no extrema exist.

Case: $b \neq 0$:

We obtain the extrema with some transformations:

$$\begin{aligned} g'(\phi) &= -b\sin(\phi_q - \phi) \\ g'(\phi) &= 0 \iff \\ \sin(\phi_q - \phi) &= 0 \iff \\ \phi &= \phi_q \mod \pi \end{aligned}$$

The distance function $d_{B,q}(\phi)$ thus has two extrema.

The minimum of $d_{B,q}(r)$ on B is then:

$$\min_{r\in B} d_{B,q}(r) = \min\left(\{d_{B,q}(\min_r), d_{B,q}(\max_r)\} \cup \{d_{B,q}(\phi) | \min_\phi \leq \phi \leq \max_\phi \wedge \phi = \phi_q \mod \pi\}\right)$$

The distance $\mathrm{dist}_{\mathcal{H}}(C, q)$ can thus be written as the minimum of four to ten point-to-point distances. Algorithm 2 collects the arguments for these distances in the variable cornerSet and returns the distance minimum as the first return value. □

Lemma 6. *Let T be a polar quadtree in the Euclidean plane, C a quadtree cell of T and q a point in the Euclidean plane. The first value returned by Algorithm 3 is the distance of C to q.*

Proof. The general distance equation for polar coordinates in Euclidean space is

$$f(r_p, r_q, \phi_p, \phi_q) = \sqrt{r_p^2 + r_q^2 - 2r_p r_q \cos(\phi_p - \phi_q)}. \tag{3.10}$$

If the query point q is within C, the distance is zero. Otherwise, the distance between q and C is equal to the distance between q and the boundary of C. We consider each boundary component separately and derive the extrema of the distance function.

Radial Boundary.

When considering the radial boundary, everything but one angle is fixed:

$$f(\phi_p) = \sqrt{r_p^2 + r_q^2 - 2r_p r_q \cos(\phi_p - \phi_q)}. \tag{3.11}$$

Since the distance is positive and the square root is a monotone function, the extrema of the previous function are at the same values as the extrema of its square $g(\phi)$:

$$g(\phi_p) = r_p^2 + r_q^2 - 2r_p r_q \cos(\phi_p - \phi_q). \tag{3.12}$$

If any of r_p and r_q are zero, the distance is independent from ϕ and any angle will work. If not, we set the derivative to zero to find the extrema:

$$\begin{align} g'(\phi_p) = 0 &\iff \tag{3.13}\\ 2r_p r_q \sin(\phi_p - \phi_q) \cdot (\phi_p - \phi_q) = 0 &\iff \tag{3.14}\\ \phi_p = \phi_q \mod \pi&. \tag{3.15}\end{align}$$

Angular Boundary.

Similar to the radial boundary, we fix everything but the radius:

$$f(r_p) = \sqrt{r_p^2 + r_q^2 - 2r_p r_q \cos(\phi_p - \phi_q)}. \tag{3.16}$$

Again, we define a helper function with the same extrema:

$$g(r_p) = r_p^2 + r_q^2 - 2r_p r_q \cos(\phi_p - \phi_q). \tag{3.17}$$

We set the derivative to zero to find the extrema:

$$\begin{align} g'(r_p) &= 0 \iff \tag{3.18}\\ 2r_p - 2r_q \cos(\phi_p - \phi_q) &= 0 \iff \tag{3.19}\\ r_p &= r_q \cos(\phi_p - \phi_q) \Rightarrow \tag{3.20}\\ g(r_p) &= r_p^2 + r_q^2 - 2r_p^2 \tag{3.21}\\ &= r_q^2 - r_p^2 \tag{3.22}\\ &= r_q^2(1 - \cos(\phi_p - \phi_q)). \tag{3.23}\end{align}$$

An extremum of f on the boundary of cell C is either at one of its corners or at the points derived in Eq. (3.15) or Eq. (3.23). If $q \notin c$, the minimum over these points and the corners, as computed by Algorithm 3, is the minimal distance between q and any point in c. If q is in C, the distance is trivially zero. □

These distance bounds support geometric range queries, by avoiding the descent into subtrees which cannot have points in the query area.

3.4 Subquadratic Generation of Threshold Random Hyperbolic Graphs

Algorithm 4 shows a high-level view of how to use a polar quadtree to generate a random hyperbolic graph. It takes as input the number of vertices n, the desired average degree k

Algorithm 4: Graph Generation

```
Input: $n, \overline{k}, \alpha$
Output: $G = (V, E)$
1  $R$ = getTargetRadius($n, \overline{k}, \alpha$);                                /* Eq. (3.24) */
2  $V = n$ vertices;
3  $T$ = empty polar quadtree of radius $g(R)$; /* Map to Poincaré with Eq.(2.7) */
4  for vertex $v \in V$ do
5      draw $\phi_v$ from $\mathcal{U}[0, 2\pi)$;
6      draw $r_{\mathcal{H}}[v]$ with density $f(r) = \alpha \sinh(\alpha r)/(\cosh(\alpha R) - 1)$;   /* Eq.(3.1) */
7      $r_E[v] = g(r_{\mathcal{H}}[v])$;                          /* Map to Poincaré with Eq.(2.7) */
8      insert $v$ into $T$ at $(\phi_v, r_E[v])$;
9  for vertex $v \in V$ do in parallel
10     $C_{\mathcal{H}}$ = hyperbolic circle around $(\phi_v, r_{\mathcal{H}}[v])$ with radius $R$;
11     $C_E$ = transformCircleToEuclidean($C_{\mathcal{H}}$);                     /* Prop. 1 */
12     for vertex $w \in T \cap C_E$) do
13         add $(v, w)$ to $E$;
14 return $G$;
```

and the dispersion parameter α, which controls the power-law exponent γ. The functions getTargetRadius and transformCircleToEuclidean are discussed afterwards.

Overview

As in previous efforts [5], vertex positions are generated randomly (lines 5 and 6). We then map these positions into the Poincaré disk (line 7) and store them in a polar quadtree (line 8). For each vertex u the hyperbolic circle defining the neighborhood is mapped into the Poincaré disk according to Proposition 1 (lines 10-11) – also see Figure 3.1b, where the neighborhood of u consists of exactly the vertices in the light blue Euclidean circle. Edges are then created by executing a Euclidean range query with the resulting circle in the polar quadtree (lines 12-13). We add edges (u, v) from u to each vertex v returned by the Euclidean range query, as the hyperbolic distance between u and v is less than R.

Radius of Hyperbolic Disk

For given values of n, α and R, an approximation of the expected average degree $\overline{k}$ is given by [93, Eq. (22)] and the notation $\xi = (\alpha/\zeta)/(\alpha/\zeta - 1/2)$:

$$\overline{k} = \frac{2}{\pi}\xi^2 n \cdot e^{-\zeta R/2} + \frac{2}{\pi}\xi^2 n \cdot \left(e^{-\alpha R}\left(\alpha \frac{R}{2}\left(\frac{\pi}{4}\left(\frac{\zeta}{\alpha}\right)^2 - (\pi - 1)\frac{\zeta}{\alpha} + (\pi - 2)\right) - 1\right)\right) \tag{3.24}$$

The value of ζ can be fixed while retaining all degrees of freedom in the model [23], we thus assume $\zeta = 1$. We then use binary search with fixed n, α and desired $\overline{k}$ to find an R that gives us a close approximation of the desired average degree $\overline{k}$. Note that the above equation is only an approximation and might give wrong results for extreme values. Our implementation could easily be adapted to skip this step and accept the commonly used [89] parameter C, with $R = 2\ln n + C$ or even accept R directly. For increased usability, we accept the average degree $\overline{k}$ as a parameter in the default version. This approach is implemented in the method `getTargetRadius`.

Circles in the Poincaré disk model.

Neighbors of a query point $u = (\phi_h, r_h)$ lie in a hyperbolic circle around u with radius R. This circle, which we denote as H, corresponds to a Euclidean circle E in the Poincaré disk. The center E_c and radius rad_E of E are in general different from u and R. All points on the boundary of E in the Poincaré disk are also on the boundary of H and thus have hyperbolic distance R from u. Among these points, the two on the ray from the origin through u are straightforward to construct by keeping the angular coordinate fixed and choosing the radial coordinates to match the hyperbolic distance: (ϕ_h, r_{e_1}) and (ϕ_h, r_{e_2}), with $r_{e_1}, r_{e_2} \in [0,1)$, $r_{e_1} \neq r_{e_2}$ and $\mathrm{dist}_{\mathcal{H}}(E_c, (\phi_h, r_e)) = R$ for $r_e \in \{r_{e_1}, r_{e_2}\}$. It follows:

Proposition 1. *E_c is at $(\phi_h, \frac{2r_h}{ab+2})$ and rad_E is $\sqrt{\left(\frac{2r_h}{ab+2}\right)^2 - \frac{2r_h^2 - ab}{ab+2}}$, with $a = \cosh(R) - 1$ and $b = (1 - r_h^2)$.*

This transformation is done in the method `transformCircleToEuclidean`.

Quadtree Range Query

A Euclidean circular range query in the polar quadtree can be handled as in any other geometric data structure – recursively descending the tree, skipping subtrees which are out of range and testing elements in leaf cells individually.

Application to Dynamic Updates

Each vertex neighborhood is computed independently. Thus, an updated vertex position requires only a modification of the quadtree and a single Euclidean range query.

3.4.1 Time Complexity

The time complexity of the generator is determined by the quadtree operations.

Quadtree Range Query.

Neighbors of a vertex u are the vertices within a Euclidean circle constructed according to Proposition 1. Let $\mathcal{N}(u)$ be this neighborhood set in the final graph, thus $\deg(u) := |\mathcal{N}(u)|$. We denote leaf cells that do not have non-leaf siblings as *bottom leaf cells*, see Figure 3.4 for an example.

Lemma 7. *Let T and n be as in Lemma 3. A range query on T returning a point set A will examine at most $\mathcal{O}(\sqrt{n} + |A|)$ bottom leaf cells with probability at least $1 - \frac{1}{n^2}$.*

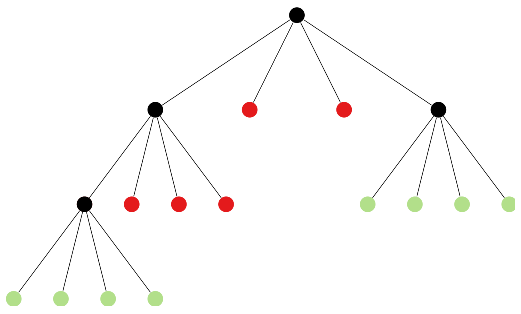


Figure 3.4: Visualization of bottom leaf cells in a quadtree. Bottom leaf cells are marked in green, non-bottom leaf cells in red and interior cells in black.

Proof. For simplicity's sake, we choose a leaf capacity of one for this proof. Since a larger leaf capacity does not increase the tree height and adds only a constant factor to the cost of examining a leaf, this choice does not result in a loss of generality.

Let L be the set of bottom leaf cells containing a vertex in A and let Q be the set of bottom leaf cells examined by the range query. Since the contents of leaf cells are disjoint, $|L| \leq |A|$ holds. The set $Q \backslash L$ consists of leaf cells which are examined by the range query but yield no points in A. These are empty leaf cells within the query circle as well as cells cut by the circle boundary.

Empty leaf cells occur when a previous leaf cell is split since its capacity is exceeded by at least one point. Therefore an empty leaf cell a in the interior of the query circle only occurs when at least two points happened to be allocated to its parent cell b. A split caused by two points creates four leaf cells, therefore there are at most twice as many empty bottom leaf cells as points.

The number of cells cut by the boundary can be bounded with a geometric argument. At depth $k = \lceil \log_4 n \rceil$, at most 4^k cells exist, defined by at most 2^k angular and 2^k radial divisions. When following the circumference of a query circle, each newly cut leaf cell requires the crossing of an angular or radial division. Each radial and angular coordinate occurs at most twice on the circle boundary, thus each division can be crossed at most twice. With two types of divisions, the circle boundary crosses at most $2 \cdot 2 \cdot 2^k = 4 \cdot 2^{\lceil \log_4 n \rceil}$ cells at depth k. Since the value of $4 \cdot 2^{\lceil \log_4 n \rceil}$ is smaller than $4 \cdot 2^{1+\log_4 n}$, this yields $< 8 \cdot \sqrt{n}$ cut cells.

In a balanced tree, all cells at depth k are leaf cells and the bound calculated above is an upper bound for $|Q \backslash L|$. For the general case of an unbalanced tree, we use auxiliary Lemma 8, which bounds to $\mathcal{O}(\sqrt{n})$ the number of bottom leaf cells descendant from cells cut at depth k. □

Lemma 8. *Let $\mathbb{D}_R$ be a disk with radius R in hyperbolic space. Let T be a quadtree on $\mathbb{D}_R$, containing n points distributed according to Section 3.1.1. Let $k := \lceil \log_4 n \rceil$ and let C be a set of $\lfloor c \cdot \sqrt{n} \rfloor$ quadtree cells at depth k, for $c \geq 1$. The total number of bottom leaf cells among the descendants of cells in C is then bounded by $4c \cdot \sqrt{n}$ whp.*

Proof. New leaf cells are only created if a point is inserted in an already full cell. We argue similarly to Lemma 7 that the descendants contain at most twice as many empty bottom leaf cells as points. The number of points in the cells of C is a random variable, which we denote by X. Since each point position is drawn independently and each point is equally likely to land in each cell at a given depth (Lemma 1), X follows a binomial distribution:

$$X \sim \text{Bin}\left(n, \frac{\lfloor c \cdot \sqrt{n} \rfloor}{4^k}\right) \tag{3.25}$$

For ease of calculation, we define a slightly different binomial distribution $Y \sim \text{Bin}\left(n, \frac{c\cdot\sqrt{n}}{n}\right)$. Since $n \leq 4^k$ and $c \cdot \sqrt{n} \geq \lfloor c \cdot \sqrt{n} \rfloor$, the tail bounds calculated for Y also hold for X.

Let $H\left(\frac{2c\cdot\sqrt{n}}{n}, \frac{c\cdot\sqrt{n}}{n}\right)$ be the relative entropy (also known as Kullback-Leibler divergence) of the two Bernoulli distributions $B\left(\frac{2c\cdot\sqrt{n}}{n}\right)$ and $B\left(\frac{c\cdot\sqrt{n}}{n}\right)$. We use a tail bound from [8] to gain an upper bound for the probability that more than $2c$ points are in the cells of C:

$$\Pr(Y \geq 2c \cdot \sqrt{n}) \leq \exp\left(-nH\left(\frac{2c \cdot \sqrt{n}}{n}, \frac{c \cdot \sqrt{n}}{n}\right)\right) \tag{3.26}$$

For consistency with our previous definition of "with high probability", we need to show that $\Pr(Y \geq 2c\sqrt{n}) \leq 1/n$ for n sufficiently large. To do this, we interpret $\Pr(Y \geq 2c \cdot \sqrt{n})/(1/n)$ as an infinite sequence and observe its behavior for $n \to \infty$. Let $a_n := \Pr(Y \geq 2c\cdot\sqrt{n})/(1/n) = n\cdot\Pr(Y \geq 2c\cdot\sqrt{n})$ and $b_n := n\cdot\exp\left(-nH\left(\frac{2c\cdot\sqrt{n}}{n}, \frac{c\cdot\sqrt{n}}{n}\right)\right)$. From Eq. (3.26) we know that $a_n \leq b_n$.

Using the definition of relative entropy, we iterate over the two cases (point within C, point not in C) for both Bernoulli distributions and get:

$$\begin{aligned}
b_n &= n \cdot \exp\left(-nH\left(\frac{2c\sqrt{n}}{n}, \frac{c\sqrt{n}}{n}\right)\right) \\
&= n \cdot \exp\left(-n\left(\left(\frac{2c}{\sqrt{n}}\right)\ln 2 + \left(1 - \frac{2c}{\sqrt{n}}\right)\ln\frac{1-\frac{2c\sqrt{n}}{n}}{1-\frac{c\sqrt{n}}{n}}\right)\right) \\
&= n \cdot \exp\left(-n\frac{2c}{\sqrt{n}}\ln 2\right)\cdot\exp\left(-n\left(1-\frac{2c}{\sqrt{n}}\right)\ln\frac{\sqrt{n}-2c}{\sqrt{n}-c}\right) \\
&= n \cdot \exp\left(-2c\sqrt{n}\ln 2\right)\cdot\exp\left(\left(n - 2c\sqrt{n}\right)\ln\frac{\sqrt{n}-c}{\sqrt{n}-2c}\right) \\
&= n \cdot \frac{1}{2^{2c\sqrt{n}}}\cdot\left(\frac{\sqrt{n}-c}{\sqrt{n}-2c}\right)^{n-2c\sqrt{n}} \\
&= n \cdot \frac{1}{4^{c\sqrt{n}}}\cdot\left(1+\frac{c}{\sqrt{n}-2c}\right)^{n-2c\sqrt{n}}
\end{aligned}$$

(While b_n is undefined for $n \in \{c^2, 4c^2\}$, we only consider *sufficiently large* n from the outset.)

We apply a variant of the root test and consider the limit $\lim_{n\to\infty}(b_n)^{\frac{1}{\sqrt{n}}}$ for an auxiliary result:

$$\begin{aligned}
&\lim_{n\to\infty}\left(n\cdot\frac{1}{4^{c\sqrt{n}}}\cdot\left(1+\frac{c}{\sqrt{n}-2c}\right)^{n-2c\sqrt{n}}\right)^{\frac{1}{\sqrt{n}}}\\
=&\lim_{n\to\infty} n^{\frac{1}{\sqrt{n}}}\cdot\frac{1}{4^c}\cdot\left(1+\frac{c}{\sqrt{n}-2c}\right)^{\sqrt{n}}\left(1+\frac{c}{\sqrt{n}-2c}\right)^{-2c}\\
=&1\cdot\frac{1}{4^c}\cdot e^c\cdot 1=\left(\frac{e}{4}\right)^c
\end{aligned}$$

From $e/4 < 0.7$, $c \geq 1$ and the limit definition, it follows that almost all elements in $(b_n)^{\frac{1}{\sqrt{n}}}$ are smaller than 0.7 and thus almost all elements in b_n are smaller than $0.7^{\sqrt{n}}$. Thus $\lim_{n\to\infty} b_n \leq \lim_{n\to\infty} 0.7^{\sqrt{n}} = 0$. Due to Eq. (3.26), we know that a_n is smaller than b_n for large n, and therefore that the number of points in C is smaller than $2c\cdot\sqrt{n}$ with probability at least $1-\frac{1}{n}$ for n sufficiently large. Again with high probability, this limits the number of non-leaf cells in C to $c\cdot\sqrt{n}$ and thus the number of bottom leaf cells to $4c\cdot\sqrt{n}$, proving the claim. □

Due to Lemma 7, the number of examined bottom leaf cells for a range query around u is in $\mathcal{O}(\sqrt{n}+\deg(u))$ with probability at least $1-\frac{1}{n^2}$. The query algorithm traverses T from the root downward. For each bottom leaf cell b, $\mathcal{O}(h(T))$ inner nodes and non-bottom leaf cells are examined on the path from the root to b. Due to Lemma 3, $h(T)$ is in $\mathcal{O}(\log n)$ whp. The time complexity to gather the neighborhood of a vertex u with degree $\deg(u)$ is thus: $T(\mathrm{RQ}(u)) \in O\left((\sqrt{n}+\deg(u))\cdot\log n\right)$ whp.

Graph Generation.

To generate a graph G from n points, the n positions need to be generated (in $\mathcal{O}(n)$) and inserted into the quadtree (in $\mathcal{O}(n\log n)$ whp, see Lemma 4). The combined complexity of this is $\mathcal{O}(n\log n)$ whp. In the next step, neighbors for all points are extracted. This has a complexity of

$$T(\text{Edges}) = \sum_v O\left(\left(\sqrt{n}+\deg(v)\right)\cdot\log n\right) = O\left(\left(n^{3/2}+m\right)\log n\right) \text{ whp.} \tag{3.27}$$

The complexity bounds for each of the n range queries hold with probability at least $1-\frac{1}{n^2}$, with a union bound we get a probability of at least $1-1/n$ for the above complexity. This dominates the quadtree operations and thus total running time. We conclude:

Theorem 2. *Generating random hyperbolic graphs can be done in $\mathcal{O}((n^{3/2}+m)\log n)$ time whp, i. e. with probability $\geq 1-1/n$ for sufficiently large n.*

Dynamic Updates.

To process an updated vertex position u using the same polar quadtree, we need to first delete the point at its old position (amortized $\mathcal{O}(\log n)$ whp), insert it at its new position (amortized $\mathcal{O}(\log n)$ whp), then execute a range query (in $\mathcal{O}\left((\sqrt{n}+\deg(u))\cdot\log n\right)$ whp).

The range query dominates, leading to a time complexity of $\mathcal{O}\left((\sqrt{n}+\deg(u))\cdot\log n\right)$ whp.

3.5 Probabilistic Neighborhood Queries

In general random hyperbolic graphs, the neighborhood of a vertex is probabilistic. We can extend Algorithm 4 to the general case by using probabilistic queries.

This task of sampling a neighborhood whose elements are probabilistic not only occurs in random hyperbolic graphs. Connection probabilities depending on the distance frequently happen in Euclidean applications as well: The probability that a customer shops at a certain physical store shrinks with increasing distance to it. In disease simulations, if the social interaction graph is unknown but locations are available, disease transmission can be modeled as a random process with infection risk decreasing with distance. Moreover, the wireless connections between units in an ad-hoc network are fragile and collapse more frequently with higher distance.

To generalize these scenarios, we define the notion of a *probabilistic neighborhood* in spatial data sets, both Euclidean and hyperbolic. Let the input dataset consist of a set P of n points in $\mathbb{R}^d$ and a distance metric dist; let a query consist of a query point $q \in \mathbb{R}^d$, and a monotonically decreasing function $f : \mathbb{R}^+ \to [0, 1]$ that maps distances to probabilities. Then, the probabilistic neighborhood $N(q, f)$ of q with respect to f is a random subset of P and each point $p \in P$ belongs to $N(q, f)$ with probability $f(\text{dist}(p, q))$. A straightforward query algorithm for sampling a probabilistic neighborhood would iterate over each point $p \in P$ and sample for each whether it is included in $\text{N}(q, f)$. This has a running time of $\Theta(n \cdot d)$ per query point, which is inefficient for small neighborhoods and can be prohibitive for repeated queries in large data sets. Thus we are interested in a faster algorithm for such a *probabilistic neighborhood query* (PNQ, spoken as "pink").

Based on the polar quadtree introduced in Section 3.3, we describe a baseline version of such a query algorithm (Section 3.5.3). This algorithm introduces the main idea, but is asymptotically not faster than the straightforward approach of probing every distance and throwing a biased coin. In Section 3.5.4, the query algorithm is refined to support faster queries.

Our quadtree operations are defined for planar datasets used with the Euclidean or hyperbolic distance metric (Section 3.3.1), which are thus also the domain for our probabilistic query algorithms. The algorithmic principle is generalizable to higher dimensions and other distance metrics, which remains future work.

Since the neighborhood of a vertex in a random hyperbolic graph is an instance of such a probabilistic neighborhood, we can use a fast PNQ query method to support a generation algorithm for general random hyperbolic graphs. Figure 3.5 shows an example of a probabilistic neighborhood query on a hyperbolic disk with 200 points.

3.5.1 Notation

As mentioned, a point p is in the probabilistic neighborhood of query point q with probability $f(\text{dist}(p, q))$. Thus, a *query pair* consists of a query point q and a function $f : \mathbb{R}^+ \to [0, 1]$ that maps distances to probabilities. The function f needs to be monotonically decreasing but may be discontinuous. Note that f can be defined differently for each

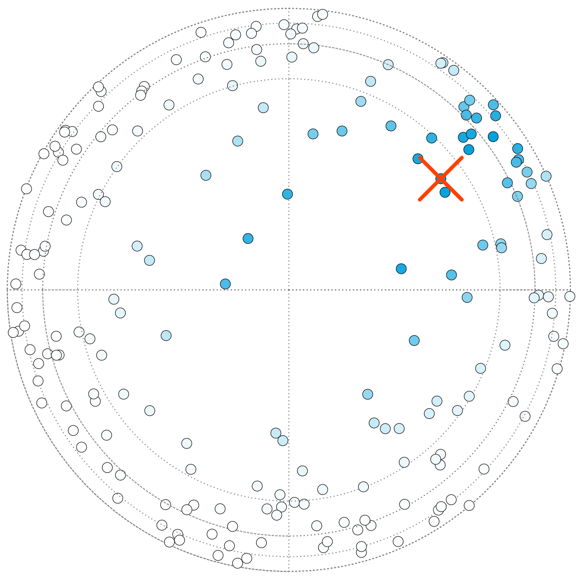


Figure 3.5: Query over 200 points in a polar hyperbolic quadtree, with $f(d) := 1/(e^{(d-7.78)} + 1)$ and the query point q marked by a red cross. Points are colored according to the probability that they are included in the result. Blue represents a high probability, white a probability of zero.

query. The query result, the probabilistic neighborhood of q with respect to f is denoted by the set $\mathrm{N}(q, f) \subseteq P$.

For the algorithm analysis, we use two additional sets for each query (q, f):

- $\mathrm{Candidates}(q, f)$: neighbor candidates examined when executing such a query,
- $\mathrm{Cells}(q, f)$: quadtree cells examined during execution of the query.

Note that $\mathrm{N}(q, f) \subseteq \mathrm{Candidates}(q, f)$ and that the sets $\mathrm{N}(q, f), \mathrm{Candidates}(q, f)$ and $\mathrm{Cells}(q, f)$ are probabilistic, thus theoretical results about their size are usually only with high probability.

3.5.2 Extension of Quadtrees for Probabilistic Neighborhood Queries

The only change in the quadtree data structure required to support probabilistic neighborhood queries is to store at each subtree the number of points contained in it. This does not change the time complexity beyond constant factors, thus Lemma 4 still holds. Figure 3.6 shows such a quadtree, corresponding to the query visualized in Figure 3.5. The number of points in each subtree is a property of the quadtree, but the coloring is a property of a specific query – the one of Figure 3.5.

3.5.3 Baseline Query Algorithm

The baseline version of our query (Algorithm 5) has a time complexity of $\Theta(n)$, but serves as a foundation for the fast version (Section 3.5.4). It takes as input a query point q, a function f and a quadtree cell c. Initially, it is called with the root node of the quadtree

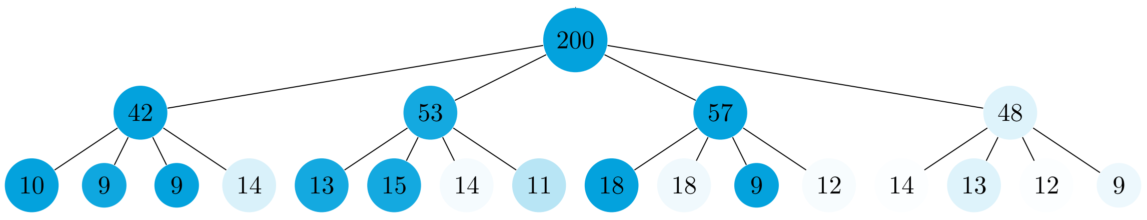


Figure 3.6: Visualization of the data structure used in Figure 3.5. Quadtree nodes are colored according to the upper probability bound for points contained in them. The color of a quadtree node c is the darkest possible shade (dark = high probability) of any point contained in the subtree rooted at c. Each node is marked with the number of points in its subtree.

and recursively descends the tree. As we prove in Prop. 2, the algorithm returns a point set $\mathrm{N}(q, f) \subseteq P$ with

$$\Pr\left(p \in \mathrm{N}(q, f)\right) = f(\mathrm{dist}(q, p)). \tag{3.28}$$

In the following discussion, line numbers refer to lines of pseudocode in Algorithm 5. This query algorithm descends the quadtree recursively until it reaches the leaves. For each leaf l that is reached, a lower bound $\underline{b}$ for the distance between the query point q and all the points in l is computed (Line 2). Since f is monotonically decreasing, this lower bound for the distance gives an upper bound $\overline{b}$ for the probability that a given point in l is a member of the returned point set (Line 3). This bound is used to select *neighbor candidates* in a similar manner as done by Batagelj and Brandes [18]: In Line 9, a random number of vertices is skipped, so that every vertex in l is selected as a neighbor candidate with probability $\overline{\mathrm{b}}$. The actual distance $\mathrm{dist}(q, a)$ between a candidate a and the query point q is at least $\underline{\mathrm{b}}$ and the probability of $a \in \mathrm{N}(q, f)$ thus at most $\overline{\mathrm{b}}$. For each candidate, this actual distance $\mathrm{dist}(q, a)$ is then calculated and a neighbor candidate is confirmed as a neighbor with probability $f(\mathrm{dist}(q, a))/\overline{\mathrm{b}}$ in Line 14.

Regarding correctness of Algorithm 5, we can state:

Proposition 2. *Let T be a quadtree as defined above, q be a query point and $f : \mathbb{R}^+ \to [0, 1]$ a monotonically decreasing function which maps distances to probabilities. The probability that a point p is returned by a PNQ (q, f) from Algorithm 5 is $f(dist(q, p))$, independently from whether other points are returned.*

Proof. Algorithm 5 traverses the whole quadtree with all of its leaves. Since each leaf is examined, we can concentrate on whether the points are sampled correctly within a leaf cell. Our proof thus consists of three steps: 1) The probability that the first point in a leaf is a candidate is $\overline{b}$. 2) Given two points p_i and p_j in the same leaf, the probability that p_i is a candidate is independent of whether p_j is a candidate. 3) The probability that a point p_i is a neighbor of the query point q is given by Eq. (3.28).

Note that the hyperbolic [Euclidean] distances, which are mapped to probabilities according to the function f, are calculated by Algorithm 2 [Algorithm 3] in Section 3.3.1. We continue the current proof with details for all three main steps.

```
Algorithm 5: QuadNode.getProbabilisticNeighborhood
  Input: query point $q$, prob. function $f$, quadtree node $c$
  Output: probabilistic neighborhood of $q$
1 N = {};
2 $\underline{b} = \text{dist}(q, c)$;
  /* Distance between point and cell, returned by Algorithms 2 and 3.
     */
3 $\overline{b}=f(\underline{b})$;
  /* Since $f$ is monotonically decreasing, a lower bound for the
     distance gives an upper bound $\overline{b}$ for the probability.        */
4 $s$ = number of points in $c$;
5 if $c$ is not leaf then
      /* internal node:  descend, add recursive result to local set    */
6     for $child \in children(c)$ do
7         add getProbabilisticNeighborhood($q$, $f$, child) to N;
8 else
      /* leaf case:  sample gaps from geometric distribution           */
9     for $i=0; i < s ; i{+}{+}$ do
10        $\delta = \ln(1 - rand)/\ln(1 - \overline{b})$;
11        $i \mathrel{+}= \delta$;
12        if $i \geq s$ then
13            break;
14        $prob = f(\text{dist}(q, \text{c.points}[i]))/\overline{b}$;
15        add c.points[$i$] to N with probability $prob$
16 return N
```

Step 1

Between two points, the jumping width δ is given by Line 9 of Algorithm 5. The probability that exactly i points are skipped between two given candidates is $(1-\overline{b})^i \cdot \overline{b}$:

$$\Pr(i \leq \delta < i+1) = \Pr(i \leq \ln(1-r)/\ln(1-\overline{b}) < i+1) = \tag{3.29}$$

$$\Pr(\ln(1-r) \leq i \cdot \ln(1-\overline{b}) \wedge \ln(1-r) > (i+1) \cdot \ln(1-\overline{b})) = \tag{3.30}$$

$$\Pr(1-(1-\overline{b})^i \leq r < (1-(1-\overline{b})^{i+1})) = 1-(1-\overline{b})^{i+1} - 1 + (1-\overline{b})^i = \tag{3.31}$$

$$(1-\overline{b})^i(1-(1-\overline{b})) = (1-\overline{b})^i \cdot \overline{b} \tag{3.32}$$

Note that in Eq. (3.29) the denominator is negative, thus the direction of the inequality is reversed in the transformation. The transformation in Eq. (3.31) works since r is uniformly distributed in $[0, 1]$.

Following from Eq. (3.32), the probability is $\overline{b}$ for $i = 0$, and if a point is selected as a candidate, the subsequent point is selected with a probability of $\overline{b}$.

Step 2

Let p_i, p_j and p_l be points in a leaf, with $i < j < l$ and let p_i be a neighbor candidate. For now we assume that no other points in the same leaf are candidates and consider the probability that p_l is selected as a candidate depending on whether the intermediate point p_j is a candidate.

Case 2.1: If point p_j is a candidate, then point p_l is selected if $l - j$ points are skipped after selecting p_j. Due to Step 1, this probability is $(1 - \overline{b})^{l-j} \cdot \overline{b}$

Case 2.2: If point p_j is *not* a candidate, then point p_l is selected if $l - i$ points are skipped after selecting p_i. Given that p_j is not selected, at least $j - i$ points are skipped. The conditional probability is then:

$$\begin{aligned} \Pr(l - i \leq \delta < l - i + 1 | \delta > j - i) &= \\ \Pr(1 - (1 - \overline{b})^{l-i} < r < (1 - (1 - \overline{b})^{l-i+1}) | \delta > j - i) &= \\ (1 - \overline{b})^{l-i} \cdot \overline{b} / (1 - \overline{b})^{j-i} = (1 - \overline{b})^{l-j} \cdot \overline{b} & \end{aligned}$$

As both cases yield the same result, the probability $\Pr(p_l \in \text{Candidates})$ is independent of whether p_j is a candidate.

Step 3

Let C be a leaf cell in which all points up to point p_i are selected as candidates. Due to Step 1, the probability that p_{i+1} is also a candidate, meaning no points are skipped, is $(1 - \overline{b})^0 \cdot \overline{b} = \overline{b}$. Due to Step 2, the probability of p_{i+1} being a candidate is independent of whether p_i is a candidate. This can be applied iteratively until the beginning of the leaf cell, yielding a probability of $\overline{b}$ for p_i being a candidate, independent of whether other points are selected.

A neighbor candidate p_i is accepted as a neighbor with probability $f(\text{dist}(p_i, q))/\overline{b}$ in Line 14. Since $\overline{b}$ is an upper bound for the neighborhood probability, the acceptance ratio is between 0 and 1. The probability for a point p to be in the probabilistic neighborhood computed by Algorithm 5 is thus:

$$\begin{aligned} \Pr(p \in \text{N}(q, f)) = \Pr(p \in \text{N}(q, f) \wedge p \in \text{Candidates}(q, f)) &= \\ \Pr(p \in \text{N}(q, f) | p \in \text{Candidates}(q, f)) \cdot \Pr(p \in \text{Candidates}(q, f)) &= \\ f(\text{dist}(p, q))/\overline{b} \cdot \overline{b} = f(\text{dist}(p, q)) & \end{aligned}$$

□

Since Algorithm 5 examines the complete quadtree, its time complexity is at least linear. We omit a more thorough analysis until the next section, in which we show how to accelerate the query process.

3.5.4 Queries in Sublinear Time by Subtree Aggregation

One reason for the linear time complexity of the baseline query is the fact that every quadtree node is visited. To reach a sublinear time complexity, we thus aggregate subtrees into *virtual leaf cells* whenever doing so reduces the number of examined cells and does not increase the number of candidates too much.

```
Algorithm 6: QuadNode.getProbabilisticNeighborhood
   Input: query point $q$, prob. function $f$, quadtree node $c$
   Output: probabilistic neighborhood of $q$
1  N = {};
2  $\underline{b} = \mathrm{dist}(q, c)$;
   /* Distance between point and cell, returned by Algorithms 2 and 3.
      */
3  $\overline{b} = f(\underline{b})$;
   /* Since $f$ is monotonically decreasing, a lower bound for the
      distance gives an upper bound $\overline{b}$ for the probability.        */
4  $s$ = number of points in $c$;
5  if $c$ is inner node and $|c| \cdot \overline{b} \geq 1$ then
       /* internal node:  descend, add recursive result to local set    */
6      for child ∈ children(c) do
7          add getProbabilisticNeighborhood($q$, $f$, child) to N;
8  else
       /* leaf case:  sample gaps from geometric distribution           */
9      for $i$=0; $i < s$ ; $i$++ do
10         $\delta = \ln(1 - rand)/\ln(1 - \overline{b})$;
11         $i$ += $\delta$;
12         if $i \geq s$ then
13             break;
14         neighbor = maybeGetKthElement($q$, $f$, $i$, $\overline{b}$, $c$);
15         add neighbor to N if not empty set;
16 return N
```

To this end, let S be a subtree starting at depth l of a quadtree T. During the execution of Algorithm 5, a lower bound $\underline{b}$ for the distance between S and the query point q is calculated, yielding also an upper bound $\overline{b}$ for the neighbor probability of each point in S. At this step, it is possible to treat S as a *virtual leaf cell*, sample jumping widths using $\overline{b}$ as upper bound and use these widths to select candidates within S. Algorithm 7 is used in a virtual leaf cell where the candidate confirmation (Line 14 of Algorithm 5) happens in an original leaf cell. Aggregating a subtree to a virtual leaf cell allows skipping leaf cells which do not contain candidates, but uses a weaker bound $\overline{b}$ and thus a potentially larger candidate set. Thus, a fast algorithm requires an aggregation criterion which keeps both the number of candidates and the number of examined quadtree cells low. [8] As stated before, we record the number of points in each subtree during quadtree construction. This information is now used for the query algorithm: We aggregate a subtree S to a virtual leaf cell exactly if $|S|$, the number of points contained in S, is below $1/f(\mathrm{dist}(S, q))$. This corresponds to less than one expected candidate within S. The changes required in Algorithm 5 to use the subtree aggregation are minor. Lines 5, 14 and 15 in the original algorithm are changed to use subtrees, shown in Algorithm 6 and marked in blue.

The main change consists in the use of the function maybeGetKthElement (Algorithm 7).

[8] In the extreme case, candidates are selected directly at the root. In this case, the distance to the query point is 0 and the probability bound $\overline{b}$ is $f(0)$, resulting in linearly many candidates.

Algorithm 7: maybeGetKthElement

```
Input: query point $q$, function $f$, index $k$, bound $\bar{b}$, subtree $S$
Output: $k$th element of $S$ or empty set
if $k \geq |S|$ then
    return $\emptyset$;
if $S.isLeaf()$ then
    acceptance = $f(\mathrm{dist}(q, \mathrm{S.positions}[k]))/\bar{b}$;
    if $1 - rand() < acceptance$ then
        return S.elements[$k$];
    else
        return $\emptyset$;
else
    /* Recursive call */
    offset := 0;
    for $child \in S.children$ do
        if $k$ − offset < |child| then
            /* |child| is the number of points in child */
            return maybeGetKthElement($q$, $f$, $k$ - offset, $\bar{b}$, child);
        offset += |child|;
```

Given a subtree S, an index k, q, f, and $\bar{b}$, the algorithm descends S to the leaf cell containing the kth element. This element p_k is then accepted with probability $f(\mathrm{dist}(q, p_k))/\bar{b}$.

Since the upper bound calculated at the root of the aggregated subtree is not smaller than the individual upper bounds at the original leaf cells, Proposition 2 also holds for the virtual leaf cells. Points are thus included in the result with the correct probabilities.

3.5.5 Query Time Complexity

Our main analytical result of this section concerns the time complexity of the faster query algorithm. Its proof relies on several lemmas presented afterwards.

Theorem 3. *Let T be a quadtree with n points and (q, f) a query pair. A query (q, f) using subtree aggregation has time complexity $\mathcal{O}((|\mathrm{N}(q, f)| + \sqrt{n}) \log n)$ whp.*

Proof. Similar to the baseline algorithm, the complexity of the faster query is determined by the number of recursive calls and the total number of loop iterations across the calls. The first corresponds to the number of examined quadtree cells, the second to the total number of candidates. With subtree aggregation, we obtain improved bounds: Lemma 11 limits the number of candidates to $\mathcal{O}(|\mathrm{N}(q, f)| + \sqrt{n})$ whp, while Lemma 12 bounds the number of examined quadtree cells to $\mathcal{O}((|\mathrm{N}(q, f)| + \sqrt{n}) \log n)$ whp. Together, this results in a query complexity of $\mathcal{O}((|\mathrm{N}(q, f)| + \sqrt{n}) \log n)$ whp. $\square$

For the lemmas required in the proof of Theorem 3 we need some additional notation: Let T be a quadtree with n points, S a subtree of T containing s points, q a query point and f a function mapping distances to probabilities. The set of neighbors $(\mathrm{N}(q, f))$, candidates $(\mathrm{Candidates}(q, f))$ and examined cells $(\mathrm{Cells}(q, f))$ are defined as in Section 3.5.1.

For the analysis we divide the space around the query point q into infinitely many bands, based on the probabilities given by f. A point $p \in \mathrm{P}$ is in band i exactly if the probability of it being a neighbor of q is between $2^{-(i+1)}$ and 2^{-i}:

$$p \in \text{band } i \iff 2^{-(i+1)} < f(\text{dist}(p, q)) \leq 2^{-i}$$

Based on these bands, we divide the previous sets into infinitely many subsets:

- $\mathrm{P}(q, f, i) := \{v \in \mathrm{P} | 2^{-(i+1)} < f(\text{dist}(v, q)) \leq 2^{-i}\}$
- $\mathrm{N}(q, f, i) := \mathrm{N}(q, f) \cap \mathrm{P}(q, f, i)$
- $\text{Candidates}(q, f, i) := \text{Candidates}(q, f) \cap \mathrm{P}(q, f, i)$
- $\text{Cells}(q, f, i) := \{c \in \text{Cells}(q, f) | 2^{-(i+1)} < f(\text{dist}(c, q)) \leq 2^{-i}\}$

Note that for fixed n, all but at most finitely many of these sets are empty. We call the quadtree cells in $\text{Cells}(q, f, i)$ to be *anchored* in band i. The region covered by a quadtree cell is in general not aligned with the probability bands, thus a quadtree cell anchored in band i ($c \in \text{Cells}(q, f, i)$) may contain points from higher bands (i.e. with lower probabilities).

We continue with two auxiliary results used in Lemma 11.

Lemma 9. *Let T be a polar hyperbolic [Euclidean] quadtree with n points and $s < n$ a natural number. Let Λ be a circle in the hyperbolic [Euclidean] plane and let $\ominus$ be the disjoint set of subtrees of T that contain at most s points and are cut by Λ. Then, the subtrees in $\ominus$ contain at most $24\sqrt{n \cdot s}$ points with probability at least $1 - 0.7^{\sqrt{n}}$ for n sufficiently large.*

Proof. Let $k := \lfloor \log_4 n/s \rfloor$ be the minimal depth at which cells have at least s points in expectation. At most 4^k cells exist at depth k, defined by at most 2^k angular and 2^k radial divisions. When following the circumference of the query circle Λ, each newly cut cell requires the crossing of an angular or radial division. Each radial and angular coordinate occurs at most twice on the circle boundary, thus each division can be crossed at most twice. With two types of divisions, Λ crosses at most $2 \cdot 2 \cdot 2^k = 4 \cdot 2^{\lfloor \log_4 n/s \rfloor}$ cells at depth k. Since the value of $4 \cdot 2^{\lfloor \log_4 n/s \rfloor}$ is at most $4 \cdot 2^{\log_4 n/s}$, this yields $\leq 8 \cdot \sqrt{n/s}$ cut cells. We denote the set of cut cells with ς. Since the cells in ς cover the circumference of the circle Λ, a subtree S which is cut by Λ is either contained within one of the cells in ς, corresponds to one of the cells or contains one. In the first two cases, all points in S are within the cells of ς. In the second case, at least one cell of ς is contained in S. As the subtrees are disjoint, this cell cannot be contained in any other of the considered subtrees. Thus, there are no more subtrees containing points not in ς than there are cells in ς, which are less than $8 \cdot \sqrt{n/s}$ many.

Due to Lemma 1, the probability that a given point is in a given cell at level k is 4^{-k}. The number of points contained in cells of ς thus follows a binomial distribution $B(n, p)$. An upper bound for the probability p is given by $\frac{8 \cdot \sqrt{ns}}{n}$, thus a tail bound for a slightly different distribution $B(n, \frac{8\cdot\sqrt{ns}}{n})$ also holds for $B(n,p)$. In the proof of Lemma 8, a similar distribution is considered. Setting the variable c to $8\sqrt{s}$, we see that the probability of ς containing more than $16 \cdot \sqrt{sn}$ points is smaller than $0.7^{\sqrt{n}}$.

The subtrees in $\ominus$ contain at most s points by definition, thus an upper bound for the number of points in these subtrees is given by $s \cdot 8 \cdot \sqrt{n/s}$ (points not in ς) + $16 \cdot \sqrt{sn}$ (points in ς). This results in at most $24 \cdot \sqrt{sn}$ points contained in $\ominus$ with probability at least $1 - 0.7^{\sqrt{n}}$. □

Lemma 10. *Let n be a natural number and let A, B be random sets with $A \subseteq B, |B| \leq n$ and the following property:* $\Pr(b \in A) \geq 0.5$, $\forall b \in B$. *Further, let the probabilities for membership in A be independent. Then, the number of points in B is in $\mathcal{O}(|A| + \log n)$ with probability at least $1 - 1/n^3$.*

Proof. Let $X = |A|$ be a random variable denoting the size of A. Since the individual probabilities for membership in A might be different, X does not necessarily follow a binomial distribution. We define an auxiliary binomial distribution $Y \sim \mathrm{Bin}(|B|, 0.5)$. Since all membership probabilities for A are at least 0.5, X has first-order stochastic dominance [136] over Y and lower tail bounds derived for Y also hold for X.

The probability that Y is less than $0.1|B|$ is then [72]:

$$\Pr(Y < 0.1|B|) \leq \exp\left(-2\frac{(0.5|B| - 0.1|B|)^2}{|B|}\right) = \exp\left(-0.32|B|\right).$$

If $|B| \leq 10\log n$, then $|B|$ is trivially in $\mathcal{O}(\log n)$, otherwise the probability $\Pr(|A| < 0.1|B|)$ is

$$\Pr(|A| < 0.1|B|) \leq \Pr(Y < 0.1|B|) \tag{3.33}$$
$$\leq \exp\left(-3.2\log n\right) = n^{-3.2} < 1/n^3. \tag{3.34}$$

Thus $|B| \leq 10|A| \in \mathcal{O}(|A|)$ with probability at least $1 - 1/n^3$. □

The following Lemmata 11 and 12 bound the number of examined candidates and examined quadtree cells, concluding the proof of Theorem 3.

Lemma 11. *Let T be a quadtree with n points and (q, f) a query pair. The number of candidates examined by a query using subtree aggregation is in $\mathcal{O}(|\mathrm{N}(q,f)| + \sqrt{n})$ whp.*

Proof. For the analysis we consider each probability band i separately. As defined above, band i contains points with a neighbor probability of $2^{-(i+1)}$ to 2^{-i}. Among the cells anchored in band i, some are original leaf cells and others are virtual leaf cells created by subtree aggregation. The virtual leaf cells contain less than one expected candidate and thus less than 2^{i+1} points. The capacity of the original leaf cells is constant. All the points in cells anchored in band i have a probability between $2^{-(i+1)}$ and 2^{-i} to be a candidate.

Among the points in virtual or original leaf cells, some are in the same band their cell is anchored in, others are in higher cells.

We divide the set of points within cells anchored in band i into four subsets:

1. points in band i and in original leaf cells
2. points in band i and in virtual leaf cells
3. points not in band i and in original leaf cells
4. points not in band i and in virtual leaf cells

The points in the first two sets are unproblematic. Since the probability that a point in these sets is a neighbor is at least $2^{-(i+1)}$, the probability for a given candidate to be a neighbor is at least $\frac{1}{2}$. Due to Lemma 10, the number of candidates in these sets is in $\mathcal{O}(|\mathrm{N}(q,f)| + \log n)$ whp, which is in $\mathcal{O}(|\mathrm{N}(q,f)| + \sqrt{n})$ whp.

Points in the third set are in cells cut by the boundary between band i and band $i+1$. Since the probabilities are determined by the distance, this boundary is a circle and we can use Lemma 9 to bound the number of points to $24\sqrt{n \cdot \text{capacity}}$ with probability at least $1 - 0.7^{\sqrt{n}}$ for n sufficiently large. The mentioned capacity is the capacity of the original leaf cells.

Likewise, points in the fourth set are in virtual leaf cells cut by the boundary between bands i and $i+1$. A virtual leaf cell, which is an aggregated subtree, contains at most 2^{i+1} points, otherwise it would not have been aggregated. Again, using Lemma 9, we can bound the number of points in these sets to $24\sqrt{n \cdot 2^{i+1}}$ points with probability at least $1 - 0.7^{\sqrt{n}}$.

We denote the union of the third and fourth sets with $\text{Overhang}(q,f,i)$. From the individual bounds derived in the previous paragraphs, we obtain an upper bound for the number of points in $\text{Overhang}(q,f,i)$ of $24(\sqrt{n \cdot \text{capacity}} + \sqrt{n \cdot 2^{i+1}})$ with probability at least $(1 - 0.7^{\sqrt{n}})^2$. Simplifying the bound, we get that $|\text{Overhang}(q,f,i)| \leq 24\sqrt{n} \cdot (2^{(i+1)/2} + \sqrt{\text{capacity}})$ with probability at least $1 - 2 \cdot 0.7^{\sqrt{n}}$.

Each of the points in $\text{Overhang}(q,f,i)$ is a candidate with a probability between 2^{-i} and $2^{-(i+1)}$. The candidates are sampled independently (see Step 2 of Lemma 2). While different points may have different probabilities of being a candidate and the total number of candidates does not follow a binomial distribution, we can bound the probabilities from above with 2^{-i}.

We proceed towards a Chernoff bound for the total number of candidates across all overhangs. Let X_i denote the random variable representing the number of candidates within $|\text{Overhang}(q,f,i)|$ and let $X = \sum_{i=0}^{\infty} X_i$ denote the total number of candidates in overhangs.

The expected value $\mathbb{E}(X)$ follows from the linearity of expectations:

$$\begin{aligned}
\mathbb{E}(X) &= \sum_{i=0}^{\infty} \mathbb{E}(X_i) \\
&= \sum_{i=0}^{\infty} (24\sqrt{n} \cdot (2^{(i+1)/2} + \sqrt{\text{capacity}}) \cdot 2^{-i}) \\
&= 24\sqrt{n} \sum_{i=0}^{\infty} \sqrt{2} \cdot 2^{-i/2} + 2^{-i}\sqrt{\text{capacity}} \\
&= 24\sqrt{n} \left(\sum_{i=0}^{\infty} \sqrt{2} \cdot 2^{-i/2} + 2^{-i}\sqrt{\text{capacity}} \right) \\
&= 24\sqrt{n} \left(\sqrt{2} \sum_{i=0}^{\infty} \left(2^{-i/2}\right) + \sum_{i=0}^{\infty} 2^{-i}\sqrt{\text{capacity}} \right) \\
&= 24\sqrt{n} \left(\sqrt{2} \sum_{i=0}^{\infty} \left(2^{-i/2}\right) + 2\sqrt{\text{capacity}} \right) \\
&= 24\sqrt{n} \left(\sqrt{2} \left(\sum_{i=0}^{\infty} \left(2^{-2i/2}\right) + \sum_{i=0}^{\infty} \left(2^{-(2i+1)/2}\right) \right) + 2\sqrt{\text{capacity}} \right) \\
&= 24\sqrt{n} \left(\sqrt{2} \left(\sum_{i=0}^{\infty} (2^{-i}) + \sum_{i=0}^{\infty} \left(\frac{1}{\sqrt{2}} \cdot 2^{-i} \right) \right) + 2\sqrt{\text{capacity}} \right) \\
&= 24\sqrt{n} \left(\sqrt{2} \left(2 + \frac{1}{\sqrt{2}} \cdot 2 \right) + 2\sqrt{\text{capacity}} \right) \\
&= 24\sqrt{n} \left(2\sqrt{2} + 2 + 2\sqrt{\text{capacity}} \right) \\
&= 48\sqrt{n} \left(1 + \sqrt{2} + \sqrt{\text{capacity}} \right)
\end{aligned}$$

(Cells anchored in the band ∞, which has an upper bound $\bar{b}$ of zero for the neighborhood probability, do not have any candidates and can be omitted here.)

Since the candidates are sampled independently with a probability of at most 2^{-i}, we can treat X as a sum of independent Bernoulli random variables. This allows us to use a multiplicative Chernoff bound [116] and we can now give an upper bound for the probability that the overhangs contain more than twice as many candidates as expected:

$$\begin{aligned}
\Pr(X > 2\mathbb{E}(X)) &\leq \left(\frac{e}{2^2} \right)^{\mathbb{E}(X)} \\
&= \left(\frac{e}{2^2} \right)^{48\sqrt{n}\left(1+\sqrt{2}+\sqrt{\text{capacity}}\right)} \\
&\leq \left(\frac{e}{2^2} \right)^{\sqrt{n}} \\
&\leq 0.7^{\sqrt{n}}
\end{aligned}$$

Including this last one, we have a chain of $2n+1$ tail bounds, each with a probability of at least $(1 - 0.7^{\sqrt{n}})$. The event that any of these tail bounds is violated is a union over each event that a specific tail bound is violated. With a union bound [116, Lemma 1.2], the probability that any of the individual tail bounds is violated is at most $(2n+1)0.7^{\sqrt{n}}$. Since $\left((2n+1)0.7^{\sqrt{n}}\right)^{-1}$ grows faster than n for n sufficiently large, we conclude that the total number of candidates is thus bounded by $\mathcal{O}(|\mathrm{N}(q,f)|)+96\sqrt{n}\left(1+\sqrt{2}+\sqrt{\text{capacity}}\right))$ with probability at least $(1-1/n)$ for n sufficiently large. The leaf capacity is constant, thus the number of candidates evaluated during execution of a query (q,f) is in $\mathcal{O}(|\mathrm{N}(q,f)|+\sqrt{n})$ whp. □

We proceed with a lemma necessary for bounding the number of examined quadtree cells in a query.

Lemma 12. *Let T be a quadtree with n points and (q,f) a query pair. The number of quadtree cells examined by a query using subtree aggregation is in $\mathcal{O}((|\mathrm{N}(q,f)|+\sqrt{n})\log n)$.*

To prove Lemma 12, we first need to introduce another auxiliary lemma:

Lemma 13. *Let $\mathbb{D}_R$ be a hyperbolic or Euclidean disk of radius R and let T be a polar quadtree on $\mathbb{D}_R$ containing n points distributed according to the probability distributions assumed when constructing the quadtree. Let $\Upsilon(q,f)$ be the set of visited, unaggregated quadtree cells that have only (virtual) leaf cells as children. With a query using subtree aggregation, $|\Upsilon(q,f)|$ is in $\mathcal{O}(|\mathrm{N}(q,f)|+\sqrt{n})$ whp.*

Proof. Let $c \in \Upsilon(q,f,i)$ be such an unaggregated quadtree cell anchored in band i that has only original or virtual leaf cells as children. It contains at least 2^i points and has four children, of which at least one is also anchored in band i. We denote this (virtual) leaf anchored in band i with l. Since each child of c contains the same probability mass (Lemma 1), each point of c is in l with probability 1/4:

$$\Pr(p \in l | p \in c) = \frac{1}{4}. \tag{3.35}$$

A point in l is a candidate (in l) with probability $f(\mathrm{dist}(q,l))$, which is between $2^{-(i+1)}$ and 2^{-i} since l is anchored in band i. The probability that a given point $p \in c$ is a candidate in l is then

$$\Pr(p \in l \wedge p \in \mathrm{Candidates}(q,f,i) | p \in c) = \frac{1}{4} \cdot f(\mathrm{dist}(q,l)) \geq 2^{-(i+3)} \tag{3.36}$$

Since the point positions and memberships in Candidates(q,f,i) are independent, we can bound the number of candidates in l with a binomial distribution $\mathrm{Bin}(|c|, 2^{-(i+3)})$. The probability that l contains no candidates is:

$$f\left(0, |c|, \frac{1}{8}\cdot 2^{-i}\right) = \left(1 - \frac{1}{8}\cdot 2^{-i}\right)^{|c|} \leq \left(1 - \frac{1}{8}\cdot\frac{1}{2^i}\right)^{2^i} \tag{3.37}$$

Considered as a function of i, this probability is monotonically increasing. In the limit of $2^i \to \infty$, it trends to $\exp(-1/8) \approx 0.88$, a value it never exceeds. The probability that the cell c contains at least one candidate is then above $1 - \frac{1}{\sqrt[8]{e}} > 0.1$.

For each cell in Υ, the probability that it contains at least one candidate is > 0.1. Let X be the random variable denoting the number of cells in Υ that contain at least one candidate. We define an auxiliary binomial distribution $\mathrm{Bin}(|\Upsilon|, 0.1)$ and use a tail bound to estimate the number of cells in Υ containing candidates. Let $Y \propto \mathrm{Bin}(|\Upsilon|, 0.1)$ be a random variable distributed according to this auxiliary distribution.

We use a tail bound from Arratia and Gordon [8] to limit the probability that $Y < 0.05|\Upsilon|$ to at most $\exp(-|\Upsilon|/80)$. Since 0.1 was a lower bound for the probability that a cell contains a candidate, this tail bound also holds for X. The probability that the set of Υ contains at least $0.05|\Upsilon|$ many candidates is then at least $(1 - \exp(-|\Upsilon|/80))$.

That $|\Upsilon| \in \mathcal{O}(\sqrt{n})$ (whp) now follows from a case distinction:

- Case $|\Upsilon| \in \mathcal{O}(\sqrt{n})$: Then it trivially follows that $|\Upsilon| \in \mathcal{O}(\sqrt{n})$ with probability at least $1 - 1/n$.
- Case $|\Upsilon| \in \omega(\sqrt{n})$: The probability $(1 - \exp(-|\Upsilon|/80))$ is then larger than $(1 - \exp(-\sqrt{n}/80))$, which is $> 1 - 1/n$ for sufficiently large n. Thus the number of examined quadtree cells during a query is linear in the number of candidates. Due to Lemma 11, this is in $\mathcal{O}(|\mathrm{N}(q, f)| + \sqrt{n})$ whp.

$\square$

The proof of Lemma 12 then follows easily:

Proof. We split the set of examined quadtree cells into three categories:

- leaf cells and root nodes of aggregated subtrees ($C1$)
- parents of cells in the first category ($C2$)
- all other ($C3$)

The third category ($C3$) then exclusively consists of inner nodes in the quadtree. When following a chain of nodes in category $C3$ from the root downwards, it ends with a node in category $C2$. The size $|C3|$ is thus at most $\mathcal{O}(|C2| \log n)$ whp, since the number of elements in a chain cannot exceed the height of the quadtree, which is $\mathcal{O}(\log n)$ by Lemma 3.

With a branching factor of 4, $|C1| = 4|C2|$ holds.

The number of cells in category $C2$ can be bounded using Lemma 13 to $\mathcal{O}(|\mathrm{N}(q, f)| + \sqrt{n})$ with high probability. The total number of examined cells is thus in $\mathcal{O}((|\mathrm{N}(q, f)| + \sqrt{n}) \log n)$. $\square$

3.6 Near-Linear Time Generation of Threshold Random Hyperbolic Graphs

The time complexity of a query using polar quadtrees cannot easily get below $\mathcal{O}(\sqrt{n})$, since a query circle might cut up to $\mathcal{O}(\sqrt{n})$ quadtree cells. To improve on this, we divide $\mathbb{D}_R$, the disk in the hyperbolic plane, into ring-shaped slabs and use these to bound the coordinates of possible neighbors in each slab.

3.6.1 Data Structure

For a given hyperbolic disk $\mathbb{D}_R$ of radius R, let $C = \{c_0, c_1, ...c_{\max}\}$ be a set of $\lceil \log n \rceil$ ordered radial boundaries, with $c_0 = 0$ and $c_{\max} = R$. We then define a *slab* S_i as the area enclosed by c_i and c_{i+1}. A point $p = (\phi_p, r_p)$ is contained in slab S_i exactly if $c_i \leq r_p < c_{i+1}$. Since slabs are ring-shaped, they partition $\mathbb{D}_R$:

$$\mathbb{D}_R = \bigcup_{i=0}^{\lceil \log n \rceil} S_i.$$

The choice of radial boundaries is an important tuning parameter. After experimenting with different divisions, we settled on a geometric sequence with ratio $p = 0.9$. The relationship between successive boundary values is then: $c_{i+1} - c_i = p \cdot (c_i - c_{i-1})$. From $c_0 = 0$ and $c_{\max} = R$, we derive the value of c_1:

$$\sum_{k=0}^{\log n - 1} c_1 p^k = R \iff c_1 \frac{1 - p^{\log n}}{1 - p} = R \iff c_1 = \frac{(1-p)R}{1 - p^{\log n}}$$

The remaining values follow geometrically.

Figure 3.7 shows an example of a graph in the hyperbolic plane, together with slab S_i. The neighbors of the bold blue vertex v are those within a hyperbolic circle of radius R (0.2R

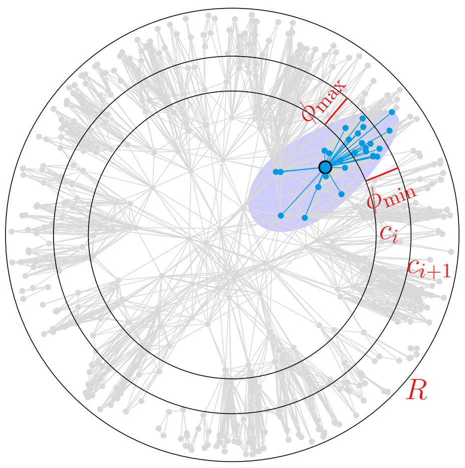


Figure 3.7: Graph in hyperbolic geometry with unit-disk neighborhood. Neighbors of the bold blue vertex are in the hyperbolic circle, marked in blue.

in this visualization), marked by the blue egg-shaped area. When considering vertices in S_i as possible neighbors of v, the algorithm only needs to examine vertices whose angular coordinate is between $\phi_{\min}$ and $\phi_{\max}$.

3.6.2 Generation Algorithm

Algorithm 8: Graph Generation

```
   Input: number of vertices $n$, average degree $\overline{k}$, power-law exponent $\gamma$
   Output: $G = (V, E)$
1  $\alpha = (\gamma - 1)/2$;
2  $R = \text{getTargetRadius}(n, \overline{k}, \alpha)$;
3  $V = n$ vertices;
4  $C = \{c_0, c_1, ...c_{\max}\}$ set of $\log n$ ordered radial coordinates, with $c_0 = 0$ and
   $c_{\max} = R$;
5  $B = \{b_0, b_1, ...b_{\max}\}$ set of $\log n$ empty sets;
6  for vertex $v \in V$ do in parallel
7      draw $\phi_v$ from $\mathcal{U}[0, 2\pi)$;
8      draw $r_v$ with density $f(r) = \alpha \sinh(\alpha r)/(\cosh(\alpha R) - 1)$;
9      insert $(\phi_v, r_v)$ in suitable $b_i$ so that $c_i \leq r_v \leq c_{i+1}$;
10 for $b \in B$ do in parallel
11     sort points in $b$ by their angular coordinates;
12 for vertex $v \in V$ do in parallel
13     for band $b_i \in B$, where $c_{i+1} > r_v$ do
14         $\min_\phi$, $\max_\phi$ = getMinMaxPhi$(\phi_v, r_v), c_i, c_{i+1}, R)$;
15         for vertex $w \in b_i$, where $\min_\phi \leq \phi_w \leq \max_\phi$ do
16             if $dist_{\mathcal{H}}(v, w) \leq R$ then
17                 add $(v, w)$ to $E$;
18 return $G$;
```

Algorithm 8 shows the generation of $G = (V, E)$ with average degree k and power-law exponent γ. The first part is similar to Algorithm 4: Vertex positions are drawn randomly and then inserted into the respective data structure.

Vertex Positions and Bands

After settling the disk boundary, the radial boundaries c_i are calculated (Line 4) as defined above, the disk $\mathbb{D}_R$ is thus partitioned into $\log n$ slabs. For each slab S_i, a set b_i stores the vertices located in the area of S_i. These sets b_i are initially empty (Line 5).

The vertex positions are then sampled randomly in polar coordinates (Lines 7 and 8) and stored in the corresponding set, i.e, vertex v is put into set b_i iff $c_i \leq r_v < c_{i+1}$ (Line 9). Within each set, vertices are sorted with respect to their angular coordinates (Lines 10 to 11).

getMinMaxPhi

The neighbors of a given vertex $v = (\phi_v, r_v)$ are those whose hyperbolic distance to v is at most R. Let b_i be the slab between c_i and c_{i+1}, and $u = (\phi_u, r_u) \in b_i$ a neighbor of v

in b_i. Since u is in b_i, r_u is between c_i and c_{i+1}. With the hyperbolic law of cosines, we conclude:

$$\begin{aligned}
\cosh R \geq \cosh r_v \cosh c_i - \sinh r_v \sinh c_i \cos|\phi_u - \phi_v| &\iff \\
\cosh r_v \cosh c_i - \cosh R \leq \sinh r_v \sinh c_i \cos|\phi_u - \phi_v| &\iff \\
\cos|\phi_u - \phi_v| \geq \frac{\cosh r_v \cosh c_i - \cosh R}{\sinh r_v \sinh c_i} &\iff \\
|\phi_u - \phi_v| \leq \cos^{-1}\left(\frac{\cosh r_v \cosh c_i - \cosh R}{\sinh r_v \sinh c_i}\right)&
\end{aligned}$$

To gather the neighborhood of a vertex $v = (\phi_v, r_v)$, we iterate over all slabs S_i and compute for each slab how far the angular coordinate ϕ_q of a possible neighbor in b_i can deviate from ϕ_v (Line 14). We call the vertices in b_i whose angular coordinates are within these bounds the *neighbor candidates* for v in b_i.

Since points are sorted according to their angular coordinates, we can quickly find the leftmost and rightmost neighbor candidate in each slab using binary search. We then only need to check each neighbor candidate (Line 15), compute its hyperbolic distance to v and add an edge if this distance is below R (Lines 16 and 17). Since edges can be found from both ends, we only need to iterate over slabs in one direction; we choose outward in our implementation (Line 3). The process is repeated for every vertex v (Line 2).

3.6.3 Time Complexity

The time complexity of Algorithm 8 depends on the quality of the angular bounds. We conjecture a time complexity of $\mathcal{O}(n \log^2 n + m)$, due to the following steps: Allocating initial data structures (Lines 2 to 5) takes time at most linear in n. Generation of coordinates (Lines 6 to 9) takes constant time per vertex, finding the correct band takes $\mathcal{O}(\log\log n)$ per vertex with binary search.

For each of the n vertices and each of the $\log n$ bands, we compute the angular bounds and search the closest points in the bands using binary search. This has a complexity of $\mathcal{O}(\log n \cdot \log \frac{n}{\log n}) \subseteq \mathcal{O}(\log^2 n)$ per vertex. If the number of neighbor candidates investigated in Line 15 is at most a linear overhead over the number of true neighbors, as the measurements seem to suggest, this results in a complexity of $\mathcal{O}(\log^2 n + \deg v)$ for sampling the neighborhood of a vertex v and thus a total complexity of $\mathcal{O}(n \log^2 n + m)$. This fits to our experimental measurements (Section 3.8) and was later proven by Manuel Penschuck [132].

3.7 Near-Linear Time Generation of General Random Hyperbolic Graphs

In (Section 3.5), we sample waiting times from a geometric distribution to generate general RHGs and in (Section 3.6) we generate threshold RHGs in near-linear time using a band data structure. Can both be combined?

The main idea of our next algorithm is to assemble the neighborhood of a vertex by sweeping through each band with increasing angular distance. Instead of aborting the sweep after the distance grows larger than R, as done for threshold RHGs, we sample geometric waiting times between neighbor candidates.

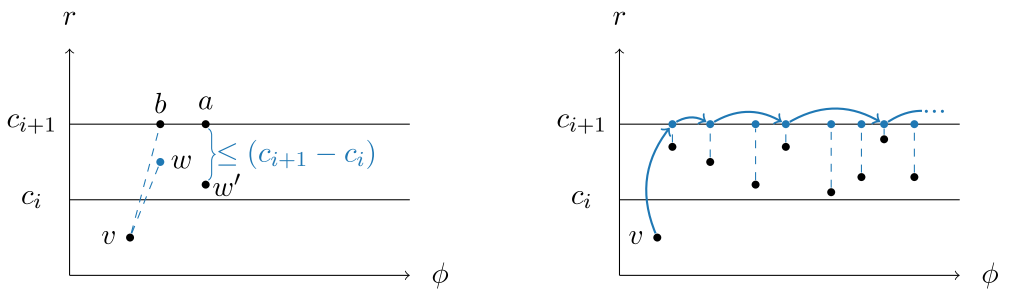
Algorithm 9: Generating general hyperbolic random graphs in near-linear time

```
Input: vertex set $V$, annuli $b_i \in B$ in which points are sorted by their angular
       coordinates
Output: Edge set $E$
1  $E \leftarrow \emptyset$;
2  for vertex $v \in V$ do in parallel
3      for band $b_i \in B$, where $c_{i+1} > r_v$ do
4          for dir $\in$ {clockwise, counterclockwise} do
               /* Sweeping in direction dir                                  */
5              $w \leftarrow$ vertex $\in b_i$ with closest angular coordinate $\phi_w$ when sweeping from
                 $\phi_v$ counterclockwise [clockwise];
6              totalSwept $\leftarrow 0$;
7              while $\Delta\phi(v,w) < 180°$ and totalSwept $\leq |b_i|$ do
8                  $\bar{b} \leftarrow f(\mathrm{dist}_{\mathcal{H}}(v, (\phi_w, c_{i+1})) - (c_{i+1} - c_i))$;
9                  $\delta \leftarrow \ln(1 - rand) / \ln(1 - \bar{b})$;
10                 $w' \leftarrow$ jump $\delta + 1$ vertices in $b_i$ counterclockwise [clockwise];
11                 $prob \leftarrow f(\mathrm{dist}_{\mathcal{H}}(v, w'))/\bar{b}$;
12                 if $r_v < r_w$ or ($r_v = r_w$ and $v < w$) then
13                     add $(v, w')$ to $E$ with probability $prob$;
14                 totalSwept += $\delta + 1$;
15                 $w \leftarrow w'$;
16 return $E$
```

Algorithm 9 shows the edge sampling pseudocode. As the sampling of vertex positions and construction of band data structure are unchanged from Algorithm 8, we omit them here and take already constructed bands as input. The sampling of neighborhoods happens independently for each vertex and can thus be parallelized (Line 2). To assemble the neighbors a vertex v has in band b_i, we sweep the half that is left of v counterclockwise and the half right of v clockwise, each so that the angular distance to v only increases during a sweep. The first vertex of each sweep is the one with the highest [lowest] coordinate that is at most [higher than] ϕ_v (Line 5). During a sweep (Line 7), the number of vertices that are skipped between neighbor candidates is sampled from a geometric distribution. To sample the gap between candidates w and w' (which is unknown), we need an upper bound ($\bar{b}$) for the probability $f(\mathrm{dist}_{\mathcal{H}}(v, w'))$ that vertex w' is a neighbor of v and for this we need a lower bound for the distance of v to w' (Line 8).

Unfortunately, the distance $\mathrm{dist}_{\mathcal{H}}(v, w)$ is *not* a lower bound for the distance $\mathrm{dist}_{\mathcal{H}}(v, w')$, since even though the angular distance $\Delta\phi(v, w')$ is at least $\Delta\phi(v, w)$, the radial coordinate $r_{w'}$ can be any between c_{i+1} and c_i, and thus the actual distance might be smaller.

Instead, we use auxiliary points $a := (\phi_{w'}, c_{i+1})$, $b := (\phi_w, c_{i+1})$ (see Figure 3.8a) and the



(a) Deriving a lower bound for the distance $\text{dist}_{\mathcal{H}}(v, w')$. The distance of v to b (which is known) is a lower bound for the distance of v to a (which is unknown). Due to the triangle inequality, the unknown distance $\text{dist}_{\mathcal{H}}(v, w)$ is at least $\text{dist}_{\mathcal{H}}(v, a) - (c_{i+1} - c_i)$.

(b) Selecting candidates within a band, done in Line 7 of Algorithm 9.

Figure 3.8: Visualization of the candidate sampling process.

triangle inequality:

$$\begin{aligned}
\text{dist}_{\mathcal{H}}(v, w') + \text{dist}_{\mathcal{H}}(w', a) &\geq \text{dist}_{\mathcal{H}}(v, a) && (3.38)\\
\Rightarrow \text{dist}_{\mathcal{H}}(v, w') &\geq \text{dist}_{\mathcal{H}}(v, a) - \text{dist}_{\mathcal{H}}(w', a) && (3.39)\\
\Rightarrow \text{dist}_{\mathcal{H}}(v, w') &\geq \text{dist}_{\mathcal{H}}(v, a) - (c_{i+1} - c_i) && (3.40)\\
\Rightarrow \text{dist}_{\mathcal{H}}(v, w') &\geq \text{dist}_{\mathcal{H}}(v, b) - (c_{i+1} - c_i). && (3.41)
\end{aligned}$$

Thus, $\text{lb}(v, w, c_{i+1}, c_i) := \text{dist}_{\mathcal{H}}(v, (\phi_w, c_{i+1})) - (c_{i+1} - c_i)$ is a lower bound for the unknown distance $\text{dist}_{\mathcal{H}}(v, w')$. It is used to compute an upper bound for the probability that any given of the following vertices is a neighbor, which is the basis for the jumping width calculation in Line 9. Lines (10 – 15) are similar to Algorithm 8 in that each candidate w' is confirmed with probability $f(\text{dist}_{\mathcal{H}}(v, w'))/\overline{b}$ and added to the edge set. Line 12 ensures that an edge is not added twice and Line 14 is to prevent a large jump circling through the sweep area multiple times. Afterwards, the procedure is repeated for the other direction.

3.7.1 Time Complexity

For each vertex v, Algorithm 9 visits every band and every neighbor of v, thus has at least a complexity of $\Omega(n \log n + m)$. The full time complexity depends on the number of candidates that *do not* yield an edge, which we name *excess candidates*. Their number is of course probabilistic, but their expected number depends on the tightness of the upper bound used for the jumping widths (Line 8).

The distance bound in (3.41) underestimates the true distance in two ways: First, the offset $(c_{i+1} - c_i)$ is almost always larger than the true distance $\text{dist}_{\mathcal{H}}(w', a)$. Second, the sweep covers increasing angular distance, and $\Delta\phi(v, w')$ is almost always larger than $\Delta\phi(v, w)$.

We address these separately in two lemmata. Lemma 14 considers the offset in the radial coordinates (the vertical dashed lines in Figure 3.8b) and Lemma 16 states the time complexity of Algorithm 9 including the angular jumps (horizontal arrows in Figure 3.8b).

Lemma 14. *Let v, w, w' and a be points as defined in (Eq.3.41), let $\Delta c := (c_{i+1} - c_i)$ be the width of band i and let $f(x) = \frac{1}{e^{(1/T)(x-R)/2}+1}$ be the edge probability function of the extended hyperbolic random graph model with $T > 0$. Then, the upper bound $f(\mathrm{dist}_\mathcal{H}(v, a) - \Delta c)$ overestimates the probability $f(\mathrm{dist}_\mathcal{H}(v, w'))$ of an edge between vertices v and w' by at most $e^{\Delta c/T}$.*

Proof. We use the definition of the upper bound and again the triangle inequality::

$$\begin{aligned} \mathrm{dist}_\mathcal{H}(v, w') &\leq \mathrm{dist}_\mathcal{H}(v, a) + \mathrm{dist}_\mathcal{H}(a, w') \\ \Rightarrow \mathrm{dist}_\mathcal{H}(v, w') &\leq \mathrm{dist}_\mathcal{H}(v, a) + \Delta c \\ \Rightarrow \mathrm{dist}_\mathcal{H}(v, w') - (\mathrm{dist}_\mathcal{H}(v, a) - \Delta c) &\leq \mathrm{dist}_\mathcal{H}(v, a) + \Delta c - (\mathrm{dist}_\mathcal{H}(v, a) - \Delta c) \\ \iff \mathrm{dist}_\mathcal{H}(v, w') - (\mathrm{dist}_\mathcal{H}(v, a) - \Delta c) &\leq 2\Delta c. \end{aligned}$$

Due to the exponential in the edge probability function, a constant distance offset leads to a multiplicative factor:

$$\begin{aligned} & f(\mathrm{dist}_\mathcal{H}(v, w') - 2\Delta c) \\ &= \left(\exp\left((1/T)(\mathrm{dist}_\mathcal{H}(v, w') - 2\Delta c - R)/2\right) + 1\right)^{-1} \\ &= \left(\exp(-2\Delta c/(2T)) \cdot \exp\left((1/T)(\mathrm{dist}_\mathcal{H}(v, w') - R)/2\right) + 1\right)^{-1} \\ &\leq \left(\exp(-\Delta c/T) \cdot \exp\left((1/T)(\mathrm{dist}_\mathcal{H}(v, w') - R)/2\right) + \exp(-\Delta c/T)\right)^{-1} \\ &= e^{\Delta c/T} \cdot \left(\exp\left((1/T)(\mathrm{dist}_\mathcal{H}(v, w') - R)/2\right) + 1\right)^{-1} \\ &= e^{\Delta c/T} \cdot f(\mathrm{dist}_\mathcal{H}(v, w')) \end{aligned}$$

□

Lemma 15. *Let v be a point, b_i a band delimited by radii c_i and c_{i+1}, let $\deg(v, b_i)$ be the number of neighbors v has in b_i and let T be fixed. Then, the while loop in Line 7 of Algorithm 4, sampling the neighborhood of v in b_i, takes time $\mathcal{O}(\deg(v, b_i) + \log |b_i|)$ in expectation.*

Proof. For the purpose of the proof, divide the points in b_i into infinitely many subsets P_j, with $P_j = \{w \in b_i | 2^{-(j+1)} < f(\mathrm{dist}_\mathcal{H}(v, (\phi_w, c_{i+1}))) \leq 2^{-(j)}\}$. All but a finite number of these will be empty.

Let w be the point of the current loop iteration and w' the next candidate after the jump in Line 10. One of two cases happens:

- w, w' are in the same subset P_j. Then, the ratio between $f(\mathrm{dist}_\mathcal{H}(v, w'))$ and $f(\mathrm{dist}_\mathcal{H}(v, w)$ is at least $\frac{1}{2}$ and by Lemma 14 the probability that the candidate w' is confirmed as a neighbor of v is at least $\frac{1}{2} \cdot e^{-\Delta c/T}$.
- w' is in a subset P_l with $l > j$.

Since the expected δ for a vertex $w \in P_j$ is at least $2^j / e^{\Delta c/T}$ (Lemma 14), the second case happens only $\mathcal{O}(\log |b_i|)$ often in expectation before all points in b_i are covered. The first case happens $\mathcal{O}(\deg(v, b_i))$ often in expectation, since each candidate is a neighbor with a probability bounded from below by a constant.

The operations in a single loop iteration, sampling the δ and computing the bounds, take constant time.

Thus, the expected time complexity of $\mathcal{O}(\deg(v, b_i) + \log |b_i|)$ follows. $\square$

Lemma 16. *Sampling the neighborhood of a point v has a complexity of $\mathcal{O}(\deg(v) + \log^2 n)$.*

Proof. Due to Lemma 15, the complexity of sampling within a single band is $\mathcal{O}(\deg(v, b_i) + \log |b_i|)$. The number of bands is $\lceil \log n \rceil$ and each of them has at most n points.

$$\sum_{b_i \in B} \deg(v, b_i) + \log |b_i| = \deg(v) + \sum_{b_i \in B} \log |b_i| \leq \deg(v) + \log^2 n$$

$\square$

Theorem 4. *Generating a graph with n points and m edges using Algorithm 9 takes $\mathcal{O}(m + n \log^2 n)$ time in expectation.*

Proof. Use Lemma 16:

$$\sum_{v \in V} \deg(v) + \log^2 n = m + n \log^2 n$$

$\square$

Note that we treated the temperature T and the band width c as constants. If treated as variables, they form a multiplicative factor of $e^{\Delta c/T}$ in the time complexity. However, for small T the rapidly increasing jumping widths limit the number of excess candidates in practice and a tighter bound is likely possible.

3.8 Experimental Results

Our algorithms were the first to have a lower asymptotic time complexity than the naive sampling method. Later, Bringmann et al. [31] presented a generation algorithm for geometric inhomogeneous random graphs (GIRG), a generalization of RHGs. To determine their performance in practice, we compare the implementations for static and dynamic graphs, both in the threshold and general model.

Implementation.

In the following comparison, let QuadGen denote the implementations of our quadtree-based algorithms (Section 3.4 and BandGen the implementations of our band-based algorithms (Section 3.6). All our implementations use the NetworKit toolkit [154], are written in C++ 11 and use OpenMP for shared-memory parallelism. QuadGen was released in NetworKit version 4.0.1 for threshold random hyperbolic graphs and in version 4.1.1 for general random hyperbolic graphs. To evaluate BandGen, we use the current version of NetworKit, which is 4.6.

Other authors also published graph generators during my PhD studies, both for general and threshold random hyperbolic graphs. Aldecoa et al. [5] implement the straightforward approach with quadratic time complexity, calculating distances and sampling edges for all $\Theta(n^2)$ vertex pairs. We denote it with Aldecoa.

Bringmann et al. [31] propose *Geometric Inhomogeneous Random Graphs* as a generalization of RHGs and describe a generation algorithm with expected linear time complexity. Bläsius et al. provide an implementation[9] of this algorithm as part of their hyperbolic embedding work [22], which won the Track B best paper award of ESA 2016. We denote it with Girg. When comparing times for dynamic graphs, we based our changes to their data structure on commit f83b46111c69819b7447fbd29fe8ed9bdb1fba3f.

All implementations are compiled with GCC 7.3. Scripts and a docker image to recreate the experiments are available at DOI 10.5445/IR/1000095237 on the KITOpenData repository.

Experimental Setup

The experiments were run on a workstation with 16 physical cores and 256GB of memory, using OpenSuse 15. Except for the parallel scaling studies, all experiments were run on a single core of a Xeon E5-2680, clocked at 2.70GHz. Other cores were kept idle to avoid influencing memory access times.

3.8.1 Time

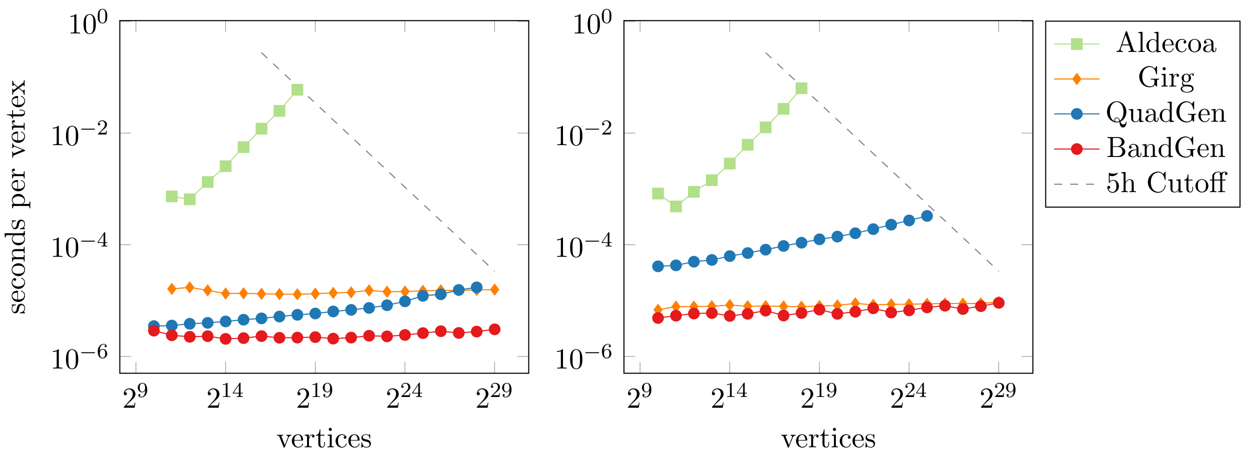
Static Generation of Threshold RHGs

Figure 3.9a shows the generation times per vertex for graphs of different sizes. As expected, in straightforward distance probing as done by Aldecoa the time per vertex scales linearly with graph size. Due to this quadratic scaling of generation times, graphs with 2^{19} vertices and more did not finish in the alloted time of five hours. The times per vertex are approximately constant for Girg, implying a near-linear scaling behavior. For QuadGen, the time per vertex does rise with the graph size, but less than expected from the worst-case complexity. At 2^{29} vertices, however, issues in floating point precision occur due to the projection in the Poincaré disc. The number of edges is less than desired and the running time thus not representative, which is why we omit the final data point. The times per vertex for BandGen are almost one magnitude faster than Girg.

Parallel Scalability

Figure 3.10 shows the scaling behavior of QuadGen and BandGen for 1 to 16 threads on graphs of constant size. It appears that slower sampling methods have better parallel scaling: QuadGen scales better than BandGen, both scale better on non-threshold graphs,

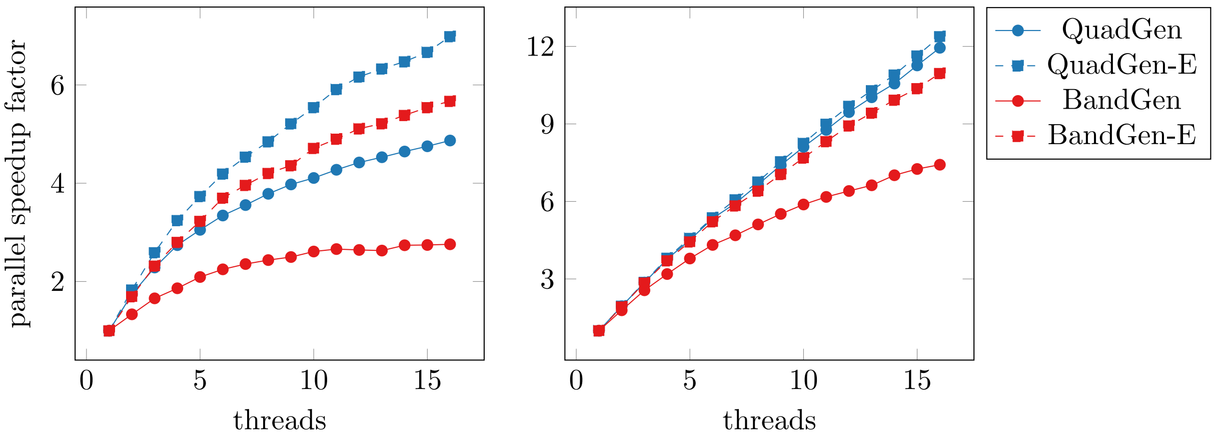
[9] `https://bitbucket.org/HaiZhung/hyperbolic-embedder`



(a) Generation times of threshold graphs, T set to 0.

(b) Generation times of graphs with non-zero temperature, T set to 0.5.

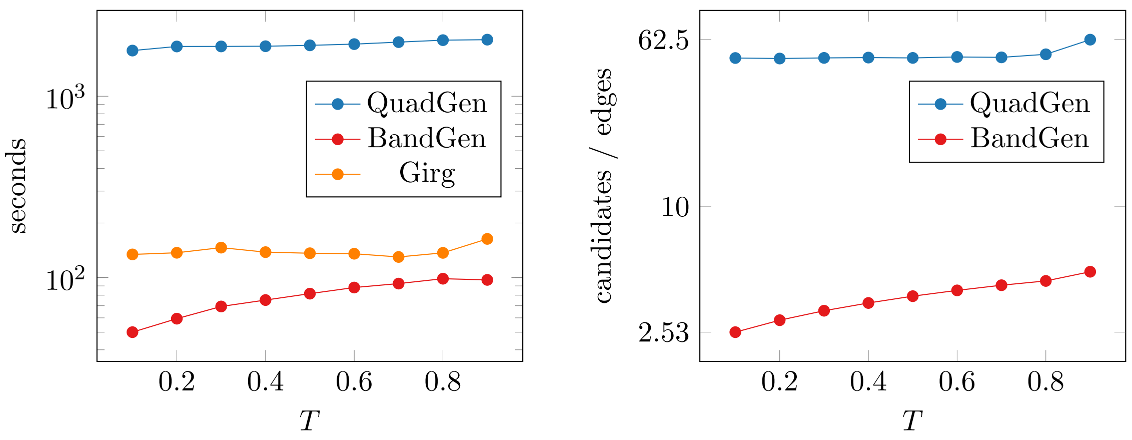
Figure 3.9: Generation times per vertex of different graph sizes. Average degree is held constant at 8 and dispersion parameter α is 1. Since the generators Aldeoca and Girg are sequential, we execute all generators sequentially, even though QuadGen and BandGen can use shared-memory parallelism. We aborted runs taking longer than 5 hours.



(a) Threshold graphs, T set to 0.

(b) General graphs, T set to 0.5

Figure 3.10: Strong scaling of QuadGen and BandGen for 10^7 vertices with average degree 8 and $\alpha = 1$. Values are averaged over 10 runs. QuadGen-E and BandGen-E denote variants that measure only the edge sampling step.



(a) Times to generate non-threshold graphs using Girg, QuadGen and BandGen.

(b) Ratio of candidates to edges in QuadGen and BandGen. In contrast to the theoretical bounds, the number of candidates in BandGen does not increase for $T \to 0$.

Figure 3.11: Running times and candidate ratios for graphs of 2^{23} vertices, average degree 8, $\gamma = 3$ and different temperatures.

which take longer to generate. This implies that the edge sampling methods themselves scale comparatively well, but other steps in the graph generation, perhaps the preparation of coordinates, allocation of memory or the final assembly of the graph data structure, do not. This is supported by measurements omitting the final graph assembly, noted as dashed lines: These achieve a higher speedup, ranging from 5.73 for BandGen on threshold graphs to 12.38 for QuadGen on non-threshold graphs.

Static Generation of General RHGs

Figure 3.9b shows running times of Aldecoa, Girg and QuadGen for the generation of general random hyperbolic graphs, with T set to 0.5. Again, the times per vertex for Girg are almost constant with increasing graph size. The running times of Aldecoa are similar to the generation of threshold graphs. This is unsurprising, given that the time for pairwise distance probing does not depend on the edge probabilities. The scaling behavior of QuadGen is more in line with the expected theoretical complexity of $\mathcal{O}(\sqrt{n}\log n)$ per vertex than in the threshold case. It is also more than an order of magnitude slower than for threshold graphs, since many more vertices need to be examined. BandGen, while also slower than for threshold random hyperbolic graphs, is still about 20% faster than Girg for graphs of up to a billion vertices.

This also holds for different temperatures, as seen in Figure 3.11. Somewhat surprisingly, the time does not increase with smaller temperatures, as could be expected by the factor of $\exp(\Delta c/T)$ in the theoretical time complexity. It seems that the probabilities fall sufficiently quickly with increasing hyperbolic distances and fewer probabilistic jumps are required.

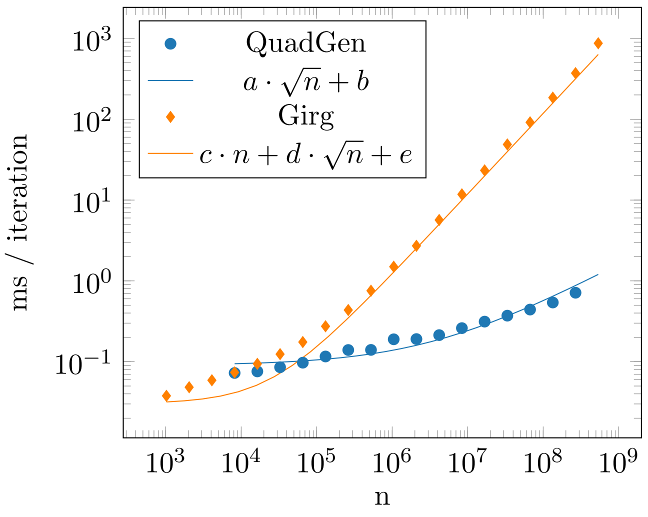


Figure 3.12: Running time of dynamic vertex movements, values are averaged over 10000 movements. The quadtree operations are up to two orders of magnitude faster and scale better with increasing graph size. Trend lines are fitted with $a = 0.48$, $b = 900$, $c = 0.00118$, $d = 0.017$ and $e = 30$.

Dynamic Generation of General RHGs

Figure 3.12 shows the experimental time measurements for dynamic updates on sparse graphs of varying sizes. For easier empirical comparisons, we conduct our benchmark on the previous dynamic model proposed by Papadopoulos et al. [127]. In each iteration of this dynamic benchmark, one point is deleted from the data structure, moved to a random location consistent with the probability distribution and reinserted into the data structure. The graph is then updated with the new position. For each graph, we execute 10 000 point movements and recreate the graph after each.

The temperature parameter T is set to 0.1, the dispersion parameter α to 0.75 and the radius R to $2 \cdot \log n - 1$, leading to an average degree of ≈ 9.3.

Since changes in a sorted sequence may require linear time to process, the data structure used by BandGen is not suited for dynamic updates and we omit it here. QuadGen is faster than Girg for graphs of at least 10^5 vertices; the improvement reaches two orders of magnitude for graphs with $2 \cdot 10^8$ vertices; a query on hundreds of milllions of vertices returning about 10 neighbors runs in the order of milliseconds.

The fastest way to obtain a static graph together with a sequence of dynamic updates would be to generate the static graph first with BandGen, then use dynamic updates with QuadGen. Moving data into a new data structure takes at least linear time, Figure 3.13 shows the number of vertex movements needed until the overhead of this preprocessing step is amortized by the faster queries. For graphs with 10^5 to 10^8 vertices, this combination of BandGen and QuadGen is faster than Girg if performing more than roughly 10^4 iterations, a value that grows only slowly with increasing graph size.

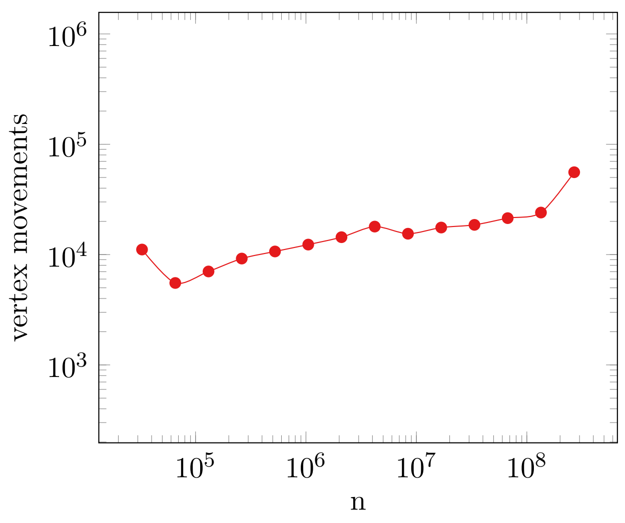
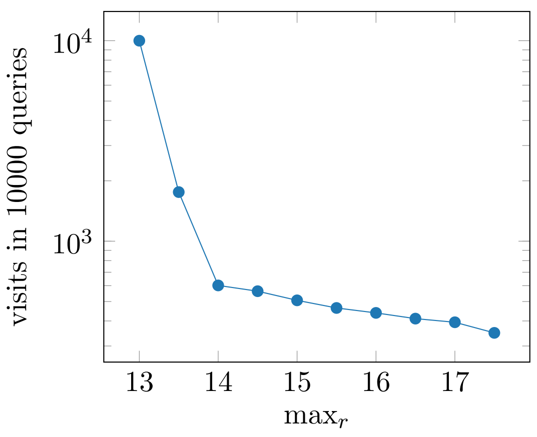


Figure 3.13: Number of vertex movements needed to amortize overhead of quadtree construction. Values are averaged over 10000 iterations.

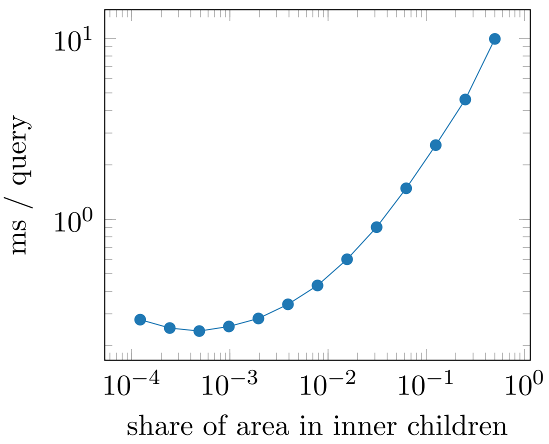
Optimizations

The expected degree of a vertex in a random hyperbolic graph depends on its radials coordinate, a smaller radius leading to a higher degree. Since the edge probabilities are symmetrical, vertices with small radius are more likely to be in the result set of a query. Due to this effect, the central cells in the quadtree are examined much more often than cells on the periphery, see Figure 3.14a.

Probably due to this effect, deliberately imbalancing the polar quadtree and allocating less probability mass to the inner children improves the running time by several orders of



(a) Probability that a given cell is examined in an update step, depending on its maximum radial coordinate $\max_r$. Cells in the center of the polar disk are visited almost certainly, while cells in outer regions are visited rarely. Measurements are made on a random hyperbolic graph with 2^{13} vertices.



(b) Influence of balance parameter on running time. Measurements are for a graph with 2^{23} vertices and averaged over 10000 queries. Deliberately imbalancing the quadtree improves running times by over one order of magnitude.

Figure 3.14: Imbalance in quadtree accesses and effect of imbalancing quadtrees.

magnitude. The splitting radius $\text{split}_{\mathcal{H}}$ which divides the outer from the inner children of T, originally given in Eq. 3.6, is then governed by a balance parameter b. For random hyperbolic graphs, this yields:

$$\text{mid}_{r\mathcal{H}} = \text{acosh}((1-b)\cdot\cosh(\alpha\cdot\max_{r_{\mathcal{H}}}) + b\cdot\cosh(\alpha\cdot\min_{r_{\mathcal{H}}}))/\alpha. \tag{3.42}$$

The original behavior of Eq. 3.6 is equivalent to setting b to 0.5. Choosing instead $b = 0.001$, which yields an allocation of 0.1% of the area to the inner two children and 99.9% to the outer children, decreases running time by more than an order of magnitude compared to a balanced tree (Figure 3.14b). The running times in the previous figures include the effect of these optimizations.

3.8.2 Network Properties

To ensure that our generation algorithms sample from the correct distribution, we compare the properties of the generated graphs with those produced by Aldecoa. For 28 combinations of the average degree k and the power law exponent γ of the degree distribution, we used both generators to sample 214 graphs each, resulting in roughly 12 000 samples in total. Figures 3.15 and 3.16 show average properties for graphs generated by Aldeoca and the relative differences to graphs from our generators. These differences of averages are on the order of 1-2%, except for the degree assortativity, which trends towards zero for dense graphs (≈ 0.00037 for $k = 256$ and $\gamma = 4$) and thus causes even small absolute fluctuations to have a large relative difference. In all cases, *including* the degree assortativity, the difference *between* the generators is less than the differences *within* the set of graphs produced by one generator, on average by an order of magnitude. More formally, the average difference between the means of the distribution of properties of the generated graphs is one order of magnitude less than the standard deviation σ of measurements for the graphs produced by the same generator for any fixed set of parameters.

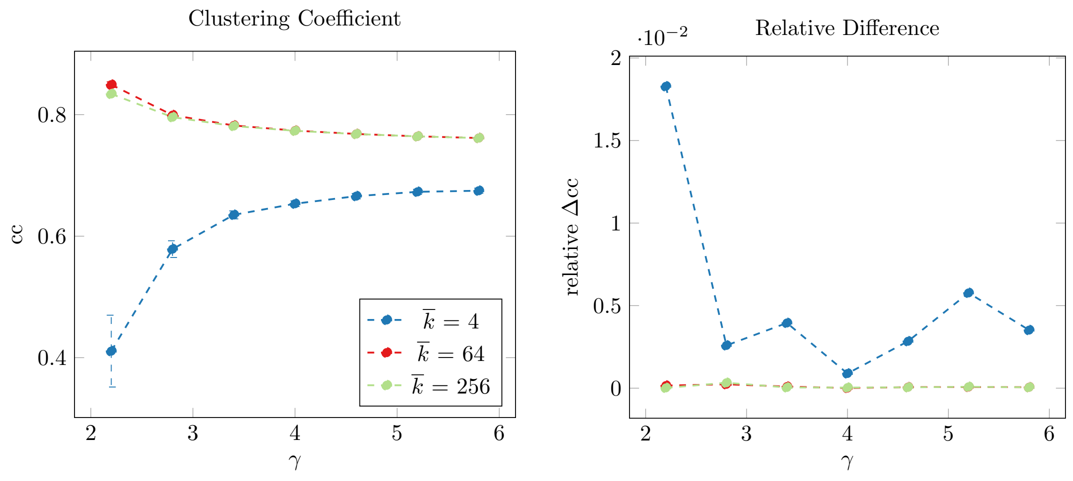


Figure 3.15: Comparison of clustering coefficients. Left are average values for the implementation of [5] (left), error bars show the standard deviation. Right are the relative differences of our implementation. Values are averaged over 214 runs.

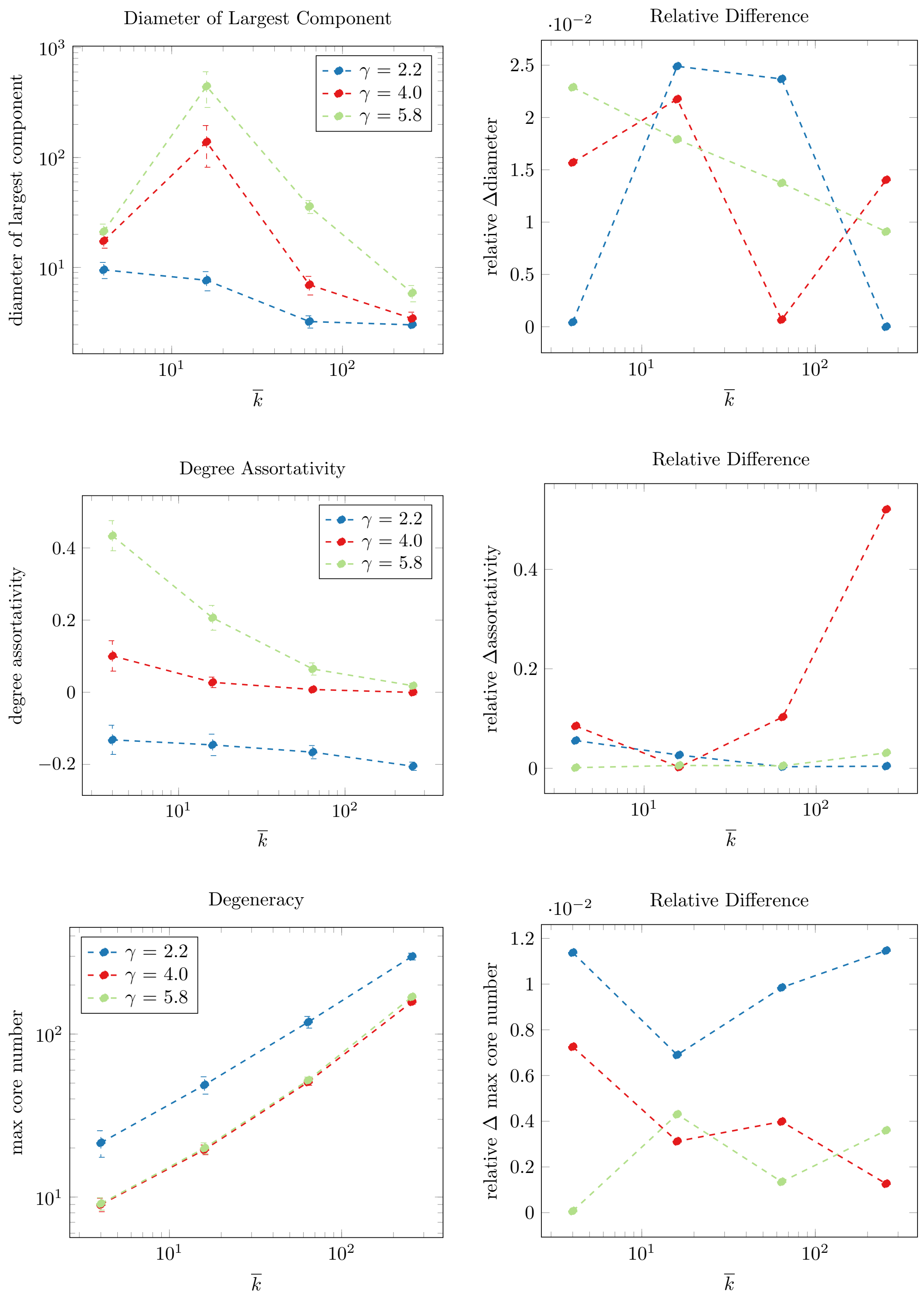


Figure 3.16: Comparison of diameter, degree assortativity and degeneracy. Degree assortativity describes whether vertices have neighbors of similar degree. A value near 1 signifies subgraphs with equal degree, a value of -1 star-like structures. Degeneracy refers to the largest core number in a k-core decomposition, a generalization of components. Left are average values for the implementation of [5] (left), error bars show the standard deviation. Right are the relative differences to our implementation. Values are averaged over 214 runs.

To investigate this rigorously, we define two hypotheses H_0 and H_1 and compute a Bayes factor [78] to estimate their relative likelihood. As the properties have different variances, we estimate the standard deviation σ separately for each property. Let H_0 be the hypothesis that for all properties, the true difference between the average outputs of the generators is at most 10% of the estimated standard deviation σ. Let H_1 be the hypothesis that this difference is above 10%. We use MCMC sampling to compute the Bayes factor comparing H_0 and H_1.It shows that hypothesis H_0 is overwhelmingly more likely, meaning that if there is a difference between the properties of the generated graphs, it is at least one order of magnitude smaller than the average difference between graphs from one generator.

3.8.3 Probabilistic Spreading

When both contact graph and travel patterns of a susceptible population are not known in detail, the resulting spreading behavior of an infectious disease is probabilistic. Contagious diseases usually spread to people in the vicinity of infected persons, but an infectious person occasionally bridges larger distances by travel and spreads the disease this way. We model this effect with our probabilistic neighborhood function f, giving a higher probability for small distances and a lower but non-zero probability for larger distances. Note that this scenario is meant as an example of the probabilistic spreading simulations possible with our algorithm and not as highly realistic from an epidemiological point of view.

In the simulation, the population is given as a set P of points in the Euclidean plane. In the initial step, exactly one point (= person) from P is marked as infected. Then, in each round, a PNQ is performed for each infected person q. All points in $N(q, f)$ become infected in the next round. We use an SIR model [69], i. e. previously infected persons recover with a certain probability in each round and stay infectious otherwise. In our simulation, persons recover with a rate of 0.8 and are then immune.

Results

We experimented on three data sets taken from NASA population density raster data[10] for Germany, France and the USA. They consist of rectangles with small square cells (geographic areas) where for each cell the population from the year 2000 is given. To obtain a set of points, we randomly distribute points in each cell to fit 1/20th of the population density. Figure 3.17 shows an example with roughly 4 million points on the map of Germany. The data sets of France and USA have roughly 3 and 14 million points, respectively.

The number of required queries naturally depends heavily on the simulated disease. For our parameters, a number of 5000 queries is typically reached within the first dozen steps. To evaluate the algorithmic speedup, Table 3.1 compares running times for 5000 pairwise distance probing (PDP) queries against 5000 fast PNQs on the three country datasets. To obtain a similar total number of infections, we use a slightly different probabilistic neighborhood function for each country and divide by the size: $f(x) := (1/x) \cdot e^7/n$, resulting in a slower initial progression for the United States. Our algorithm achieves a speedup factor of at least two orders of magnitude, even including the quadtree construction time.

[10]`http://sedac.ciesin.columbia.edu/data/set/gpw-v3-population-density/data-download`

Country	5000 PDP queries	Construction QT	5000 QT queries
France	1007 seconds	1.6 seconds	1.2 seconds
Germany	1395 seconds	2.8 seconds	1.3 seconds
USA	4804 seconds	8.7 seconds	0.7 seconds

Table 3.1: Running time results for polar Euclidean quadtrees on population data. The query points were selected uniformly at random from P, the probabilistic neighborhood function is $f(x) := (1/x) \cdot e^7/n$.

3.9 Concluding Remarks

Based on the popular generative model of random hyperbolic graphs, we defined a dynamic model for gradual vertex movement.

We developed four generation algorithms for random hyperbolic graphs, three of which were published and were the fastest at their time of publication, often by several orders of magnitude. This was achieved through geometric data structures adapted for hyperbolic geometry. For dynamic updates to random hyperbolic graphs, our algorithm remains the fastest one available, both in theory and practice. Others have built on this work, especially Penschuck [132] and Lamm [96], leading to even faster generators for external and distributed memory. This culminated in a joint work of Funke et al. [58] (in which I was a co-author), which won the IPDPS 2018 best paper award. An extended version was accepted to the IPDPS 2018 Special Issue of the *Journal of Parallel and Distributed Computing*.

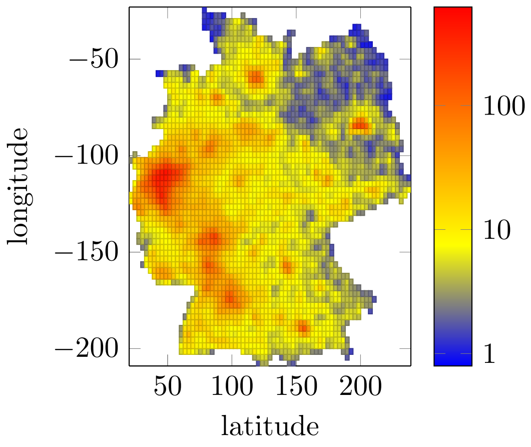


Figure 3.17: Heat map of the 23rd time step of a simulated carnival fever progression through Germany.

4. Graph Partitioning

We present two graph partitioners, designed for very different applications.

After considering a small part of the rather extensive previous work done in graph partitioning (Section 4.2), we discuss the issue of partitioning protein graphs for an application in quantum chemistry (Section 4.3). Compared to general graph partitioning, it has several additional constraints due to properties of the chemical simulations. We adapt several existing approaches to fulfill these additional constraints. For a scenario with a restricted solution space, we provide an optimal dynamic programming algorithm. This work was done jointly with Henning Meyerhenke, Mario Wolter and Christoph Jacob and was first presented at SEA'16 [166].

The second graph partitioning algorithm (Section 4.4) is designed for a geophysics application. The WAVE project[11] creates an open-source toolbox for full waveform inversion, to which we contribute the load balancing. Since geometric information is available, we adapt the well-known k-means algorithm to yield balanced blocks suitable for partitioning. Using MPI for distributed memory parallelism, it scales to tens of thousand of processes and partitions graphs with billions of vertices in seconds. The algorithm and implementation for balanced k-means were presented at ICPP 2018 [165] as joint work with Henning Meyerhenke and Charilaos Tzovas.

4.1 Problem Definition

Given a graph, the *Graph Partitioning Problem* (GPP) asks how to best cut it into parts. More formally, let $G = (V, E)$ be a graph, $k \in \mathbb{N}_0$ the desired number of parts, $\epsilon \in \mathbb{R}^+$ a balance constraint and $\omega : E \to \mathbb{R}^+$ an edge weight function. Then, the problem consists of finding a k-partition Π of V so that the blocks $(V_1, \ldots, V_k)$ are pairwise disjoint, each has at least 1 and at most $\lceil (1+\epsilon)\frac{|V|}{k} \rceil$ vertices and some objective function is optimized.[12]

[11] `http://wave-toolbox.org/`

[12] In the mathematical use of "partition", it is implied that the blocks are non-empty and pairwise disjoint. We specified it for completeness sake.

The most common objective function is the *edge cut*: The sum of the weights of all edges whose incident vertices are in different blocks.

$$\text{edgeCut}(\Pi) := \sum_{e=\{u,v\}\in E: u\in V_i, v\in V_j, i\neq j} \omega(e) \tag{4.1}$$

Apart from the edge cut, another suitable objective function when partitioning for parallel numerical simulations is the *communication volume*, as it measures communication costs in some scenarios more accurately [67]. For each vertex, it counts the number of blocks the vertex is connected to:

$$\text{commVol}(\Pi) := \sum_{V_i\in\Pi} \sum_{v\in V_i} |\{V_j \in \Pi : \exists u \in V_j \wedge \{u,v\} \in E\}| \tag{4.2}$$

To evaluate the shape of a block V_i, its diameter is of interest. In contrast to (2.2), this distance dist_{V_i} only considers vertices and paths in block V_i:

$$\text{diam}(V_i) := \max_{u,v\in V_i\times V_i} \text{dist}_{V_i}(u,v). \tag{4.3}$$

The graph partitioning problem is $\mathcal{NP}$-complete and hard to approximate [32] for most objective functions, thus heuristics are used in practice.

4.1.1 Applications

Graph partitioning is relevant for many applications, both when the interest is in the graph itself or in the underlying phenomenon. In simulations of spatial phenomena, it is common to discretize the simulation domain into a geometric graph called *mesh*. In the common use case of modeling with partial differential equations (PDEs) [117], this discretization ultimately leads to linear systems or explicit time-stepping methods. The resulting matrices are typically very large and sparse, requiring parallelization for efficient solutions. When the matrix rows are modeled as vertices and the data dependencies between them as edges, a partition of this graph yields a parallelization scheme.

Electricity networks sometimes fail, which is unpleasant for those who need them [139]. A way to reduce the effects of a failure is to partition the power grid into a set of self-sufficient islands, which can continue service in the face of a wider disruption. Load imbalances between these islands should be minimized, while fulfilling other constraints [100].

In parallel graph analysis tasks, partitioning the input graph is often a first step to a parallelization [1]. In route planning, partitioning the street graph into natural parts can give high speedups in preprocessing [19].

Many other applications exist.[13] Note that graph partitioning is different from *Graph Clustering*, which also requires a partition of the vertex set, but usually does not specify the number of parts in advance and does not enforce a balance constraint.

[13]Of course, since GPP is NP-complete, all problems in NP can be phrased as graph partitioning problems. Needless to say, addressing the whole of NP is beyond the scope of this work.

4.1.2 Extensions and Additional Constraints

The graph partitioning problem can be extended with additional data and constraints. While each application may bring its own unique set, the following are the most common.

Vertex Weights

Different computational loads can be represented by a vertex weight function w. The balance constraint is then extended to the sum of vertex weights per block. For general weights, deciding whether a partition with a certain balance constraint is possible is equivalent to the bin packing problem, which is $\mathcal{NP}$-complete [60]. To ensure a partition is always possible, we relax the constraint by adding the difference between the largest and smallest weight:

$$\sum_{v \in V_i} w(v) \overset{!}{\leq} \left\lceil (1+\epsilon)\frac{\sum_v w(v)}{k} + \max_v w(v) - \min_v w(v) \right\rceil .$$

Geometry

Many meshes from numerical simulations come with geometric information, which give a global view useful for partitioning heuristics.

Multiple Weights

Some problems are best modeled with multiple vertex weights and multiple balance constraints [82, 81]. One example are numerical simulations using different solvers in which computational cost and memory requirements scale differently. Each processing element should get a subproblem with a similar amount of computational load, but also not exceeding the available local memory.

Unequal Block Sizes

Due to the increasing use of GPUs and accelerators for scientific computing [115], parallel computation on heterogeneous infrastructure becomes common. When partitioning for parallel computations in such a scenario, the tasks assigned to these computing elements should reflect the different computational power to avoid stragglers or inefficient use of memory. Thus, the balance constraint is replaced by a list of target block sizes, each modeling the capabilities of one computing element.

4.2 Previous Work

4.2.1 Exact Methods

As graph partitioning is $\mathcal{NP}$-complete, exact methods are possible, but usually take very long. Numerous authors approached this problem using branch-and-bound or semidefinite programming. For example, Ferreira et al. [54] develop separation heuristics based on graph-based inequalities, they use these to partition graphs of a few hundred vertices

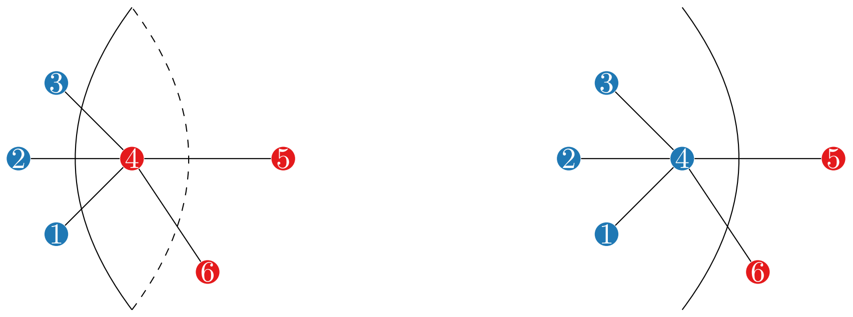


Figure 4.1: Moving vertex 4 from the red block to the blue block has a gain of 1. Before the move, three edges were cut. After the move, only two edges are cut.

optimally. Delling et al. develop a combinatorial branch-and-bound algorithm [42] for minimum graph bisection, the case where $k = 2$. They achieve provably optimal results on graphs with small cuts in the range of millions of vertices. In current graph partitioning, exact methods and approximations are of mostly theoretical interest [33].

4.2.2 Local Search

Local search methods, also called *local refinement* methods, improve an existing partition by changing the assignments of individual vertices. Most current methods can be traced back to the foundational pair-swapping heuristic by Kernighan and Lin [84], which swaps pairs of adjacent vertices between different blocks in the order of induced improvement in the target function. This improvement is called the *gain* of a move. When optimizing the edge cut, the gain is the difference between the edge cuts before and after the move. Fiduccia and Mattheyses (FM) [55] adapt this method to unpaired vertex movements, enabling larger improvements. Figure 4.1 shows an example of a move with a gain of 1. After every vertex has been moved once, even those with negative gain, the solution with the best quality is chosen. Such a phase is repeated several times, each running in time $\mathcal{O}(m)$.

The original Fiduccia-Mattheyses targets *bisections*, i. e. partitions for $k = 2$. Karypis and Kumar [83], among others, extend it to general k-way refinement.

Parallelization of local refinement algorithms is challenging, as the gain of moving a vertex depends on the block assignments of its neighbors. In addition, if the same block is modified by several processes concurrently, changes to its size have to be communicated to avoid violating the balance constraints.

Holtgrewe et al. [73] approach these issues by allocating one process per block and computing an edge coloring of the block graph. They then perform local refinement in distinct rounds, one per color. In each round, processes sharing an edge with that round's color communicate to perform FM on the common border between their blocks. This limits communication to pairs of processes and avoids race condition regarding the block size. A downside is that it limits the partitioner to have the same number of processes and blocks.

4.2.3 Multilevel Heuristic

The multilevel metaheuristic [66] consists of coarsening the input graph in a way that preserves its topology, using an initial partitioning method on the coarse representation, and then uncoarsening it again to project the partitioning solution to the original graph. During uncoarsening, local search methods can be used to improve the solution quality.

This heuristic turned out to be so successful that after it was presented in the mid-90s, most current tools now use some variant of it [33]. An intuition for its effectiveness is that moving single vertices on the coarse level induces large changes in the finer levels, allowing fast progress towards a good solution and avoiding getting stuck in local minima. Also, since the coarse representation is much smaller than the input graph, the initial partitioning can be done with a method that yields high quality but would be too expensive to apply to the whole graph. When edges are merged during coarsening, their edge weights are summed, thus the edge cut of a partition is the same for all levels.

The performance and solution quality of this approach depend on the tools employed in the three phases: In the *coarsening* phase, in which a hierarchy of graphs $G_0, \dots, G_l$ is computed, it is important that the coarsest and thus smallest graph is of manageable size but still topologically similar to the input graph. Depending on the graph type, coarsening based on matchings or clusterings is more effective [1], possibly used together with edge ratings. After coarsening, an initial solution is computed on the coarsest level. Among the many methods used for this step are region growing [135], recursive bisection [85] and spectral methods [68], which exploit the connection between a graph's topology and the Eigenvalues of its Laplacian matrix. For local refinement during uncoarsening, variants of the Fiduccia-Mattheyses method remain popular.

Probably the most popular implementation of the multilevel heuristic with Fiduccia-Mattheyses-style local refinement is the parallel tool ParMetis [145], which is particularly appreciated for its fast running time. Other parallel graph partitioners include PT-Scotch [131] and the parallel versions of Jostle [167], DibaP [108], and KaHIP [111].

While yielding high solution quality, the construction of successively smaller graphs seems to be limited in scalability, though. The tool xtraPulp [151] uses distributed label propagation, avoiding the scalability issues with the multi-level method. This makes it specifically apt for complex networks, when applied to meshes it yields a reduced quality.

4.2.4 Geometric Methods

When the input consists of a mesh with attached geometry information, geometric partitioning methods can be used; those are often significantly faster.

Due to its relevance in numerical simulations, numerous survey articles and books cover the partitioning of meshes from numerical simulations [144, 33, 21]. Most geometric partitioning techniques in wide use consider points and optimize for load balance [43, 103, 150]. Established geometric methods include the recursive coordinate bisection (RCB [150, 20]) and recursive inertial bisection (RIB [155, 170]). Deveci et al. [43] introduce a multisection algorithm, called MultiJagged (MJ), as a generalization of the traditional recursive techniques. The space is divided into rectangles while minimizing the weight of the largest rectangle. This method has better running time and is more scalable but yields less balanced partitions compared to RCB. Many common geometric partitioning methods are implemented in the Zoltan toolbox [26].

Another class of geometric techniques uses space-filling curves (SFCs), usually the Hilbert curve [15] discussed in Section 2.3. These techniques are also fast and scalable and rely on the fact that two points whose indices on the curve are close, are also often close in the original space. While load balance is fairly easy to maintain, the quality of the computed partitions in terms of graph-based methods is relatively poor for non-trivial meshes [75]. Implementations of partitioning algorithms using space-filling curves are available in the ParMetis and Zoltan packages.

4.2.5 Shape optimization

The benefit of optimizing block shapes has been acknowledged in a number of publications, not only for certain applications [45], but also for established graph metrics [110]. However, previous shape-optimizing approaches suffer from a relatively high running time for static partitioning [109, 110, 57] and limited scalability [45, 108].

The bubble framework introduced by Diekmann et al. [45] achieves well-shaped partitions by repeatedly selecting center vertices and growing blocks around them using constrained breadth-first search. This concept is similar to the k-means problem (discussed in Section 4.2.6 below), except that cluster membership and centers are computed with graph-theoretic instead of geometric distances. Due to the discrete nature of graph distances, the center selection can be computationally demanding.

Shape optimization with Bubble-FOS/C [110], a variation of the bubble framework with diffusion distances for the part resembling k-means, has been shown to have some theoretical foundation [112]: similar to spectral partitioning, it computes the global optimum of a relaxed edge cut optimization problem. At the same time, the quality of diffusive partitioning is in practice typically higher than that of spectral methods.

4.2.6 k-Means

The k-means problem, common in unsupervised machine learning and clustering, consists of a set of points P in d-dimensional space $\mathbb{R}^d$ and a number k of target clusters. It asks for an assignment C of the points in P to k clusters so that the sum of squared distances of each point to the mean of its cluster is minimized. This target function is as follows, with clusters denoted as c:

$$f(C) = \sum_{c \in C} \sum_{p \in c} \mathrm{dist}(p, \mathrm{center}(c)). \tag{4.4}$$

Optimizing this target function (Eq. 4.4) yields clusterings with small within-cluster distance and larger between-cluster distance, which is often considered important for a good clustering [46]. In practice, it is useful in load balancing parallel computations on geometric datasets [107]. Also, even though this target function bears no relation to our graph-based metrics, local minima yield convex Voronoi diagrams, useful for our geometric partitioning phase with shape optimization. Unfortunately, optimizing it is $\mathcal{NP}$-hard, even in the Euclidean plane [105].

The first successful heuristic for the k-means problem in Euclidean space is Lloyd's greedy algorithm [101]. It consists of repeatedly alternating two steps:

- For every point $p \in P$: Assign p to the cluster c so that the distance between p and center(c) is minimized.
- For every cluster c: Set cluster center center(c) to the arithmetic mean of all points in c.

The algorithm stops when the maximum movement of cluster centers is below a user-defined threshold, at the latest if no cluster membership changes.

In each step, the sum of squared distances between each point and the center of its cluster decreases. As distances are nonnegative and the number of possible cluster assignments is bounded, the algorithm eventually converges to a local optimum. The number of necessary iterations is exponential in the worst case (bounded by the number of possible assignments $\mathcal{O}(k^n)$), but in practice polynomial. Arthur et al. [9] bridge this difference by showing that input instances with exponential running time need to be carefully constructed; randomly perturbing any input data instance will again yield expected polynomial running time.

Which local optimum is reached after convergence depends on the choice of initial centers. A straightforward option is to choose them uniformly at random among the input points, with erratic and arbitrarily bad results [10]. Alternatives include K-Means++ [10], which chooses the first center at random and then iteratively chooses each subsequent center to maximize the distance to all existing centers. This method yields an expected approximation ratio of $\mathcal{O}(\log n)$. Unfortunately, it is inherently sequential and the k passes over n points lead to a complexity of $\mathcal{O}(nk)$.

Bachem et al. [13] present a probabilistic seeding method using *Markov Chain Monte Carlo* (MCMC) sampling and claim an effective complexity of $\mathcal{O}(n + k)$ for similar quality. Still, the empirical running times are in the order of minutes for a few million points [13].

Several approaches exist to accelerate the main phase of Lloyd's algorithm, many of them exploiting the triangle inequality. Elkan [51] keeps one upper bound and k lower bounds for each data point p, avoiding distance calculations to clusters that cannot possibly be the closest one. Hamerly et al. [64] simplify this approach to one upper bound for the distance of each point to its own center and one lower bound for the distance to the *second-closest* center. Figure 4.2 shows a visualization. When for a point p this first bound is below the second, the cluster membership of p cannot have changed and the loop over all centers can be skipped. In both works, the bounds are relaxed when the cluster centers move and are updated to their exact values when a distance calculation becomes necessary.

As the computations of distances between points and centers happens independently for each point, this step can be parallelized easily [44].

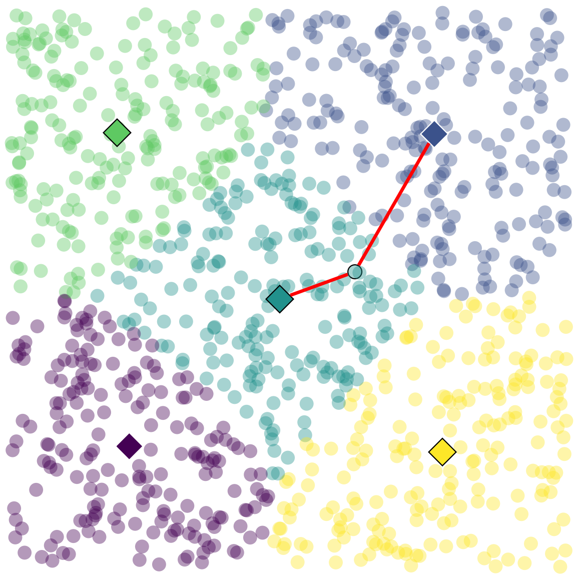


Figure 4.2: Visualization of a k-means partition on uniformly distributed points in the plane. Cluster centers are marked as colored diamonds and each point has the color of its closest center. The marked point is closer to the teal center than to the blue center, thus it is in the teal-colored cluster (lines of shortest distance drawn in red).

The simplicity and empirical effectiveness of k-means have inspired diverse further research, both in improving it and analyzing it theoretically: Hartigan and Wong [65] apply a different objective function and avoid some of the local minima of Lloyd's algorithm [157]. Ding and He [46] investigate the relationship between k-means and spectral dimensionality reduction, showing that principal component analysis can be described as a continuous version of the k-means cluster membership indicators. Sculley [146] presents a sampling method for k-means using gradient descent, with a reported speedup of two orders of magnitude over methods using the triangle inequality.

Balanced k-Means

The classical k-means problem does not require equal sizes of clusters, even empty clusters may occur. Borgwardt et al. [27] present a balanced k-means algorithm for weighted point sets using linear programming. They also bound the number of necessary iterations needed for its convergence. In the solution yielded by their algorithm, a point can have partial membership in several clusters. To the best of our knowledge, no implementation of this algorithm is available.

4.3 Partitioning of Protein Graphs

The biological role of proteins is largely determined by their interactions with other proteins and small molecules. Quantum-chemical methods, such as *Density Functional Theory* (DFT), provide an accurate description of these interactions based on quantum mechanics. A major drawback of DFT is its time complexity, which has been shown to be cubic with respect to the protein size in the worst case [39, 79]. For special cases this complexity

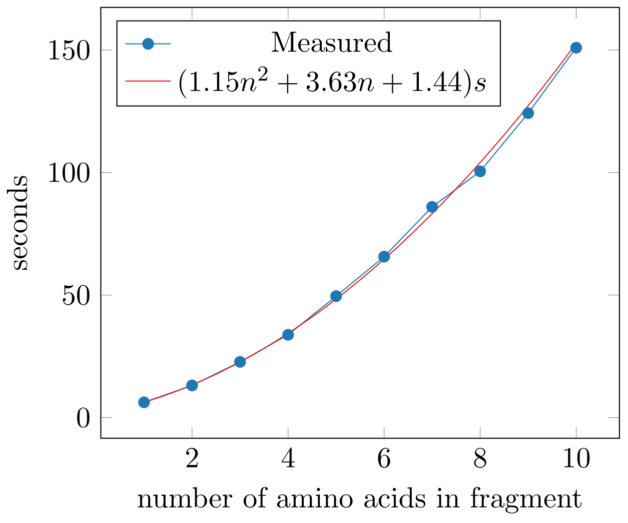


Figure 4.3: Time in seconds required for quantum chemical density functional (DFT, BP86, DZP) calculations of protein fragments on 16 Intel Xeon cores (2x Haswell-EP/2640v3/2.6 GHz) executed with pyADF [76] and ADF program package [156]. As seen by the close match of the red fit line, time grows quadratically.

can be reduced to being linear [123, 62]. Typical DFT implementations, though, show quadratic behavior with significant constant factors, making simulations of proteins bigger than a few hundred amino acids prohibitively expensive [39, 79]. As an example, Figure 4.3 shows an excerpt from experimental running times of quantum-chemical calculations on protein fragments which support this quadratic dependence.

To mitigate the computational cost, quantum-chemical *subsystem methods* have been developed [61, 77]. In such approaches, large molecules are separated into fragments (= subsystems) which are then treated individually. A common way to deal with individual fragments is to assume that they do not interact with each other. The error this introduces for protein-protein or protein-molecule interaction energies (or for other local molecular properties of interest) depends on the size and location of fragments: A partition that cuts right through the strongest interaction in a molecule will give worse results than one that carefully avoids this. It should also be considered that a protein consists of a *main chain* (also called *backbone*) of amino acids. This main chain folds into 3D-secondary-structures, stabilized by non-bonding interactions (those not on the backbone) between the individual amino acids. These different connection types (backbone vs non-backbone) have different influence on the interaction energies.

Motivation

Subsystem methods are very powerful in quantum chemistry [61, 77] but so far require manual cuts with chemical insight to achieve good partitions [86]. Currently, when automating the process, domain scientists typically cut every X amino acids along the main chain (which we will call the *naive approach* in the following). This gives in general suboptimal and unpredictable results.

We construct graphs representing the proteins by considering amino acids as vertices and interactions between them as edges. The edges are weighted with the expected error that would result when assigning the incident vertices to different subsystems and thus neglecting their interactions. Graph partitions with a small cut, i. e. partitions of the vertex set whose inter-fragment edges have low total weight, should then correspond to a low error for interaction energies. A general solution to this problem has high significance, since it is applicable to any subsystem-based method and since it will enable such calculations on larger systems with controlled accuracy. Yet, while several established graph partitioning algorithms exist, none of them are directly applicable to our problem scenarios due to additional domain-specific optimization constraints (which are outlined in Section 4.3.1).

Contributions

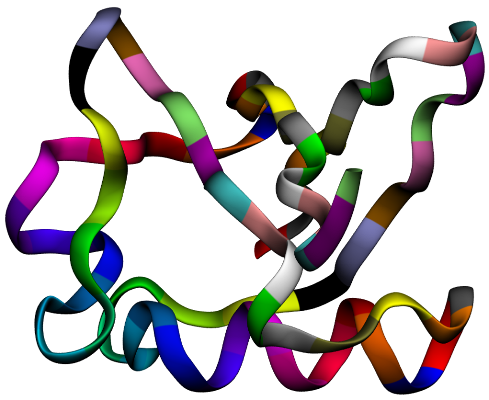
For the first of two problem scenarios, the special case of continuous fragments along the main chain, we provide in Section 4.3.2 a dynamic programming (DP) algorithm. We prove that it yields an optimal solution with a worst-case time complexity of $\mathcal{O}(n^3/k)$ for n vertices and k blocks.

For the general protein partitioning problem, we provide three algorithms using established partitioning concepts, now equipped with techniques for adhering to the new constraints (see Section 4.3.3): (i) a greedy agglomerative method, (ii) a multilevel algorithm with Fiduccia-Mattheyses [55] refinement, and (iii) a simple postprocessing step that "repairs" partitions from external graph partitioners.

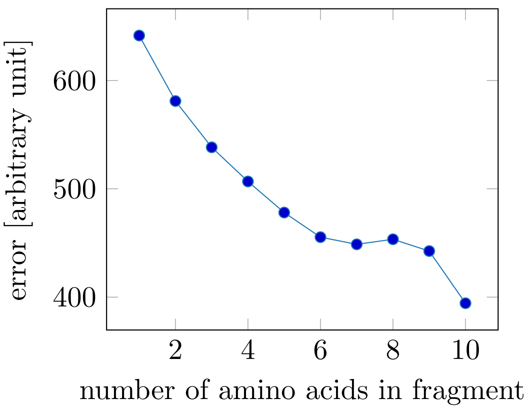
Our experiments (Section 4.3.4) use several protein graphs representative of DFT calculations. They are rather small (up to 357 vertices), but are complete graphs. The results show that our algorithms are usually better in quality than the naive approach. Even though the optimality guarantees of the DP algorithm only holds for the special case of main chain partitioning, its solutions are also the most robust in general protein partitioning, since they are always as least as good in quality as the naive approach. Running all single algorithms and picking the best solution would still take only about ten seconds per instance. This meta algorithm takes a time that is negligible in the quantum-chemical workflow and would improve the expected error compared to the naive approach by 13.5% to 20%, depending on the imbalance.

4.3.1 Modeling

We represent a protein as a weighted undirected graph. Vertices represent amino acids, edges represent bonds or other interactions between amino acids. (This is different from protein interaction networks [129], in which vertices model whole proteins and edges model interactions between them.) Figure 4.4a shows a spatial visualization of Ubiquitin, a common protein, colored on the level of amino acids. Edge weights are determined both by the strength of the bond or interaction and the importance of this edge to the protein function. Such a graph can be constructed from the geometrical structure of the protein using chemical heuristics.



(a) 3D-Visualization of Ubiquitin. Each amino acid is shown in a different color. A helical secondary structure can be seen at the bottom, beta-sheet-like secondary structures in the upper left and right.



(b) Predicted error for interaction energies with naive fragmentation every X amino acids. Unpredictable minima and maxima occur, depending on the location of the cuts along the main chain.

Figure 4.4: Visualization and error prediction for Ubiquitin, a small protein named for its abundance.

Partitioning into fragments yields faster running time for DFT since the time required for a fragment is quadratic in its size. The cut weight of a partition corresponds to the total error caused by dividing this protein into fragments. A balanced partition is desirable as it maximizes this acceleration effect. However, relaxing the constraint with a small $\epsilon > 0$ makes sense as this usually helps in obtaining solutions with a lower error.

Note that the positions on the main chain define an ordering of the vertices. From now on we assume the vertices to be numbered along the chain. Figure 4.4b shows a prediction for the interaction error when partitioning naively along this order.

New Constraints.

Established graph partitioning tools using the model of the previous section cannot be applied directly to our problem since protein partitioning introduces additional constraints due to chemical idiosyncrasies:

- The first constraint is caused by so-called *cap molecules* added for the subsystem calculation. These cap molecules are added at fragment boundaries (only in the DFT, not in our graph) to obtain chemically meaningful fragments. They cannot overlap spatially or be too close, thus cuts in the main chain cannot be too close for the same fragment. This means for the graph that if vertex i and vertex $i + 2$ belong to the same fragment, vertex $i + 1$ must also belong to that fragment. We call this the *gap* constraint; Figure 4.5a shows an example where the gap constraint is violated.

- More importantly, some graph vertices can have a charge. It is difficult to obtain robust convergence in quantum-mechanical calculations for fragments with more than one charged vertex. Therefore, together with the graph a (possibly empty) list of charged vertices is given and two charged vertices must not be in the same fragment. This is called the *charge* constraint, Figure 4.5b shows an example where the charge constraint is violated. Note that, different from how we do it, adding a charge constraint could also be modeled as a special case of graph partitioning with multiple balance constraints.

We consider here **two problem scenarios** (with different chemical interpretations) in the context of protein partitioning:

- **Partitioning along the main chain:** The main chain of a protein gives a natural structure to it. We thus consider a scenario where partition fragments are forced to be continuous on the main chain. This minimizes the number of cap molecules necessary for the simulation and has the additional advantage of better comparability with the naive partition.

  Formally, the problem can be stated like this: Given a graph $G = (V, E)$ with ascending vertex IDs according to the vertex's main chain position, an integer k and a maximum imbalance ϵ, find a k-partition which respects the balance and charge constraints and with minimum cut weight such that

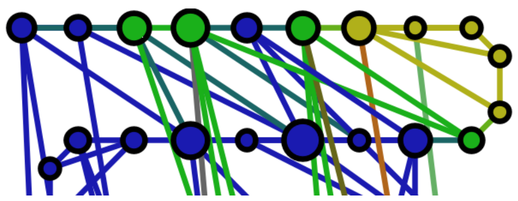
  $$v_j, v_j + l \in V_i \to v_j + 1 \in V_i, l \in \mathbb{N}^+.$$

  The gap constraint is always fulfilled trivially, since all fragments are continuous along the main chain.

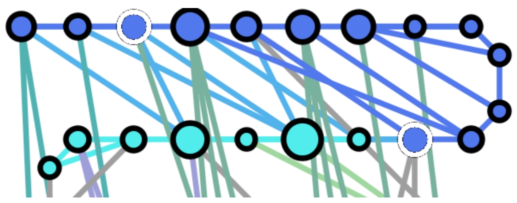
- **General protein partitioning:** The general problem does not require continuous fragments on the main chain, just minimizing the cut weight while adhering to the balance, gap, and charge constraints.

4.3.2 Solving Main Chain Partitioning Optimally

As discussed in the introduction, a protein consists of a main chain, which is folded to yield its characteristic spatial structure. Aligning a partition along the main chain uses the locality information in the vertex order and minimizes the number of cap molecules



(a) Excerpt from a partition where the gap constraint is violated, since vertices 4 and 6 (counting clockwise from the upper left) are in the green fragment, but vertex 5 is in the blue fragment.



(b) Excerpt from a partition where the charge constraint is violated. Vertices 3 and 13 are charged, indicated by the white circles, but are both in the blue fragment.

Figure 4.5: Examples of violated gap and charge constraints, with fragments represented by colors.

necessary for a given number of fragments. The problem of finding fragments with continuous vertex IDs is equivalent to finding a set of $k-1$ *delimiter vertices* $v_{d_1}, v_{d_2}, ...v_{d_{k-1}}$ that separate the fragments. Note that this is not a vertex separator, instead the delimiter vertices induce a set of cut edges due to the continuous vertex IDs. As a convention, delimiter vertex v_{d_j} belongs to fragment j, $1 \leq j \leq k-1$. We further define maxSize as a shorthand for $\lceil \frac{(1+\epsilon)n}{k} \rceil$, as it will come up several times.

Consider the delimiter vertices in ascending order. Given the vertex v_{d_2}, the optimal placement of vertex v_{d_1} only depends on edges among vertices $u < v_{d_2}$, since all edges $\{u, v\}$ from vertices $u < v_{d_2}$ to vertices $v > v_{d_2}$ are cut no matter where v_{d_1} is placed. Placing vertex v_{d_2} thus induces an optimal placement for v_{d_1}, using only information from edges to vertices $u < v_{d_2}$. With this dependency of the positions of v_{d_1} and v_{d_2}, placing vertex v_{d_3} similarly induces an optimal choice for v_{d_2} and v_{d_1}, using only information from vertices smaller than v_{d_3}. The same argument can be continued inductively for vertices $v_{d_4} \dots v_{d_k}$.

Algorithm 10 is our dynamic-programming-based solution to the main chain partitioning problem. It uses the property stated above to iteratively compute the optimal placement of $v_{d_{j-1}}$ for all possible values of v_{d_j}. Finding the optimal placements of $v_{d_1}, \dots v_{d_{j-1}}$ given a delimiter v_{d_j} at vertex i is equivalent to the subproblem of partitioning the first i vertices into j fragments, for increasing values of i and j. If n vertices and k fragments are reached, the desired global solution is found. We allocate (Line 3) and fill an $n \times k$ table partCut with the optimal values for the subproblems. For each i and j, the table entry partCut$[i][j]$ denotes the minimum cut weight of a j-partition of the first i vertices. After the initialization of data structures in Lines 2 and 3, the initial values are set in Line 4: A partition consisting of only one fragment has a cut weight of zero.

All further partitions are built from a *predecessor partition* and a new fragment. A j-partition $\Pi_{i,j}$ of the first i vertices consists of the jth fragment and a $(j-1)$-partition with fewer than i vertices. A valid predecessor partition of $\Pi_{i,j}$ is a partition $\Pi_{l,j-1}$ of the first l vertices, with l between $i -$ maxSize and $i-1$. Vertex charges have to be taken into account when compiling the set of valid predecessors. If a backwards search for $\Pi_{i,j}$ from vertex i encounters two charged vertices a and b with $a < b$, all valid predecessors of $\Pi_{i,j}$ contain at least vertex a (Line 7).

The additional cut weight induced by adding a fragment containing the vertices $[l+1, i]$ to a predecessor partition $\Pi_{l,j-1}$ is the weight sum of edges connecting vertices in $[1, l]$ to vertices in $[l+1, i]$: $c[l][i] = \sum_{\{u,v\} \in E, u \in [1,l], v \in [l+1,i]} w(u, v)$. Line 8 computes this weight difference for the current vertex i and all valid predecessors l.

For each i and j, the partition $\Pi_{i,j}$ with the minimum cut weight is then found in Line 10 by iterating backwards over all valid predecessor partitions and selecting the one leading to the minimum cut. To reconstruct the partition, we store the predecessor in each step (Line 11). If no partition with the given values is possible, the corresponding entry in partCut remains at ∞.

After the table is filled, the resulting minimum cut weight is at partCut$[n][k]$, the corresponding partition is found by following the predecessors (Line 14).

Algorithm 10: Main Chain Partitioning with Dynamic Programming

```
Input: Graph $G = (V, E)$, fragment count $k$, bool list $isCharged$, imbalance $\epsilon$
Output: partition $\Pi$
1  maxSize = $\lceil |V|/k \rceil \cdot (1 + \epsilon)$;
2  allocate empty partition $\Pi$;
3  partCut[i][j] = $\infty, \forall i \in [1, n], \forall j \in [1, k]$;
   /* initialize empty table partCut with $n$ rows and $k$ columns */
4  partCut[i][1] = 0, $\forall i \in [1, \text{maxSize}]$;
5  for $1 \leq i \leq n$ do
6      windowStart = max($i$ − maxSize, 1);
7      if necessary, increase windowStart so that [windowStart, i] contains at most one
         charged vertex;
8      compute column $i$ of cut cost table $c$;
9      for $2 \leq j \leq k$ do
10         partCut[i][j] = $\min_{l \in [windowStart, i]}$ partCut[$l$][$j-1$] + $c$[$l$][$i$];
11         pred[i][j] = $\text{argmin}_{l \in [windowStart, i]}$ partCut[$l$][$j-1$] + $c$[$l$][$i$];
12 set $i = n$;
13 for $j = k$; $j \geq 2$; $j- = 1$ do
14     $nextI$ = pred[$i$][$j$];
15     assign vertices between $nextI$ and $i$ to fragment $\Pi_j$;
16     i = $nextI$;
17 return $\Pi$
```

Correctness

The correctness of Algorithm 10 depends on the correct filling of table partCut:

Lemma 17. *After the execution of Algorithm 10,* partCut[i][j] *contains the minimum cut value for a continuous j-partition of the first i vertices. If such a partition is impossible,* partCut[i][j] *contains ∞.*

Proof. By induction over the number of partitions j.

Base Case: $j = 1, \forall i$. A 1-partition is a continuous block of vertices. The cut value is zero exactly if the first i vertices contain at most one charge and i is not larger than maxSize. This cut value is written into partCut in Lines 3 and 4 and not changed afterwards.

Inductive Step: $j - 1 \rightarrow j$. Let i be the current vertex: A cut-minimal j-partition $\Pi_{i,j}$ for the first i vertices contains a cut-minimal $(j-1)$-partition $\Pi_{i',j-1}$ with continuous vertex blocks. If $\Pi_{i',j-1}$ were not minimum, we could find a better partition $\Pi'_{i',j-1}$ and use it to improve $\Pi_{i,j}$, a contradiction to $\Pi_{i,j}$ being cut-minimal. Due to the induction hypothesis, partCut[l][$j-1$] contains the minimum cut value for all vertex indices l, which includes i'. The loop in Line 10 iterates over possible predecessor partitions $\Pi_{l,j-1}$ and selects the one leading to the minimum cut after vertex i. Given that partitions for $j-1$ are cut-minimal, the partition whose weight is stored in partCut[i][j] is cut-minimal as well.

If no allowed predecessor partition with a finite weight exists, partCut[i][j] remains at infinity. □

Time Complexity

Theorem 5. *Algorithm 10 computes the optimal main chain partition in time* $\mathcal{O}(n^2 \cdot \text{maxSize})$.

Proof. The nested loops in Lines 5 and 9 require $\mathcal{O}(n \cdot k)$ iterations in total. Line 7 is executed n times and has a complexity of maxSize. At Line 10 in the inner loop, up to maxSize predecessor partitions need to be evaluated, each with two constant time table accesses. Computing the cut weight column $c[\cdot][i]$ for fragments ending at vertex i (Line 8) involves summing over the edges of $\mathcal{O}(\text{maxSize})$ predecessors, each having at most $\mathcal{O}(n)$ neighbors. Since the cut weights constitute a reverse prefix sum, the column $c[\cdot][i]$ can be computed in $\mathcal{O}(n \cdot \text{maxSize})$ time by iterating backwards. Line 8 is executed n times, leading to a total complexity of $\mathcal{O}(n^2 \cdot \text{maxSize})$. Following the predecessors and assigning vertices to fragments is possible in linear time, thus the $\mathcal{O}(n^2 \cdot \text{maxSize})$ to compile the cut cost table dominates the running time. □

4.3.3 Algorithms for General Protein Partitioning

As discussed in Section 4.3.1, general-purpose graph partitioning programs in general do not fulfill the new constraints required by the DFT calculations. The DP-based algorithm, while optimal for main chain partitioning, is also not optimal in general.

Thus, we propose three algorithms for the general problem: The first two, a greedy agglomerative method and Multilevel-FM, build on existing graph partitioning knowledge but incorporate the new constraints directly into the optimization process. The third one is a simple postprocessing repair procedure that works in many cases. It takes the output of a traditional graph partitioner and fixes it so as to fulfill the constraints.

Greedy Agglomerative Algorithm

The greedy agglomerative approach, shown in Algorithm 11, is similar in spirit to Kruskal's MST algorithm and to approaches proposed for clustering graphs with respect to the objective function modularity [37]. It initially sorts edges by weight and puts each vertex into a singleton fragment. Edges are then considered iteratively with the heaviest first; the fragments belonging to the incident vertices are merged if no constraints are violated. This is repeated until no edges are left or the desired fragment count is achieved.

The initial edge sorting takes $\mathcal{O}(m \log m)$ time. Initializing the data structures is possible in linear time. The main loop (Line 5) has at most m iterations. Checking the size and charge constraints is possible in constant time by keeping arrays of fragment sizes and charge states. The time needed for checking the gaps and merging is linear in the fragment size and thus at most $\mathcal{O}(\text{maxSize})$.

The total time complexity of the greedy algorithm is thus:

$$T(\text{Greedy}) \in \mathcal{O}(m \cdot \max\{\text{maxSize}, \log m\}).$$

Algorithm 11: Greedy Agglomerative Algorithm

```
Input: Graph $G = (V, E)$, fragment count $k$, list charged, imbalance $\epsilon$
Output: partition $\Pi$
1  $\tilde{E}$ = sort edges $E$ by weight, descending;
2  $\Pi$ = create one singleton partition for each vertex;
3  chargedPart = partitions containing a charged vertex;
4  maxSize = $\lceil |V|/k \rceil \cdot (1 + \epsilon)$;
5  for edge $\{u, v\} \in \tilde{E}$ do
6      allowed = True;
7      if $\Pi[u] \in$ chargedPart and $\Pi[v] \in$ chargedPart then    // Charge Constraint
8          allowed = False;
9      if $|\Pi[u]| + |\Pi[v]| >$ maxSize then                       // Balance Constraint
10         allowed = False;
11     for vertex $x \in \Pi[u] \cup \Pi[v]$ do                          // Gap Constraint
12         if $x + 2 \in \Pi[u] \cup \Pi[v]$ and $x + 1 \notin \Pi[u] \cup \Pi[v]$ then
13             allowed = False;
14     if allowed then
15         merge $\Pi[u]$ and $\Pi[v]$;
16         update chargedPartitions;
17     if number of fragments in $\Pi$ equals $k$ then
18         break;
19 return $\Pi$
```

Multilevel Algorithm with Fiduccia-Mattheyses Local Search

Algorithm 12 uses non-binary (i. e. $k > 2$) Fiduccia-Mattheyses (FM) local search, as do existing multilevel partitioners. Our adaptation incorporates the constraints throughout the whole partitioning process. First a hierarchy of graphs $G_0, G_1, \ldots G_l$ is created by recursive coarsening (Line 1). The edges contracted during coarsening are chosen with a local matching strategy. An edge connecting two charged vertices stays uncontracted, thus ensuring that a fragment contains at most one charged vertex even in the coarsest partitioning phase. The coarsest graph is then partitioned into Π_l using region growing or recursive bisection. If an optional input partition Π' is given, it is used as a guideline during coarsening and replaces Π_l if it yields a better cut. We execute both our greedy and DP algorithm and use the partition with the better cut as input partition Π' for the multilevel algorithm.

After obtaining a partition for the coarsest graph, the graph is iteratively uncoarsened and the partition projected to the next finer level. As we relax the balance constraint for coarser levels for greater flexibility, a rebalancing step is necessary after uncoarsening (Line 6). A Fiduccia-Mattheyses step is then performed to yield local improvements (Line 10): For a partition with k fragments, this non-binary FM step consists of one priority queue for each fragment. Each vertex v is inserted into the priority queue of its current fragment, the maximum gain (i. e. reduction in cut weight when v is moved to another fragment) is used as key. While at least one queue is non-empty, the highest vertex of the largest queue is moved if the constraints are still fulfilled, and the movement recorded. After all vertices

Algorithm 12: Multilevel-FM

```
   Input: Graph $G = (V, E)$, fragment count $k$, list charged, imbalance $\epsilon$, $[\Pi']$
   Output: partition $\Pi$
1  $G_0, \ldots, G_l$ = hierarchy of coarsened Graphs, $G_0 = G$;
2  $\Pi_l$ = partition $G_l$ with region growing or recursive bisection;
3  for $0 \leq i < l$ do
4      uncoarsen $G_i$ from $G_{i+1}$;
5      $\Pi_i$ = projected partition from $\Pi_{i+1}$;
6      rebalance $\Pi_i$;
       /* Local improvements                                    */
7      gain = NaN;
8      repeat
9          oldcut = cut($\Pi'_i, G$);
10         $\Pi'_i$ = Fiduccia-Mattheyses-Step of $\Pi_i$ with constraints;
11         gain = cut($\Pi'_i, G$) - oldcut;
12     until gain == 0;
```

have been moved, the partition yielding the minimum cut is taken. Of course, vertices are only moved if the charge constraint stays fulfilled.

Repair Procedure

To be able to use existing tools, we propose a simple repair procedure (shown in Algorithm 13) that takes an existing partition and modifies it to fulfill the charge and gap constraints. It performs three sweeps over all vertices: In the first sweep (Line 6), only the charge constraint is considered and fixed. For each fragment with more than one charged vertex, all but one of the charged vertices are moved to previously uncharged fragments. They are moved to those fragments where the move yields the highest gain, not considering the gap or size constraints. In the second sweep (Line 11), violations of the gap constraint are repaired in a way that does not violate the charge constraint. This is achieved by swapping the block assignment of a vertex v within a gap with its right neighbor, or embedding v in the surrounding block. After the second sweep, the gap and charge constraints are fulfilled, but the blocks are possibly imbalanced. The third sweep (Line 14) thus consists of a modified Fiduccia-Mattheyses step: Vertices in oversized blocks are moved to other blocks, as long as this move would not violate a gap, size or charge constraint. Among its possible targets, each surplus vertex v is moved to the block where the move yields the highest gain.

The changes made by Algorithm 13 will most likely increase the edge cut compared to the "raw" external partition, since some low-cut solutions are no longer possible. However, it is not impossible to find a better solution during the repair procedure. In Line 20, a new block is added if no other way can be found to satisfy the constraints. In the intended application, the total edge cut matters more than the number of blocks, thus this solution might still be valuable.

Algorithm 13: Repairing an external partition

```
Input: Graph $G = (V, E)$, $k$-partition $\Pi$, set $C \subseteq V$ of charged vertices, imbalance $\epsilon$
Output: partition $\Pi'$
1  $k \leftarrow |\Pi|$;
2  cutWeight$[i][j] = 0, 1 \leq i \leq n, 1 \leq j \leq k$;
3  for edge $\{u, v\}$ in $E$ do
4      cutWeight$[u][\Pi(u)] += w(u, v)$;
5      cutWeight$[v][\Pi(v)] += w(u, v)$;
6  for block $b \in \Psi$ do                                          // First Sweep
7      while number of charged vertices in $b$ > 1 do
8          $\Psi \leftarrow \{t$ without charged vertices$|t \in \Pi\}$;
9          $v$, target $\leftarrow \text{argmax}_{v \in b \cap C, t \in \Psi}$ cutWeight$[v][t]$;
10         move $v$ to target, update cutWeight;
11 for vertex $0 \leq v < |V|$ do                                     // Second Sweep
12     if $v - 1, v + 1 \in V$ and $\Pi[v-1] == \Pi[v+1] \neq \Pi[v]$ then
13         repair gap at $v$ with local swaps while preserving charges;
14 for vertex $v$ in $V$ do                                           // Third Sweep
15     $b \leftarrow$ block of $v$;
16     if $|b| > \lceil \frac{(1+\epsilon)n}{k} \rceil$ then
17         $\Psi \leftarrow \{t$ acceptable target for $v|t \in \Pi\}$;
18         target $= \text{argmax}_{t \in \Psi}\{$cutWeight$[v][t]\}$;
19         if $\Psi$ is empty then
20             target $\leftarrow$ create new fragment for $v$;
21         move $v$ to target, update cutWeight;
```

Time complexity.

The cut weight table allocated in Line 2 takes $\mathcal{O}(n \cdot k + m)$ time to create. Whether a constraint is violated can be checked in constant time per vertex by counting the number of vertices and charges observed for each fragment. Finding the best target partition (Lines 9 and 18) takes $\mathcal{O}(k)$ iterations, updating the cut weight table after moving a vertex v is linear in the degree $\deg(v)$ of v. The total time complexity of a repair step is thus: $\mathcal{O}(n \cdot k + m + n \cdot k + \sum_v \deg(v)) = \mathcal{O}(n \cdot k + m)$.

4.3.4 Experiments

We evaluate our algorithms on graphs derived from several proteins and compare the resulting cut weight. As main chain partitioning is a special case of general protein partitioning, the solutions generated by our dynamic programming algorithm are valid solutions of the general problem, though perhaps not optimal. Other algorithms evaluated are Algorithm 11 (Greedy), 12 (Multilevel), and the external partitioner KaHiP [142], used with the repair step discussed in Section 12. The algorithms are implemented in C++ and Python using the NetworKit tool suite [154], the source code is available from a git repository.[14] Scripts and a docker image to recreate the experiments are available at DOI 10.5445/IR/1000095237 on the KITOpenData repository.

[14]`https://github.com/kit-parco/networkit-chemfork`

We use graphs derived from five common proteins, covering the most frequent structural properties. Ubiquitin [137] and the Bubble Protein [124] are rather small proteins with 76 and 64 amino acids, respectively. Due to their biological functions, their overall size and their diversity in the contained structural features, they are commonly used as test cases for quantum-chemical subsystem methods [86]. The Green Fluorescent Protein (GFP) [125] plays a crucial role in the bioluminescence of marine organisms and is widely expressed in other organisms as a fluorescent label for microscopic techniques. Like GFP, the protein Bacteriorhodopsin (bR) [98] and the Fenna-Matthews-Olson protein (FMO) [159] are large enough to render quantum-chemical calculations on the whole proteins practically infeasible. Yet, investigating them with quantum-chemical methods is key to understanding the photochemical processes they are involved in. The graphs derived from the latter three proteins have 225, 226 and 357 vertices, respectively. They are complete graphs with weighted $n(n-1)/2$ edges. All instances can be found in the mentioned git repository in folder `input/`.

In our experiments we partition the graphs into fragments of different sizes (i. e. we vary the fragment number k). The small proteins ubiquitin and bubble are partitioned into 2, 4, 6 and 8 fragments, leading to fragments of average size 8–38. The other proteins are partitioned into 8, 12, 16, 20 and 24 fragments, yielding average sizes between 10 and 45. As maximum imbalance, we use values for ϵ of 0.1 and 0.2. While this may be larger than usual values of ϵ in graph partitioning, fragment sizes in our case are comparably small and an imbalance of 0.1 is possibly reached with the movement of a single vertex.

On these proteins, the running time of all partitioning implementations is on the order of a few seconds even with small machines, we therefore omit detailed time measurements, especially since the partitioning time is insignificant in the whole quantum-chemical workflow.

Charged Vertices.

Some of the amino acids in proteins may carry charges. Each protein has some *potentially charged* amino acids, whether they are actually charged depends on the environment. As discussed in Section 4.3.1, at most one charge is allowed per fragment. We repeatedly sample $\lfloor 0.8 \cdot k \rfloor$ random charged vertices among the potentially charged, under the constraint that a valid main chain partition is still possible. To smooth out random effects, we perform 20 runs with different random vertex charges. Introducing charged vertices may cause the naive partition to become invalid. In these cases, we use the repair procedure on the invalid naive partition and compare the cut weights of other algorithms with the cut weight of the repaired naive partition.

Results

For the uncharged scenario, Figure 4.6 shows a comparison of cut weights for different numbers of fragments and a maximum imbalance of 0.1. The cut weight is up to 34.5% smaller than with the naive approach (or 42.8% with $\epsilon = 0.2$). The best algorithm choice depends on the protein: For Ubiquitin, the green fluorescent protein, and the Fenna-Matthew-Olson protein, the external partitioner KaHiP in combination with the repair

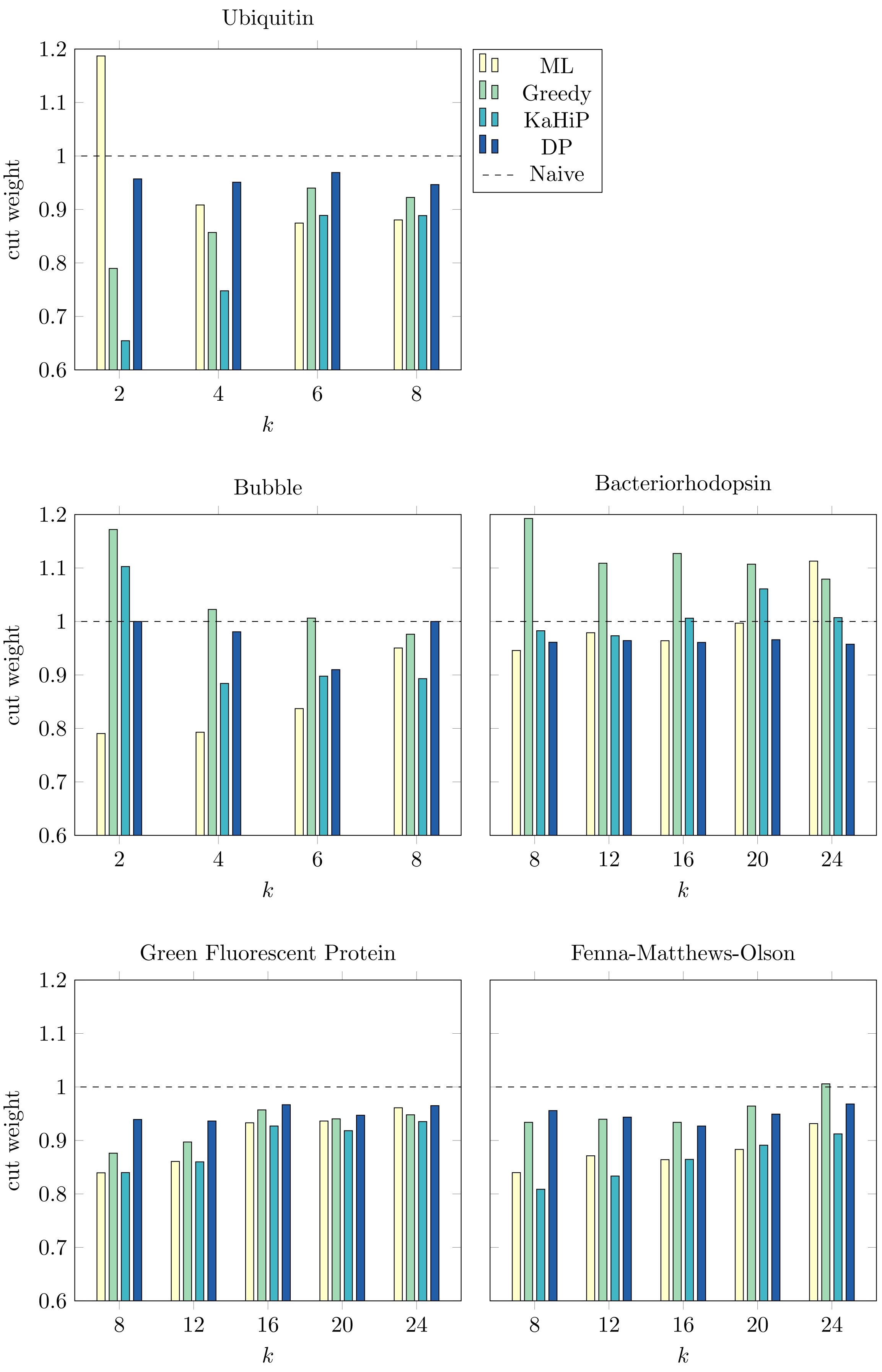


Figure 4.6: Comparison of partitions given by several algorithms and proteins, for $\epsilon = 0.1$. The partition quality is measured by the cut weight in comparison to the naive solution.

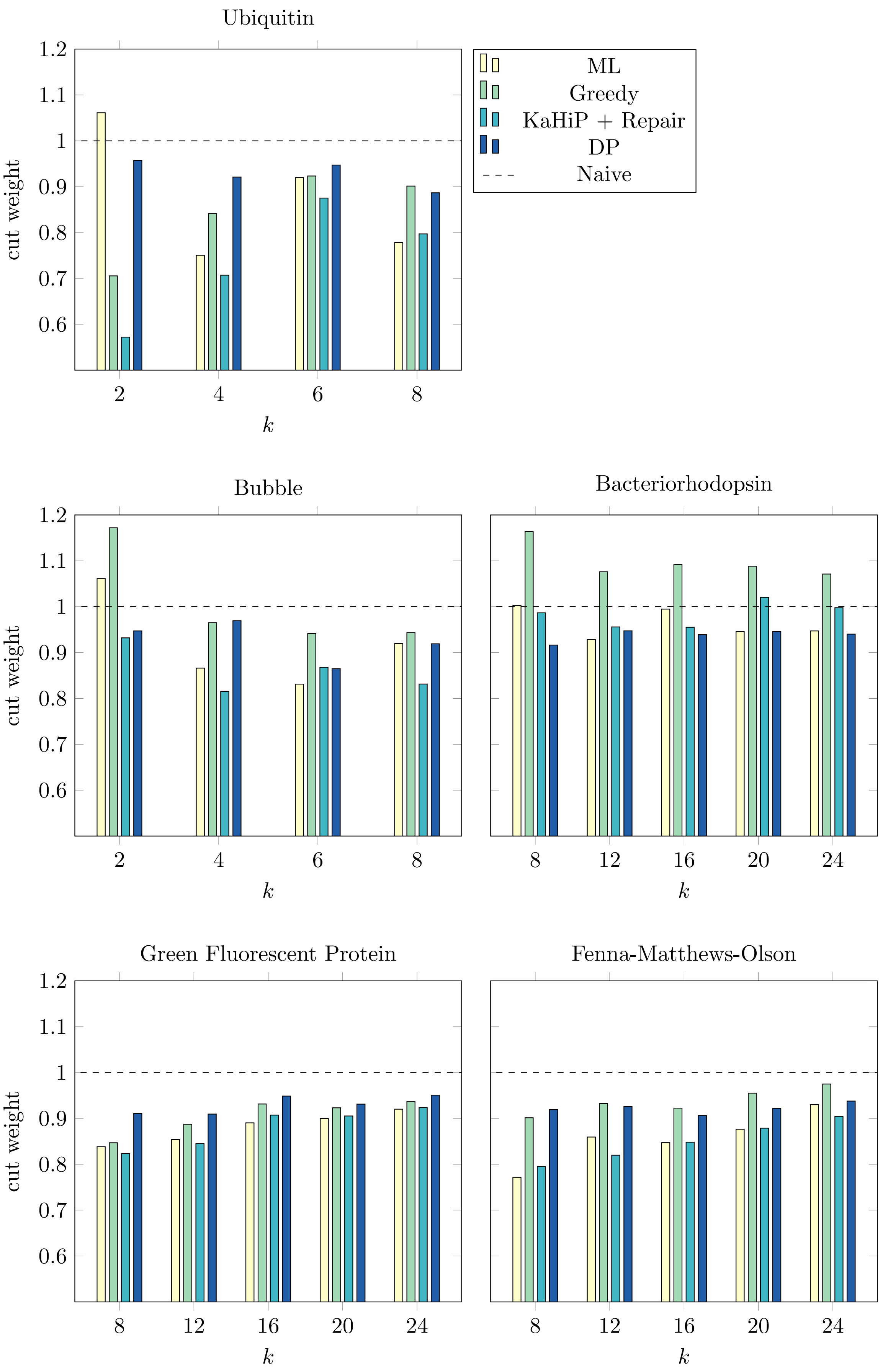


Figure 4.7: Comparison of cut weights for $\epsilon = 0.2$.

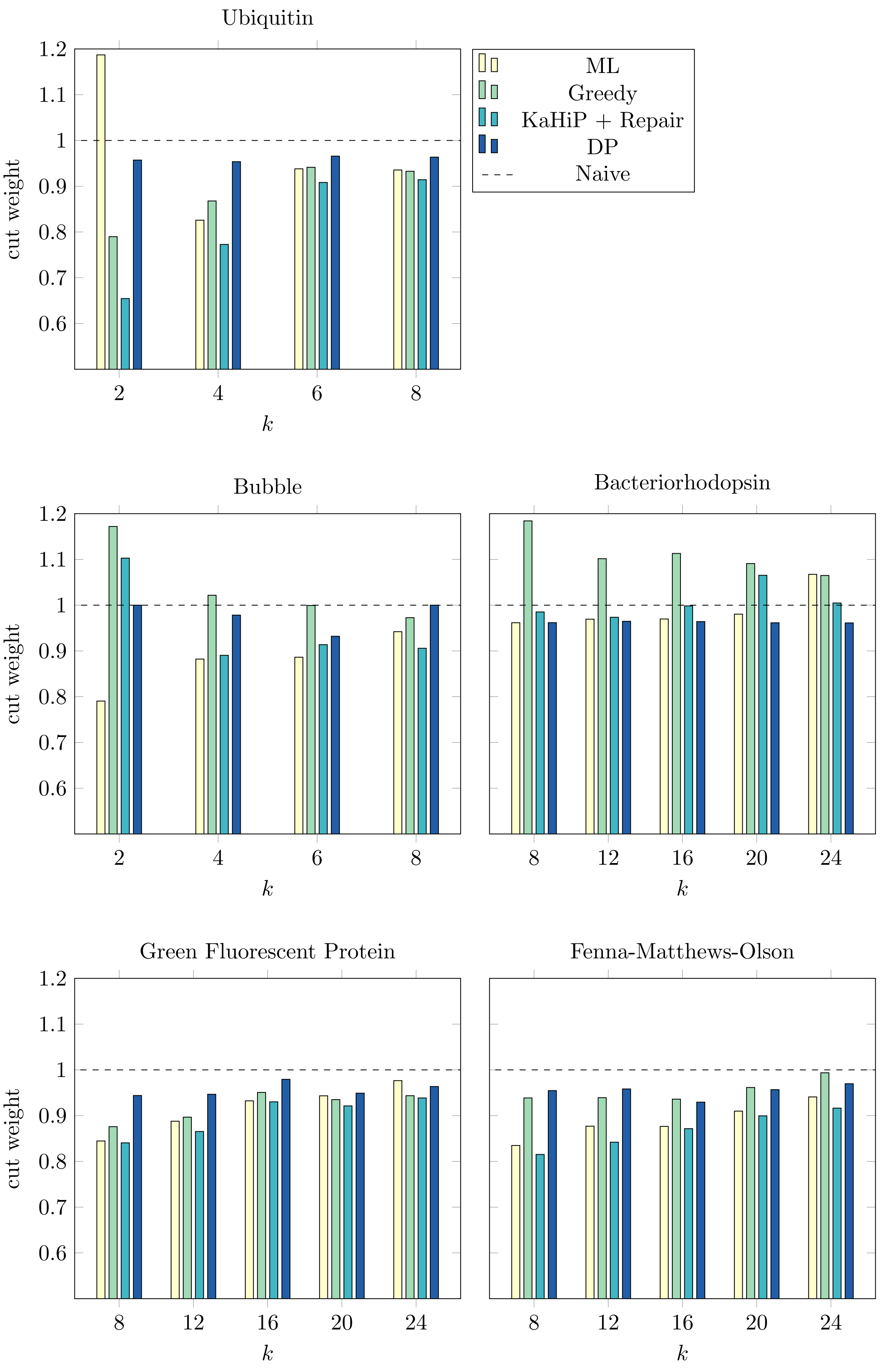


Figure 4.8: Comparison of cut weights for $\epsilon = 0.1$ and vertex charges.

step described in Section 12 gives the lowest cut weight when averaged over different fragment sizes. For the bubble protein, the multilevel algorithm from Section 19 gives on average the best result, while for bacteriorhodopsin, the best cut weight is achieved by the dynamic programming (DP) algorithm. The DP algorithm is always as least as good as the naive approach. This already follows from Theorem 5, as the naive partition is aligned along the main chain and thus found by DP in case it is optimal. DP is the only algorithm with this property, all others perform worse than the naive approach for at least one combination of parameters.

The general intuition that smaller fragment sizes leave less room for improvements compared to the naive solution is confirmed by our experimental results. Figure 4.7 shows the comparison with imbalance $\epsilon = 0.2$. While the general trend is similar and the best choice of algorithm depends on the protein, the cut weight is usually more clearly improved. Moreover, a meta algorithm that executes all single algorithms and picks their best solution yields average improvements (geometric mean) of $13.5\%, 16\%$, and 20% for $\epsilon = 0.1, 0.2$, and 0.3, respectively, compared to the naive reference. Such a meta algorithm requires only about ten seconds per instance, negligible in the whole DFT workflow.

Randomly charging vertices changes the results only insignificantly, as seen in Figure 4.8. The necessary increase in cut weight for the algorithm's solutions is likely compensated by a similar increase in the naive partition due to the necessary repairs.

4.3.5 Concluding Remarks and Future Work

Partitioning protein graphs for subsystem quantum-chemistry is a problem with unique constraints which general-purpose graph partitioning algorithms were unable to handle. Thus, several simple algorithmic approaches were sufficient to achieve partitions of significantly improved quality. For the special case of partitioning along the main chain, dynamic programming already yields an optimal algorithm. With our algorithms chemists are now able to address larger problems in an automated manner with smaller error.

Promising future work exists on both sides of this collaboration. On the chemical side, usability of results could be improved by modeling the approximation error and chemical constraints more precisely. For example, preliminary results indicate that also gaps of two or three amino acids are detrimental to the simulation fidelity, as close proximity of cap molecules induces spurious interactions.

Algorithmically, the resulting gaps of arbitrary size cannot easily be repaired with local swaps, requiring a more general solution. Further complicating our abstraction, errors in density calculations can have both positive and negative signs. They can thus cancel out, leading to a lower total error than expected when considering them individually. The corresponding optimization problem would be *mixed-sign graph partitioning problem* – something we have never seen before in the literature. We adapted our dynamic programming solution to accept mixed-sign edge weights and minimize the absolute sum, further research on its effectiveness in practice is required. Concluding, this first foray into the partitioning of protein graphs already yielded significant benefits, but is far from exhaustive.

4.4 Balanced k-means for Parallel Geometric Partitioning

In the context of the WAVE project, we target geometric meshes with billions of vertices, partitioned into tens of thousands of well-shaped blocks. A high degree of parallelism is required to solve these problems in reasonable time. Established graph partitioners mostly use the multi-level approach to achieve a good edge cut [33], but unfortunately, previous work showed an increase in running time of this approach when using more than a few hundred processes [73, 88, 111].

Thus, for large-scale simulations the research community has moved to more scalable geometric methods, e. g. space-filling curves [15], or seemingly simpler space-partitioning methods [43]. Their solution quality leaves something to be desired, though.

Our motivation is thus: At a range of parallelism past where the multi-level method is useful, can we get a better quality than existing geometric approaches?

Besides the traditional quality metrics, we are also interested in good block shapes: Connected, compact, to some extent convex. These are not only beneficial for certain applications [45], often they tend to induce a high partitioning quality w. r. t. established graph metrics [109]. Graph-based tools are usually not satisfactory in this regard unless specifically designed for this purpose [113].

Other key application areas for massively parallel numerical solvers are the atmosphere and ocean simulations in weather and climate models; they feature prominently on the list of exascale challenges [143].

Although the simulations are run in 3D, their vertical extent is typically very small and variable over the application domain. Thus, the mesh tends to be partitioned in 2D and then extended to a 3D mesh during the simulation using topography information; therefore this type of mesh/problem is sometimes called 2.5-dimensional. The computational effort depends on the number of 3D grid points and is reflected in the 2D mesh as a vertex weight.

Following from these requirements, we are interested in a scalable mesh partitioning algorithm for 2D and 3D meshes that yields high quality in terms of block shapes *and* relevant graph metrics.

4.4.1 Weighted Balanced k-Means for Mesh Partitioning

Many requirements of a good geometric partition for parallel load balancing are fulfilled by Lloyd's algorithm: Convex clusters, fast convergence in practice and mostly independent calculations, which imply good scalability [44]. Indeed, k-means has been used for load balancing in parallel n-body physics simulations [107]. The *balancing* part in *load balancing*, however, implies that the clusters should be of equal size. This is not delivered by Lloyd's algorithm, its solutions can be unboundedly imbalanced.

While existing approaches for balancing k-means exist [27], these do not have the other required properties of a scalable load balancer, for example parallel scalability.

Formally, we extend the k-means problem as follows:

Definition 1. *Let P be a set of points in d-dimensional space $\mathbb{R}^d$ and $w : P \to \mathbb{R}^+$ be an optional weight function. Let $k \in \mathbb{N}$ be a target cluster count and $\epsilon \in \mathbb{R}^+$ be the allowed imbalance.*

*The **balanced** k**-means problem** is then: Find a partition of P into disjoint subsets C_i, $0 \leq i < k$, so that the weight $w(C_i)$ of each subset is at most $\lceil \frac{w(P)}{k} \rceil \cdot (1 + \epsilon)$ and the sum of squared point-center distances is minimized.*

This is $\mathcal{NP}$-hard, as it contains the classical k-means problem. When non-uniform block sizes are desired, for example when partitioning for heterogeneous architectures, this can easily be adapted: Let t_i be the target size of block i, the balance constraint then demands that the weight $w(C_i)$ of subset C_i is at most $t_i \cdot (1 + \epsilon)$.

Starting from Lloyd's algorithm (see Section 4.2.6), we discuss the changes we made to address parallelization, balancing and geometric optimizations, finally presenting the overall algorithm.

Parallelization and Space-Filling Curves

Lloyd's algorithm parallelizes well and our extensions do not change that. Each processor stores a subset of the points, while the cluster centers and influence values are replicated globally. The computationally most expensive phase is assigning points to the appropriate cluster, which can be done independently for each point. After points are assigned, k parallel sum operation are performed ($\log p$ each [95]) to calculate the new cluster centers and sizes. Using p processes to perform one k-means iteration of n points into k clusters has then a complexity of $\mathcal{O}(kn/p + k \log p)$.

As preparation for the geometric optimizations, we globally sort and redistribute all points according to their index on a space-filling curve, thus ensuring that each processor has local points that are grouped spatially and their bounding box is reasonably tight. For this distributed sorting step, we use the scalable quicksort implementation of Axtmann et al. [12].

Balancing

To achieve balanced cluster sizes, we add an *influence* value to each cluster, initialized to 1. In the assignment phase, instead of assigning each point p to the cluster with the smallest distance, we assign it to the cluster c for which the term $\text{dist}(p, \text{center}(c))/\text{influence}(c)$ is minimized. We call this term the *weighted distance* of p to $\text{center}(c)$. This approach results in the creation of *weighted Voronoi diagrams* [11] (which are not necessarily convex).

After all points are assigned, the global weight sum is calculated for each block. The influence values of oversized blocks are decreased, those of undersized blocks increased. How strongly to increase or decrease the influence in response to an imbalanced partition is a tuning parameter. Our decision is guided by geometric considerations: The volume of a d-dimensional hypersphere with radius r scales with r^d. Assuming a roughly uniform point density, increasing the weighted distance of a cluster to all points by a factor of b

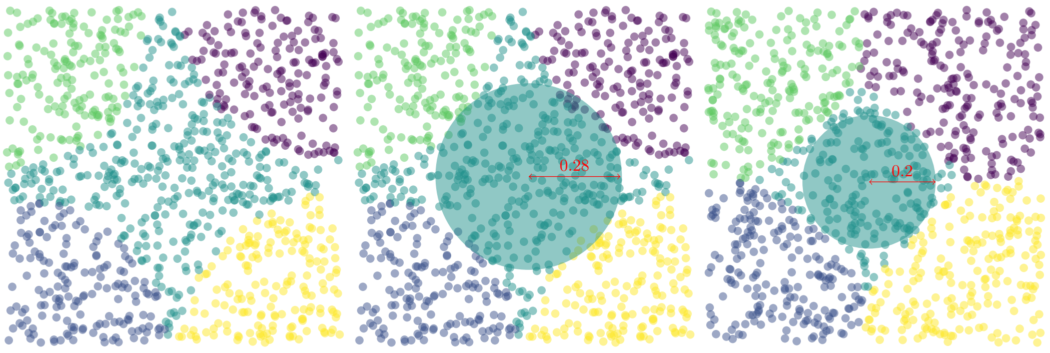


Figure 4.9: Example of an oversized cluster, whose influence is adjusted down to yield better balance.

leads, all else being equal, to a change in size of b^{-d}. Thus, if the ratio of the target size and current size for a cluster c is $\gamma(c)$, we set:

$$\text{influence}[c] \leftarrow \text{influence}[c]/\gamma(c)^{1/d}. \tag{4.5}$$

Then, the new expected size of cluster c is $\left(\frac{1}{\gamma(c)^{1/d}}\right)^{-d} \cdot \text{size}_{\text{old}} = \left(\gamma(c)^{1/d}\right)^{d} \cdot \text{size}_{\text{old}} = \gamma(c) \cdot \text{size}_{\text{old}} = \text{size}_{\text{target}}$.

Figure 4.9 shows an example of influence adjustment on points which are uniformly distributed in the plane. Of course, the input points are usually not uniformly distributed and more than one balance iteration is needed. To prevent oscillations, we restrict the maximum influence change in one step to 5%. This approach is repeated for a maximum number of balancing steps or until the maximum imbalance is at most ϵ; then centers are moved and a new assign-and-balance phase starts. The maximum number of balancing iterations between center movements is a tuning parameter.

In very heterogeneous point distributions, it can happen that clusters need very small or very large influence values to gain a reasonable size. If, after a movement phase, cluster centers with very different influence values are in close proximity, anomalies such as empty or absurdly large clusters might occur. To avoid such cases, we add an *influence erosion* scheme: When cluster centers move, we regress their influence value according to a sigmoid function of the moved distance. Let $\delta(c)$ be the distance that center(c) moved in the last phase and let $\beta(C)$ be the average geometric cluster diameter. We then define an *erosion factor* $\alpha(c)$ between 0 and 1, controlling how strongly the influence is eroded:

$$\alpha(c) = \frac{2}{1 + \exp(\min(-\delta(c)/\beta(C), 0))} - 1$$
$$\text{influence}(c) \leftarrow \exp((1 - \alpha(c)) \cdot \log(\text{influence}(c)))$$

The influence value of stationary centers does not move, but after moving more than the average distance between centers, the influence value is thus almost back to 1. This reflects that an influence appropriate for one neighborhood of clusters might not be appropriate for a different neighborhood.

Geometric Optimizations

We adapt the distance bounds of Hamerly et al. [64] for weighted distances, as nearest-neighbor data structures like kd-trees are outperformed by simpler distance bounds in most published experiments [48, 64].

Our adaptations are thus: Let p be a point and $c := c(p)$ its assigned cluster; then $\mathrm{ub}(p)$ stores an upper bound for the weighted distance of p and c; $\mathrm{lb}(p)$ stores a lower bound for the second-smallest weighted distance. If $\mathrm{lb}(p) > \mathrm{ub}(p)$ holds when evaluating the new cluster assignment of point p, it is still in its previous cluster and distance calculations to other clusters can be skipped.

When a cluster center moves or its influence value changes, these bounds need to be relaxed to stay valid. Again, let $\delta(c)$ be the distance that $\mathrm{center}(c)$ moved in the last phase. For each point p in cluster c, the new upper bound $\mathrm{ub}'(p)$ is then:

$$\mathrm{ub}'(p) = \mathrm{ub}(p) - \delta(c)/\mathrm{influence}(c). \tag{4.6}$$

The lower bounds are relaxed with the maximum combination of δ and influence, as any cluster could be the second-closest one:

$$\mathrm{lb}'(p) = \mathrm{lb}(p) + \max_{c' \in C} \delta(c')/\mathrm{influence}(c'). \tag{4.7}$$

Using these bounds, the innermost loop can be skipped in about 80% of the cases, more in the later phases where centers and influence values change less.

Bounding Boxes

For a given point p, most clusters are unlikely candidates. In fact, the likely cluster centers lie roughly in a d-dimensional hypersphere around p. As discussed earlier (Section 4.4.1), we parallelize k-means by dividing the points among processors and sorting them according to a space-filling curve, achieving some degree of locality. Thus, for each process the bounding box $\mathbf{B}$ around the process-local points covers only a small area and most cluster centers will be outside of it. When assigning point p, we only need to consider centers c with $\mathrm{dist}(c, \mathbf{B}) < \mathrm{ub}(p)$. By sorting the cluster centers by their weighted distance to $\mathbf{B}$, we can avoid evaluating possible clusters for a point p when their minimum weighted distance is above the ones for already found candidates. Figure 4.10 shows a visualization.

In a scenario with uniformly distributed input points, this reduces the expected complexity of one k-means iteration from $\mathcal{O}(kn/p + k \log p)$ to $\mathcal{O}(n/p + k \log p + k \log k)$, as for a fixed volume, the surface of a d-hypersphere is bounded, independently of d [138].

Algorithm

Algorithm 14 shows the resulting assign-and-balance phase of our k-means and is executed by all processors in parallel. The first 5 lines prepare data structures and optimizations,

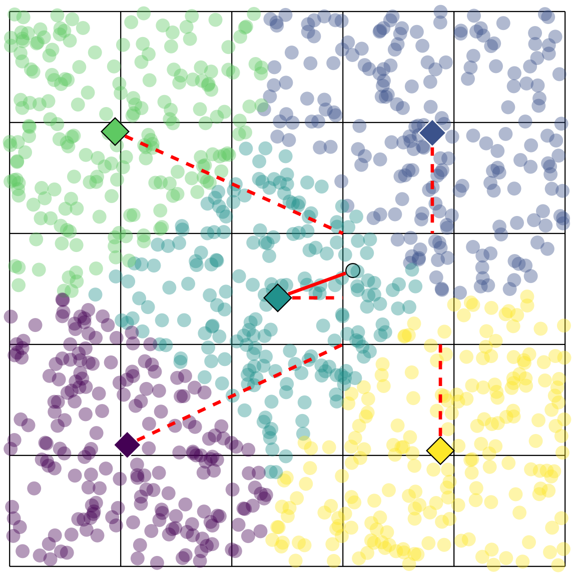


Figure 4.10: Combining Hamerly bounds with bounding boxes. The grid represents the division into process-local points and the dashed lines show the distance between other cluster centers and the bounding box of the marked point. The marked point is closest to the teal cluster center, distance calculations to other cluster centers can be stopped as their distance to the point's bounding box is larger.

the main loop starts in Line 7. If the distance bounds for a point p guarantee that its cluster assignment has not changed, the inner loop can be skipped (Line 9). If not, we iterate over the cluster centers and assign p to the one with the smallest weighted distance (Line 17). As soon as the weighted distance between a cluster center and the bounding box of local points is higher than the second best value found so far, the remaining clusters cannot improve on that and can be skipped (Line 14).

We update bounds for all points where distance calculations were necessary (Lines 22 and 23). Finally, global block sizes are computed as sums of all local block sizes (Line 25). This is the only part requiring communication in the balance routine, marked in blue.

If the global block sizes are imbalanced, we use Eq. (4.5) to adapt the influence values for the next round (Line 29). As the block sizes were communicated in Line 25, this can be done independently by each process. The algorithm returns either when the blocks are sufficiently balanced or when a maximum number of balancing iterations is reached. In our experiments with $\epsilon \in \{0.03, 0.05\}$, balance was always achieved when allowing a sufficient number of balance and movement iterations.

Algorithm 15 shows our main k-means algorithm. We first sort and redistribute the points according to a space-filling curve to improve spatial locality (Lines 4 – 6), place initial centers in equal distances on the sorted points (Line 7) and initialize data structures (Lines 8 and 9). Deriving initial centers from the space-filling curve in this way yields a beneficial geometric spread.

The main loop consists of calling Algorithm 14 until the centers converge sufficiently or a

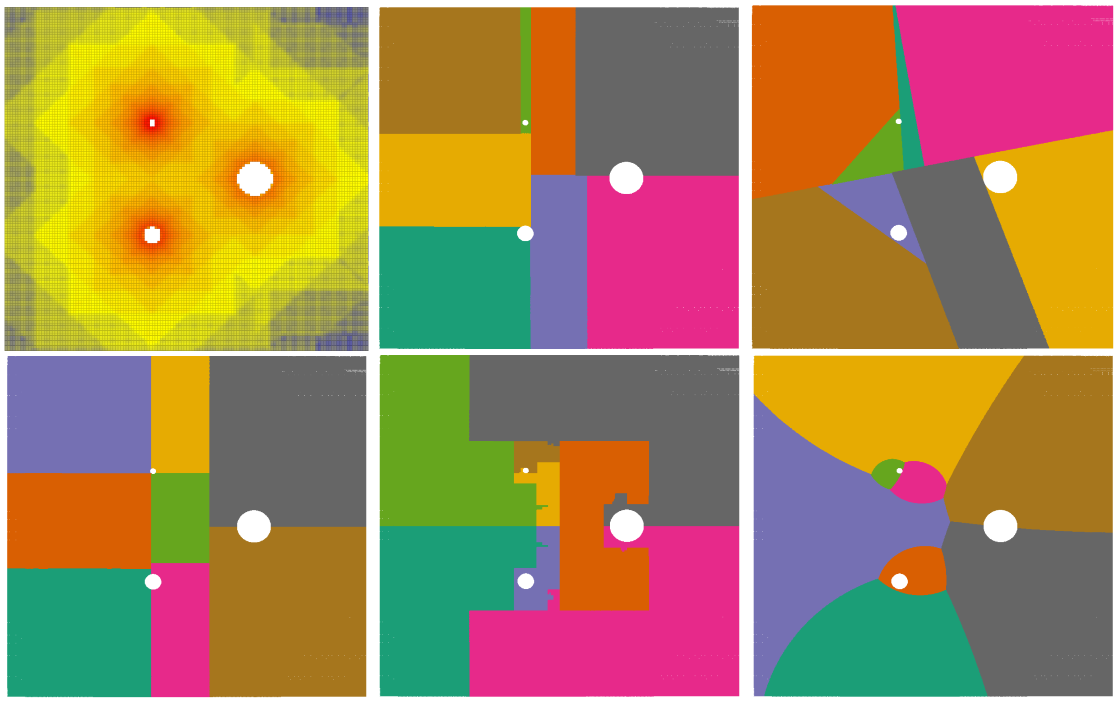


Figure 4.11: Partition of hugetric-0000 in 8 blocks with different tools. From left to right, the pictures show the input and the results of RCB, RIB, MultiJagged, zoltanSFC and Geographer. The varying density of the mesh is indicated by the heat map in the first picture, it ranges over three orders of magnitude.

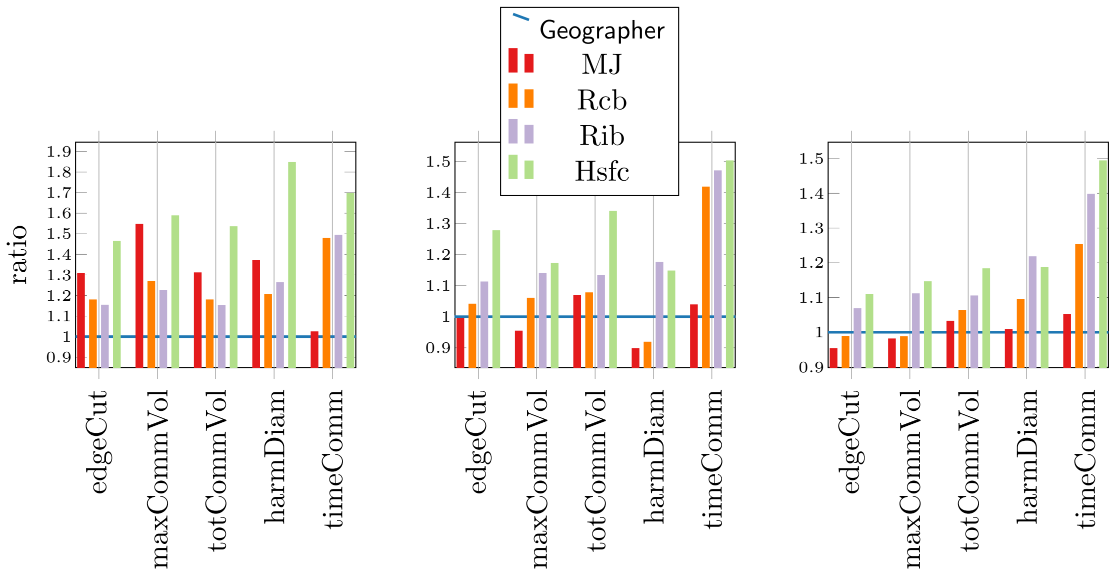
maximum number of iterations is reached. New cluster centers are the weighted average of the assigned points; this can be computed efficiently with a local sum and two global MPI vector sum operations (Line 13). Apart from the initial setup, all communication steps are global reduction operations, for which efficient implementations exist. Lines needing communication are marked in blue. Note that the number of blocks the point set is partitioned into is independent from the number of parallel processes that are used to do it.

One optimization omitted from the pseudocode for the sake of brevity is *random initialization*. In the initial phases of k-means, cluster centers and influence values change rapidly and the geometric bounds are of little help. However, during these wild fluctuations not as much precision is required as in the later fine-tuning stages. To exploit this effect, each process permutes its local points randomly and then picks the first 100 as initial sample. After each round with center movement, the sample size is doubled. These $\lceil \log_2(n_{\text{local}}/100) \rceil$ initialization rounds take about as much time as one round with the full point set, but move centers much closer to their final positions. Starting with only a randomly sampled subset of points does not impact the quality noticeably.

4.4.2 Experimental Evaluation

Implementation

Our graph partitioner Geographer is implemented in C++11 and parallelized with MPI. To increase portability and usability, we develop the partitioner within LAMA [28], a portable



(a) DIMACS graphs (2D) (b) Climate graphs (2.5D) (c) Alya and Delaunay (3D)

Figure 4.12: Average ratio of between (geometric mean except diameter, see text) of the evaluation metrics for all tools. Baseline: Geographer.

framework for distributed linear algebra and other numerical applications. LAMA provides high-level data structures and communication routines for distributed memory, abstracting away the specifics of the MPI communicator and also supporting other parallelization mechanisms. In the course of this work, we contributed several optimized communication routines to LAMA that are used by our partitioner.

Our implementation is publicly available at `https://github.com/kit-parco/geographer`.

Experimental Settings

Machine

We perform our experiments on Thin Phase 1 nodes of the SuperMUC petascale system at the Leibniz Supercomputing Centre of the Bavarian Academy of Sciences and Humanities. Each node is equipped with 32 GB RAM and two Intel Xeon E5-2680 processors (Sandy Bridge) at 2.7 GHz and 8 cores per processor. In our experiments, we allocate one MPI process to each core. Both our code and the evaluated competitors are compiled with GCC 5.4 and parallelized with IBM MPI 1.4.

Compared Partitioners

We compare our new partitioner Geographer with several established geometric partitioning implementations from the ParMetis (4.0.3) and Zoltan 2 (part of the Trilinos 12.10 Project) toolboxes. The geometric partitioner within the ParMetis package uses space-filling curves, the Zoltan package contains implementations of Recursive Coordinate Bisection (RCB), Recursive Inertial Bisection (RIB), space-filling curves (zoltanSFC) and the MultiJagged algorithm, mentioned in Section 4.2. Since the ParMetis version of space-filling curves is dominated by the space-filling curves in the Zoltan package, we omit it from the detailed presentation.

While we see geometric partitioners as our main competitors, we also compare with the graph-based partitioners Mt-KaHiP and ParHIP [111] and the corresponding graph datasets. Both are variants of the sequential partitioner KaHIP, which is known for its high solution quality [142]. It is expected that both tools will achieve a better quality at the expense of a higher running time. For ParHIP, we select the “fastmesh” preconfiguration for its appropriate tradeoff between running time and edge cut for our scenario.

Test Data

We evaluate the partitioners on a variety of datasets: a collection of benchmark meshes from the 10th DIMACS implementation challenge [14], 2.5D meshes with vertex weights from climate simulations [41], 3D meshes from the PRACE Unified European Applications Benchmark Suite (UEABS) [134] and Delaunay triangulations[15] of random points in two and three dimensions. More precisely, the graphs `hugetrace`, `hugetric` and `hugebubbles` are 2D adaptively refined triangular meshes from the benchmark generator created by Marquardt and Schamberger [106]; they represent synthetic numerical simulations and have approx. 5M to 20M vertices. `333SP`, `AS365`, `M6`, `NACA0015` and `NLR` are 2D finite element triangular meshes from approx. 3.5M vertices and 11M edges up to approx. 21M vertices and 31M edges. `rgg_n` are 2D random geometric graphs with 2^n vertices for $n = 20, \ldots, 24$. All these graphs come from the 10th DIMACS Implementation Challenge [14]. The graphs in the `DelaunayX` series are Delaunay triangulations of `X` random 2D points in the unit square [73]. The smallest graph in the series has 8M vertices and approximately 24M edges; the largest one has 2B vertices and approximately 6B edges. We also generated five 3D `Delaunay` triangulations from approx. 1M to 16M vertices using the generator of Funke et al. [58]. The graphs `alyaTestCaseA` with 9.9 million vertices and `alyaTestCaseB` with 30.9 million vertices (representing the respiratory system) are from the PRACE benchmark suite [134].

Metrics

We evaluate the generated partitions with respect to several metrics. In addition to the classical edge cut, we report both $\max\text{comm}$, the maximum communication volume and $\sum\text{comm}$, the total communication volume.

To evaluate the effect of shape optimization, we also measure the graph diameter for each block. Since a precise computation of the diameter scales at least quadratically with the number of vertices, we instead use a lower bound generated by executing the first 3 rounds of the iFUB algorithm by Crescenzi et al. [40]. This lower bound is a 2-approximation of the exact diameter, but often already tight.

To estimate the impact on the communication patterns of a parallel application, we measure the communication time of a parallel SpMV benchmark.

[15]We thank Christian Schulz, who kindly provided the generator [73] for the `Delaunay` graphs.

Other Parameters

In all experiments, we set the number of blocks k to the number of processes p and the maximum imbalance ϵ to 3%, which was respected by all tools. Reported values are averaged over 5 runs to account for random fluctuations.

Results

For a brief first visual impression of the results of Geographer, RCB, RIB, MultiJagged, and zoltanSFC, see Fig. 4.11. Recursive coordinate and inertial bisection produce thin, long blocks, MultiJagged produces rectangles with a better aspect ratio, zoltanSFC's blocks have wrinkled boundaries, balanced k-means produces curved blocks.

Both Mt-KaHiP and ParHIP achieved a significantly improved quality on small graphs, but ran out of time (25 minutes) or memory (256GB) for graphs with more than ≈30 million vertices. As we target mainly larger graphs, we omit these tools from most of the detailed comparisons.

A collection of raw measurements are available at the corresponding data archive at DOI 10.5445/IR/1000095237 on the KITOpenData repository.

Quality

Figure 4.12 compares the partitions yielded by the tested tools under the metrics edgeCut, maximum and total communication volume and diameter. For easier presentation, we report the relative value compared to Geographer and aggregate the results by graph class using the geometric mean.

In some cases, blocks are disconnected and thus have an infinite diameter. To avoid a potentially infinite mean diameter, we use the harmonic instead of the geometric mean to aggregate the diameter over all blocks.

The first instance class consists of the 2D geometric benchmark meshes from the DIMACS challenge, the second consists of the 2.5D graphs from climate simulations. The third class consists of the alya test case and Delaunay triangulations in the unit cube, all 3D meshes.

In all graph classes, Geographer produces on average the partition with the lowest total communication volume. The advantage is most pronounced on the 2D geometric meshes from the DIMACS collection, but visible also in other classes. This does not mean that Geographer achieves *always* the best results, as these are aggregated values.

The performance as measured by the edge cut differs: On the DIMACS graphs, Geographer is leading with 15% difference, on the 2.5D and 3D graphs MultiJagged has an advantage of 0.5% and 4%, respectively. Similar developments are visible also for other metrics.

The empirical average communication time within the SpMV benchmarks (timeComm in Figure 4.12) correlates little with the more established measures. Results fluctuate, but Geographer has on average the smallest SpMV communication time.

Note that the behavior of balanced k-means is *stable*: If it performs worse on some class, then not by much. None of the evaluated competitors clearly dominates: While MultiJagged, for example, has a lower mean edge cut (5%) on 3D graphs, its performance on the DIMACS graphs is clearly worse, with a 30% higher edge cut.

Detailed results for individual graphs are shown in Tables 4.1 and 4.2.

Table 4.3 shows a comparison against the graph-based partitioners Mt-KaHiP and ParHIP. Mt-KaHiP is a shared-memory partitioner, thus the size of instances was limited to those fitting into 256GB of memory. On average, Mt-KaHiP has a 15% better cut than the best geometric partitioner and a running time that is higher by two orders of magnitude. Surprisingly, the lower edge cut does not lead to shorter communication in the SpMV benchmark. The average edge cut of ParHIP is similar to the best geometric partitioner, with a difference of under one percent. The running time, though, is about one order of magnitude higher.

Statistical Analysis

Some of the compared algorithms use randomness, and even the deterministic ones may be sensitive to hardware effects and peculiarities of the input. To determine whether the lower total communication volume and shorter SpMV communication time are possibly just the result of random fluctuations, we model the performance of Geographer and MultiJagged as dependent random variables with input graphs $g_1, g_2 \dots g_n$ and a multiplicative Gaussian noise term ϵ:

$$\log_2(\mathrm{commVolG(g_i)}) = \alpha + \log_2(\mathrm{commVolMJ(g_i)}) + \epsilon \tag{4.8}$$

$$\log_2(\mathrm{commSpMVG(g_i)}) = \beta + \log_2(\mathrm{commSpMVMJ(g_i)}) + \epsilon \tag{4.9}$$

This is equivalent to drawing from parametrized normal distributions:

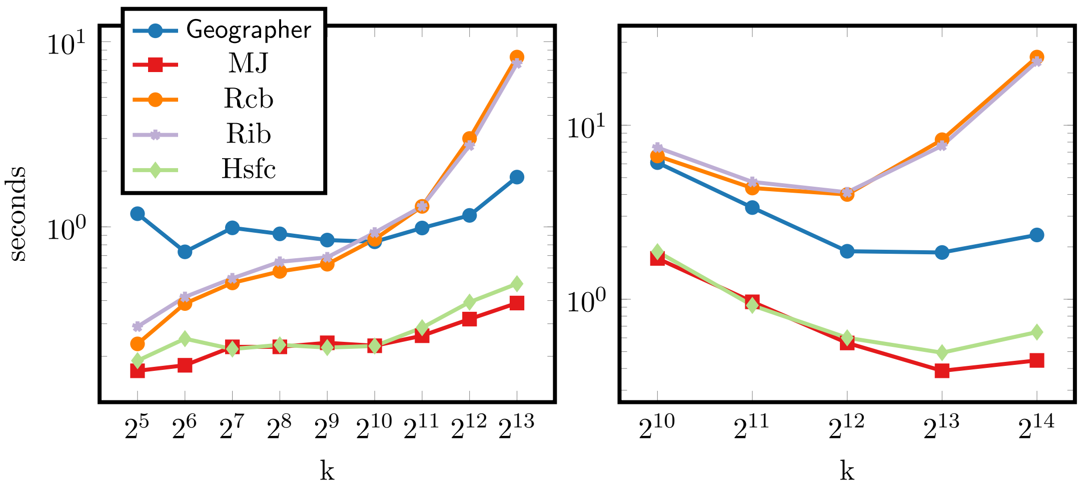
$$\log_2(\mathrm{commVolG(g_i)}) \sim \mathcal{N}(\alpha + \log_2(\mathrm{commVolMJ(g_i)}), \sigma_\epsilon) \tag{4.10}$$

$$\log_2(\mathrm{commSpMVG(g_i)}) \sim \mathcal{N}(\beta + \log_2(\mathrm{commSpMVMJ(g_i)}), \sigma_\epsilon) \tag{4.11}$$

Since the results are log-transformed, the additive Gaussian noise term models a multiplicative error. Thus, we model the communication volume of a graph g_i partitioned by Geographer to follow a parametrized log-normal distribution, with the mean depending on the result of MultiJagged for the same instance.

We then perform a Bayesian linear regression on our entire graph set to get estimates for α and β. For both α and β, we use weakly informative normal priors with mean 0 and standard deviation 10. This represents the prior belief that the performance of algorithms is equivalent. For the variance σ_ϵ of the likelihood function, we use a half-normal distribution with standard deviation 1.

We approximate the posterior distributions of α, β and σ using Markov Chain Monte Carlo sampling provided by the PyMC3 [140] package. Stable estimates are reached after 500



(a) Weak scaling for `DelaunayX` graph series.

(b) Strong scaling for `Delaunay2B` graph

Figure 4.13: Scaling results on `DelaunayX` regarding (a) strong and (b) weak scaling. The number of blocks k is kept equal to the number of processes.

tuning and 10000 sampling iterations. In the resulting posterior distributions, the 0.95-credible intervals for α and β are $[-0.308, -0.205]$ and $[-0.21469671, -0.062]$ respectively, implying that with 95% probability, the true average performance ratios of the algorithms are between 0.80 and 0.87 (for the total communication volume) and between 0.86 and 0.96 (for the SpMV communication time). Note that this does not mean that Geographer always yields solutions with a 13–20% smaller total communication volume than MultiJagged. Instead, after accounting for random noise in the measurements, the *true average improvement* in total communication volume is 13–20% with probability at least 95%. The same holds for the 4–14% improvement in SpMV communication time.

As an alternative to Bayesian analysis, we also offer a Wilcoxon [169] test on the paired results of MultiJagged and Geographer. The null hypothesis of both algorithms being from the same distribution yields p-values of 1.27×10^{-17} for the total communication volume and 0.007 for the SpMV communication time. Strictly speaking, evaluating several metrics is equivalent to testing multiple hypothesis in the null hypothesis statistical testing (NHST) framework, requiring a correction to reduce the risk of false positives. However, our values are far lower than commonly used significance thresholds, thus even an overly conservative Bonferroni [49] correction would not change this conclusion

Weak Scaling

Fig. 4.13a shows a direct comparison of weak scaling performance on the `DelauanyX` graph series. We start with 32 processes (p) and blocks (k) on 8 million vertices and repeatedly double both until we reach 8 192 processes on 2 billion vertices, keeping the ratio fixed at approximately 250 000 vertices per process. Geographer exhibits similar behavior as MultiJagged and zoltanSFC: they scale almost perfectly up to 1024 PEs, then increase roughly by a factor of two over the next three doublings. The recursive methods RIB and RCB show an immediate increase in running time with larger inputs and scale especially poorly on more than 1 024 processes, more than doubling the running time for each doubling of input and process count. A comparison of running times on all graphs can be seen in Fig. 4.14. The scaling behavior of the different tools follows similar trends as on the `DelaunayX` series. Fitted trend lines show a better weak scaling behavior of Geographer than on `Delaunay` graphs alone, which may also be an artifact of the higher number of smaller graphs in our collection.

Strong Scaling

We perform strong scaling experiments (see Fig. 4.13b) with the largest graph at our disposal, `Delaunay2B`. With 2 billion vertices, it is small enough to fit into the memory of 1 024 processing elements but also sufficiently large to partition it into 16 384 blocks. Note that our experiments are not, strictly speaking, strong scaling, as we increase the number of blocks along with the number of processors.

Similarly to the weak scaling results, Geographer, MultiJagged and zoltanSFC have similar scaling behavior: almost perfect scalability for up to 4 096 PEs (MultiJagged also for 8 192). RCB and RIB start with the slowest running times, around 6.5 seconds for $k = 1024$ and climb to 23 seconds for $k = 16384$ showing poor scalability. For all tools, the running time increases from 8 192 to 16 384 processes; we attribute this to the SuperMUC architecture: an island in SuperMUC contains 8 192 cores and communication is more expensive across islands.

Components

The main parts of Geographer contributing to the running time are the initial partitioning with a Hilbert curve, the redistribution of coordinates according to this initial partition and finally the balanced k-means itself. As the number of processes increases, the relative share of these components changes: For small instances, the computation of Hilbert indices and the balanced k-means iterations constitute a majority of the time, while for higher number of processes, the redistribution step dominates. For example, when partitioning `Delaunay2B` with 1024 processes, calculating the partition using a space filling curve takes 20%, data (graph, coordinates and vertex weights vectors) redistribution 32% and k-means takes 47% of the total running time. For the same graph and 16384 processes, the space filling curve step requires 10%, data redistribution 46% and k-means 42% of the total running time.

Local Refinement

For the case that graph adjacency data is available and quality is of higher importance, we also offer an implementation of the multi-level heuristic together with parallel local refinement based on the algorithm by Fiduccia-Mattheyses [55]. It is used to subsequently improve the partition computed by the k-means algorithm. On average, this graph-based refinement takes 10-20 seconds and decreases the cut by a further 10%. Performance is less predictable, though, which is why we did not include it in our main experiments.

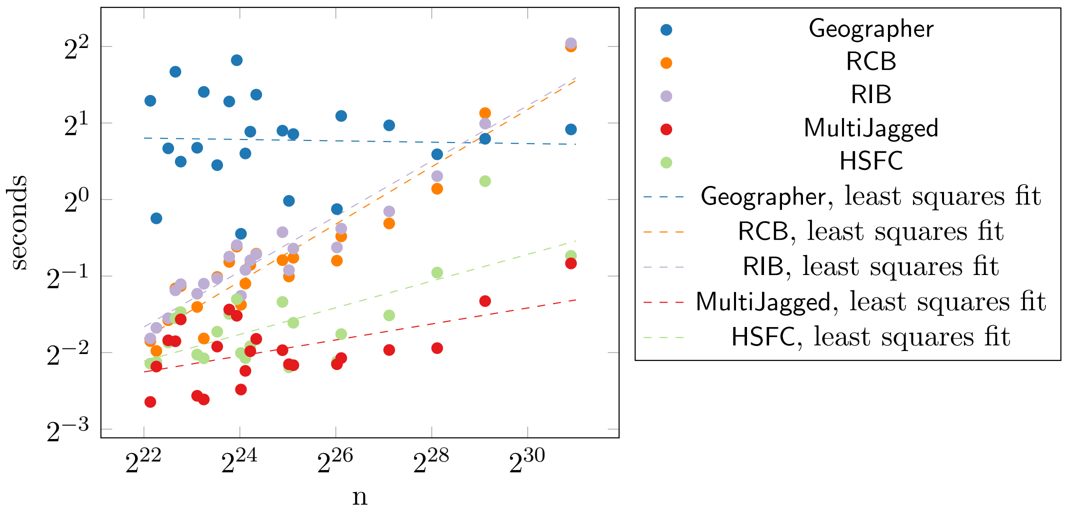


Figure 4.14: Comparison of running times with different tools, graphs and numbers of processes. Each dot represents the running time of one tool on one graph. We select the number of blocks k (and processes k) as a power of two and aim for 250 000 points per block. For example, a graph with 4.5 million vertices is partitioned into 16 blocks, since $4.5 \cdot 10^6/16 = 281\,250$ is closer to 250000 than $4.5 \cdot 10^6/32 = 140\,625$.

4.4.3 Conclusion

We designed and implemented Geographer, a balanced, scalable version of k-means for partitioning geometric meshes. Combined with space-filling curves for initialization, it scales to thousands of processors and billions of vertices, partitioning them in a matter of seconds.

An evaluation on a wide range of input meshes shows that the total communication volume and resulting SpMV communication time of the resulting partitions is on average 5–15% better than those of state-of-the-art competitors. This difference is most pronounced on meshes from the DIMACS benchmark collection, but also measurable on graphs from climate simulations and 3D meshes. Concerning the edge cut, another common metric to evaluate graph partitioners, MultiJagged also performs well, giving the best results on 3D meshes. No partitioner dominates on all point sets.

Graph-based partitioners which focus on quality, as for example Mt-KaHiP and ParHIP, do achieve lower edge cuts and communication volumes on small graphs, but do not scale to the large graph sizes which are our main focus.

Future work will be concerned with further improving quality and scalability, in particular for 3D meshes. A faster redistribution routine, necessary to achieve scalability for a higher number of processes, is also of independent interest.

Our implementation also includes a graph phase using the multi-level heuristic, local refinement and diffusion steps. First experiments indicate a promising effect of not following the classical multi-level paradigm with an initial partitioning on the coarsest graph, but instead performing a geometric partitioning step *before* the coarsening. Further exploring this effect is also part of future work.

Finally, the requirement of having geometric coordinates could be lifted by generating artificial coordinates with a sufficiently fast graph drawing or embedding algorithm [88]. So far, these algorithms tend to not scale sufficiently to be useful for the range of parallelism we target, addressing this is thus also part of future work.

Algorithm 14: AssignAndBalance

```
Input: centers $C$, local points $P_{\text{local}}$, vertex weights $W$, influence, previous
       assignments $A_{\text{old}}$, ub, lb, $\epsilon$
Output: assignments $A$, new influence, bounds ub, lb
1  bb $\leftarrow$ bounding box around local points $P_{\text{local}}$;
2  for center $c \in C$ do
3      distToBb[$c$] $\leftarrow$ maxDist(bb,$c$)/influence[$c$];
4      localBlockSizes[$c$] $\leftarrow$ 0;
5  sort centers $C$ by distToBb;
6  for $i \in \{0, ..., \text{maxBalanceIter}\}$ do
7      for $p \in P_{local}$ do
8          if $ub[p] < lb[p]$ then
9              $A[p] \leftarrow A_{\text{old}}[p]$;
10         else
11             bestValue, secondBestValue $\leftarrow \infty$;
12             for $c \in C$ do
13                 if $distToBb[c] > secondBestValue$ then
14                     break;
15                 effDist $=$ dist$(c, p)$/influence[$c$];
16                 if $effDist < bestValue$ then
17                     $A[p] \leftarrow c$;
18                     secondBestValue $\leftarrow$ bestValue;
19                     bestValue $\leftarrow$ effDist;
20                 else if $effDist < secondBestValue$ then
21                     secondBestValue $\leftarrow$ effDist;
22             ub[$p$] $\leftarrow$ bestValue;
23             lb[$p$] $\leftarrow$ secondBestValue;
24         localBlockSizes[$A[p]$]$+=W[p]$;
25     globalSizes $\leftarrow$ globalSumVector(localBlockSizes);
26     if $imbalance(globalSizes) < \epsilon$ then
27         return $A, I, ub, lb$
28     for $c \in C$ do
29         influence($c$) $\leftarrow$ adaptInfluence(influence($c$), globalSizes[$c$]);
                                                              /* Eq. (4.5) */
30     update ub;
31     update lb;
32 return $A, I, ub, lb$
```

Algorithm 15: BalancedKMeans

```
Input: points $P$, number of blocks $k$, maximum imbalance $\epsilon$, deltaThreshold
Output: assignments $A$
1  $n \leftarrow \#P$;
2  #proc $\leftarrow$ number of processors;
3  $r \leftarrow$ rank of processor;
4  sfcIndex[$p$] $\leftarrow$ index of $p$ on space-filling curve $\forall p \in P$ ;
5  sortedPoints $\leftarrow$ sortGlobal($P$, key=sfcIndex);
6  $P_{\text{local}} \leftarrow$ sortedPoints[$r \cdot n/\#\text{proc}, ..., (r+1) \cdot n/\#\text{proc}$];
7  $C[i] \leftarrow$ sortedPoints[$i \cdot n/k + n/2k$] for $i \in \{0, ...k-1\}$;
8  $I[c] \leftarrow 1$ for $c \in C$;
9  ub[$p$] $\leftarrow$ inf, lb[$p$] $= 0$ $\forall p \in P_{\text{local}}$;
10 for $i \in \{0, ..., \text{maxIter}\}$ do
11     $A$, $I$, ub, lb $\leftarrow$ AssignAndBalance($C$, $P_{\text{local}}$, $I$, $A$, ub, lb, $\epsilon$);
12     $C'_{\text{local}}[c] \leftarrow$ mean of $p \in P_{\text{local}}$ with $A[p] = c$;
13     $C' \leftarrow$ globalWeightedMeanVector($C'_{\text{local}}[c]$);
14     if $\max \delta(C, C') <$ *deltaThreshold* then
15         return $A$
16     $C \leftarrow C'$;
17     adapt bounds ub, lb with Eq. (4.6) and (4.7);
18 return $A$
```

Table 4.1: Comparison of results for large graphs for $k = p = 1024$. Best values are marked in bold. Times for SpMV communications are given in microseconds.

graph name	tool	time(s)	cut	maxComm	Σ comm	diam	timeSpMVComm
`alyaTestCaseB`	Geographer	0.35	5 823 055	6 508	**5 403 716**	63	198.77
n = 30 959 144	HSFC	0.04	6 613 710	8 275	6 802 889	83	361.43
	MultiJagged	**0.02**	**5 364 660**	**6 062**	5 482 086	**62**	**198.04**
	RCB	0.10	6 188 060	6 526	5 825 470	79	315.97
`delaunay250M`	Geographer	0.83	**2 037 960**	**2 183**	**2 033 939**	**457**	**34.61**
n = 250 000 000	HSFC	**0.23**	2 356 510	2 918	2 349 673	570	136.38
	MultiJagged	0.23	2 118 810	2 214	2 114 657	494	74.32
	RCB	0.85	2 118 220	2 211	2 113 916	491	126.76
`delaunay2B`	Geographer	6.09	**5 771 443**	**6 136**	**5 754 364**	-	-
n = 2 000 000 000	HSFC	1.87	6 335 780	6 807	6 314 790	-	-
	MultiJagged	**1.72**	5 994 110	6 164	5 976 573	-	-
	RCB	6.66	5 995 910	6 137	5 978 340	-	-
`fesom-jigsaw`	Geographer	0.77	33 135 424	1 539	680 638	392	**37.69**
n = 14 349 744	HSFC	**0.02**	33 749 900	1 500	735 964	339	58.12
	MultiJagged	0.02	**27 472 100**	**1 111**	**641 574**	320	43.53
	RCB	0.08	30 447 900	1 524	642 714	**320**	67.78
`refinedtrace-06`	Geographer	1.50	**813 450**	**1 596**	**1 380 977**	**1 052**	**40.95**
n = 289 383 634	HSFC	0.52	1 044 170	2 856	1 909 341	1 834	119.55
	MultiJagged	**0.26**	1 063 020	3 790	1 801 389	1 607	71.64
	RCB	1.10	930 479	3 864	1 552 315	1 335	108.35
`refinedtrace-07`	Geographer	3.54	**1 144 423**	**2 205**	**1 948 602**	**1 483**	**51.98**
n = 578 551 252	HSFC	0.81	1 467 320	4 054	2 689 631	2 609	156.08
	MultiJagged	**0.57**	1 521 740	4 644	2 551 801	2 285	89.13
	RCB	2.04	1 314 980	6 988	2 189 002	1 898	146.98

Table 4.2: Comparison for smaller graphs and $k = p = 64$. Best values marked in bold. Times for SpMV communications are given in microseconds.

graph name	tool	time(s)	cut	maxComm	Σ comm	diam	timeSpMVComm
333SP	Geographer	1.49	**32 170**	**1 203**	**32 306**	**205**	81.66
n = 3 712 815	HSFC	**0.05**	83 162	2 341	83 077	845	61.70
	MultiJagged	0.08	90 650	3 899	90 793	517	**33.96**
	RCB	0.08	59 558	2 291	59 700	357	45.76
AS365	Geographer	0.47	**51 666**	1 114	**51 832**	**274**	29.65
n = 3 799 275	HSFC	0.06	87 274	1 863	87 355	487	64.69
	MultiJagged	**0.05**	64 312	1 796	64 480	364	**21.74**
	RCB	0.10	55 880	**1 075**	56 040	313	52.14
M6	Geographer	0.29	**51 971**	1 145	**52 131**	**259**	27.71
n = 3 501 776	HSFC	0.04	82 905	1 749	82 938	416	64.17
	MultiJagged	**0.04**	58 270	1 080	58 430	310	**22.89**
	RCB	0.09	56 867	**1 052**	57 027	301	52.29
NACA0015	Geographer	0.10	**27 841**	**549**	**27 997**	**152**	24.26
n = 1 039 183	HSFC	**0.01**	56 902	1 367	56 897	370	51.45
	MultiJagged	0.01	40 314	759	40 476	244	**19.41**
	RCB	0.03	29 484	603	29 638	162	31.60
NLR	Geographer	0.37	**56 805**	**1 073**	**56 969**	**281**	28.93
n = 4 163 763	HSFC	**0.05**	85 740	1 704	85 803	377	63.83
	MultiJagged	0.06	61 034	1 102	61 193	306	**23.68**
	RCB	0.11	60 703	1 170	60 863	306	47.99
alyaTestCaseA	Geographer	0.96	894 845	18 341	841 593	113	**214.97**
n = 9 938 375	HSFC	0.12	988 168	21 325	1 001 416	132	356.05
	MultiJagged	**0.08**	**839 540**	17 798	839 437	**107**	250.60
	RCB	0.17	847 188	**17 726**	**839 377**	108	274.82
alyaTestCaseB	Geographer	1.86	1 869 558	38 203	**1 774 168**	163	**329.37**
n = 30 959 144	HSFC	0.39	1 857 670	39 351	1 880 792	171	589.69
	MultiJagged	**0.25**	**1 772 580**	**37 435**	1 785 528	**156**	511.83
	RCB	0.57	1 784 790	37 447	1 785 598	167	574.01
delaunay017M	Geographer	0.73	**122 875**	**2 235**	**122 634**	**476**	**30.13**
n = 17 000 000	HSFC	0.25	131 407	2 428	131 012	549	103.83
	MultiJagged	**0.18**	125 088	2 302	124 823	489	38.32
	RCB	0.39	124 789	2 263	124 545	499	106.32
fesom-f2glo04	Geographer	0.45	**4 758 930**	**1 590**	**66 472**	**547**	**29.48**
n = 5 945 730	HSFC	**0.07**	7 677 820	2 372	108 910	957	45.74
	MultiJagged	0.08	5 866 330	2 263	85 789	740	31.85
	RCB	0.14	5 596 840	1 755	79 220	631	42.22
fesom-fron	Geographer	0.51	**3 870 505**	1 565	**61 576**	731	33.20
n = 5 007 727	HSFC	**0.06**	5 272 540	1 993	90 305	1 067	32.05
	MultiJagged	0.06	4 214 460	**1 485**	70 794	788	**26.91**
	RCB	0.09	4 365 920	1 846	74 330	**706**	34.41
fesom-jigsaw	Geographer	1.09	8 444 620	4 225	166 006	6 306	42.33
n = 14 349 744	HSFC	0.18	8 224 760	4 269	177 071	2 159	121.37
	MultiJagged	**0.18**	**6 185 490**	**3 078**	**142 214**	**1 843**	**41.01**
	RCB	0.39	8 637 620	3 980	173 583	2 800	123.33
hugebubbles-20	Geographer	2.58	**47 763**	**1 636**	**81 556**	**1 048**	**31.00**
n = 21 198 119	HSFC	**0.28**	63 053	2 561	118 785	2 002	75.07
	MultiJagged	0.28	60 958	2 430	105 714	1 453	31.91
	RCB	0.61	57 482	2 003	98 404	1 299	78.98
hugetrace-20	Geographer	1.48	**43 522**	**1 471**	**74 122**	**948**	31.02
n = 16 002 413	HSFC	0.20	50 800	2 081	98 367	1 585	55.83
	MultiJagged	**0.19**	50 493	1 872	86 699	1 117	**27.60**
	RCB	0.39	50 851	1 911	84 526	1 084	39.04
hugetric-20	Geographer	0.82	**29 298**	**972**	**49 899**	**615**	**28.78**
n = 7 122 792	HSFC	0.16	39 373	1 658	72 388	1 210	57.25
	MultiJagged	**0.09**	39 382	2 203	67 949	899	30.33
	RCB	0.18	35 512	1 364	60 706	810	54.82
rdg-3d	Geographer	0.32	**1 481 602**	12 683	**761 610**	**45**	**251.51**
n = 4 194 304	HSFC	**0.04**	1 596 600	13 391	821 701	50	328.72
	MultiJagged	0.06	1 537 820	**12 552**	790 899	48	278.77
	RCB	0.09	1 537 980	12 558	791 086	48	297.71
rgg-24	Geographer	0.18	**185 371**	**1 791**	**94 132**	**273**	27.63
n = 4 194 304	HSFC	0.05	207 415	2 029	104 841	311	28.88
	MultiJagged	**0.04**	191 502	1 856	96 863	290	**27.05**
	RCB	0.08	189 795	1 864	96 539	303	28.13

Table 4.3: Comparison against Mt-KaHiP and ParHIP for smaller graphs and $k = p = 64$. Best values marked in bold. As Mt-KaHiP only supports shared memory, it was limited to 16 threads. Where values for ParHIP are missing, it crashed or did not finish in the alloted 25 minutes. Times for SpMV communications are given in microseconds.

graph name	tool	time(s)	cut	maxComm	Σ comm	diam	timeSpMVComm
`333SP`	Geographer	1.49	**32 170**	1 203	32 306	**205**	81.66
n = 3 712 815	HSFC	**0.05**	83 162	2 341	83 077	845	61.70
m = 11 108 633	MultiJagged	0.08	90 650	3 899	90 793	517	**33.96**
	RCB	0.08	59 558	2 291	59 700	357	45.76
	Mt-KaHiP	16.10	33 499	**972**	**31 883**	245	55.21
`AS365`	Geographer	0.47	**51 666**	1 114	51 832	274	29.65
n = 3 799 275	HSFC	0.05	87 274	1 863	87 355	487	64.69
m=11 368 076	MultiJagged	**0.05**	64 312	1 796	64 480	364	**21.74**
	RCB	0.10	55 880	1 075	56 040	313	52.14
	Mt-KaHiP	20.10	52 343	**1 045**	**47 988**	**268**	34.80
	ParHIP	22.40	55 733	1 183	55 849	-	-
`M6`	Geographer	0.29	**51 971**	1 145	52 131	**259**	27.71
n = 3 501 776	HSFC	0.04	82 905	1 749	82 938	416	64.17
m=10 501 936	MultiJagged	**0.04**	58 270	1 080	58 430	310	**22.89**
	RCB	0.09	56 867	1 052	57 027	301	52.29
	Mt-KaHiP	17.40	52 186	**944**	**48 841**	280	31.74
	ParHIP	24.25	55 939	1 164	56 099	-	-
`NACA0015`	Geographer	0.11	**27 841**	549	27 997	**152**	24.26
n = 1 039 183	HSFC	**0.01**	56 902	1 367	56 897	370	51.45
m = 3 114 818	MultiJagged	0.01	40 314	759	40 476	244	**19.41**
	RCB	0.03	29 484	603	29 638	162	31.60
	Mt-KaHiP	6.90	28 284	**513**	**25 764**	161	26.40
`NLR`	Geographer	0.37	**56 805**	1 073	56 969	**281**	28.93
n = 4 163 763	HSFC	**0.06**	85 740	1 704	85 803	377	63.83
m=12 487 976	MultiJagged	0.06	61 034	1 102	61 193	306	**23.68**
	RCB	0.11	60 703	1 170	60 863	306	47.99
	Mt-KaHiP	9.50	58 107	**1 024**	**53 019**	304	40.02
	ParHIP	25.06	61 241	1 267	61 402	-	-
`alyaTestCaseA`	Geographer	0.96	894 845	18 341	841 593	113	**214.97**
n = 9 938 375	HSFC	0.12	988 168	21 325	1 001 416	132	356.05
m = 39 338 978	MultiJagged	**0.08**	839 540	17 798	839 437	**107**	250.60
	RCB	0.17	847 188	**17 726**	**839 377**	108	274.82
	Mt-KaHiP	51.20	**801 765**	23 315	1 089 682	138	270.82
`hugebubbles-20`	Geographer	2.58	47 763	1 636	81 556	**1 048**	**31.00**
n = 21 198 119	HSFC	**0.28**	63 053	2 561	118 785	2 002	75.07
m =31 790 179	MultiJagged	0.28	60 958	2 430	105 714	1 453	31.91
	RCB	0.61	57 482	2 003	98 404	1 299	78.98
	Mt-KaHiP	19.90	**40 602**	**1 476**	**67 574**	1 166	34.19
	ParHIP	30.40	44 686	1 917	89 082	-	-
`hugetrace-20`	Geographer	1.48	43 522	1 471	74 122	**948**	31.02
n = 16 002 413	HSFC	0.20	50 800	2 081	98 367	1 585	55.83
m = 23 998 813	MultiJagged	**0.19**	50 493	1 872	86 699	1 117	**27.60**
	RCB	0.39	50 851	1 911	84 526	1 084	39.04
	Mt-KaHiP	24.90	**35 314**	**1 306**	**59 702**	1 012	37.24
	ParHIP	31.00	39 303	1 576	78 373	-	-
`hugetric-20`	Geographer	0.82	29 298	972	49 899	**615**	**28.78**
n = 7 122 792	HSFC	0.16	39 373	1 658	72 388	1 210	57.25
m=8 733 523	MultiJagged	**0.09**	39 382	2 203	67 949	899	30.33
	RCB	0.18	35 512	1 364	60 706	810	54.82
	Mt-KaHiP	14.40	**23 889**	**843**	**40 496**	856	33.14
	ParHIP	23.80	26 531	1 122	52 789	-	-

5. Concluding Remarks

While graphs have been used to abstract away from geometry as early as Euler's problems in Königsberg, several graph problems can profit from considering the underlying geometry.

In Chapter 3, we presented several algorithms for the fast sampling of random hyperbolic graphs, a generative model for complex networks. Among these are the first subquadratic generation algorithms for both the threshold and general case, with a complexity of $\mathcal{O}((n^{3/2} + m) \log n)$ to generate a graph with n vertices and m edges. In later work, we reduced this to a near-linear time complexity, with $\mathcal{O}(n \log^2 n + m)$ for threshold graphs and general graphs with fixed temperature. The implementations reach a speedup of several orders of magnitude over the naive sampling algorithm, the main component of these improvements is the development of suitable geometric data structures. For the general case of non-threshold graphs, our implementation is still the fastest in practice. Our sampling algorithms also enable efficient dynamic graph updates and sampling in geometric datasets in Euclidean space.

In Chapter 4, we considered graph partitioning problems with additional constraints and geometric information. An application in quantum chemistry approximates electron densities on protein graphs by dividing the protein graphs into subsystems. For this division, we provide a dynamic programming algorithm and prove its optimality for a restricted problem scenario. Together with versions of the multi-level heuristics extended to incorporate additional constraints, we reduce the average approximation error by 13.5%. Our second application scenario are geometric meshes derived from numerical simulations. We adapt Lloyd's algorithm of the popular k-means problem to balanced block size; we also add geometric optimizations and seeding with space-filling curves, yielding fast convergence. A circular problem common in parallel graph partitioning is that to ensure a fast running time of the partitioner, first a good partition is needed. We address this by first sorting input points along a space-filling curve, a fast and rough partition to ensure some degree of spatial locality before our main partitioning phase. Our implementation scales to tens of thousands of processes and billions of vertices in distributed memory, partitioning them in seconds. The resulting partitioning solutions show a smaller communication volume than those of current geometric partitioners. While this result is useful for the targeted application, it does not generalize to higher dimensions as well as we had hoped. Graph partitioning has been active for decades, abstracting away communication patterns to graphs. Still, many applications have additional constraints or information and even on those problems which have been studied previously – partitioning geometric point sets – results can still be improved.

5.1 Future Work

Many directions for future work follow immediately from the contributions presented in this thesis.

Distributed Generation of Random Hyperbolic Graphs

With the expected linear generation algorithm of Bringmann et al., the theoretically optimal complexity has been reached, thus further developments will focus on empirical improvements. A combination of BandGen, the general model of random hyperbolic graphs and the request-centric approach by Funke et al. [58] would allow the efficient generation of random hyperbolic graphs in distributed memory, and for larger graph sizes.

Theoretical and Further Empirical Analysis of Balanced k-Means

While the balanced k-means converges in acceptable time in practice, this is not guaranteed – in fact, ensuring a balanced partition of weighted point sets is at least as difficult as the *subset sum* problem, which is known to be $\mathcal{NP}$-hard. A smoothed analysis of expected convergence time could close this gap and give additional insight on the algorithm's behavior. Other aspects to explore are partitionings and clusterings of other datasets. How does our algorithm work on street networks, for example, or higher-dimensional clustering problems? Employing hybrid parallelization with both OpenMP and MPI will likely further improve the running time.

Evaluation of Graph Partitioning Results

The experimental results concerning the graph-theoretic communication volume show that it is not a good predictor for the communication time in distributed sparse-matrix-vector multiplications. This is a common problem in the evaluation of graph partitioning algorithms, where graph-theoretic performance metrics are easily measured, but reduction in application running time is targeted. A set of simple yet robust performance models for the most important distributed application building blocks would make a step towards closing the gap.

Beyond Input Coordinates

Our graph partitioner uses coordinates to partition graphs, using geometry information to make decisions about topology. Earlier, we mentioned *graph drawing* and *embedding*, approaches to find geometric coordinates fitting a given topology. Can these usefully be combined? One could find coordinates for a graph lacking them and then use them to partition it. In reverse, one could partition a graph to draw it better, reducing dependencies between coordinates. Which of these directions are fruitful is a purely empirical question. In the case of creating artificial coordinates to then use geometric partitioning methods, no new information is gained by this detour through geometry, but it might work out in practice if the algorithms for the subproblems fit well together. A combination with centrality measures to only embed important nodes seems like a promising direction.

Bibliography

[1] Amine Abou-Rjeili and George Karypis. Multilevel algorithms for partitioning power-law graphs. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 10–pp. IEEE, 2006.

[2] Pankaj K Agarwal, Boris Aronov, Sariel Har-Peled, Jeff M Phillips, Ke Yi, and Wuzhou Zhang. Nearest neighbor searching under uncertainty II. In *Proceedings of the 32nd Symposium on Principles of Database Systems (PODS)*, pages 115–126. ACM, 2013.

[3] William Aiello, Fan Chung, and Linyuan Lu. A random graph model for massive graphs. In *Proceedings of the 32nd Symposium on Theory of Computing (STOC)*, pages 171–180. ACM, 2000.

[4] R. Albert and A.L. Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1):47, 2002.

[5] Rodrigo Aldecoa, Chiara Orsini, and Dmitri Krioukov. Hyperbolic graph generator. *Computer Physics Communications*, 196:492–496, 2015.

[6] L. A. N. Amaral, A. Scala, M. Barthélémy, and H. E. Stanley. Classes of small-world networks. *Proceedings of the National Academy of Sciences*, 97(21):11149–11152, 2000.

[7] Lars Arge and Kasper Green Larsen. I/O-efficient spatial data structures for range queries. In *SIGSPATIAL Special*, volume 4, pages 2–7. ACM, July 2012.

[8] R. Arratia and L. Gordon. Tutorial on large deviations for the binomial distribution. *Bulletin of Mathematical Biology*, 51(1):125–131, 1989.

[9] David Arthur, Bodo Manthey, and Heiko Röglin. Smoothed analysis of the k-means method. *Journal of the ACM (JACM)*, 58(5):19, 2011.

[10] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the 18th annual ACM-SIAM Symposium on Discrete algorithms (SODA)*, pages 1027–1035. SIAM, 2007.

[11] Franz Aurenhammer and Herbert Edelsbrunner. An optimal algorithm for constructing the weighted Voronoi diagram in the plane. *Pattern Recognition*, 17(2):251–257, 1984.

[12] Michael Axtmann, Armin Wiebigke, and Peter Sanders. Lightweight MPI communicators with applications to perfectly balanced quicksort. In *Proceedings of the 32nd International Parallel and Distributed Processing Symposium (IPDPS)*, pages 254–265. IEEE, 2018.

[13] Olivier Bachem, Mario Lucic, Hamed Hassani, and Andreas Krause. Fast and provably good seedings for k-means. In *Advances in Neural Information Processing Systems (NIPS)*, pages 55–63, 2016.

[14] David Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner, editors. *Proceedings of the 10th DIMACS Implementation Challenge*, Contemporary Mathematics. American Mathematical Society, 2012.

[15] Michael Bader. *Space-Filling Curves: An Introduction with Applications in Scientific Computing.* Springer, 2012.

[16] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[17] Albert-László Barabási and Eric Bonabeau. Scale-free networks. *Scientific American*, 288(5):60–69, 2003.

[18] Vladimir Batagelj and Ulrik Brandes. Efficient generation of large random networks. *Physical Review E*, 71(3):036113, 2005.

[19] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm. *Journal of Experimental Algorithmics (JEA)*, 15:2–3, 2010.

[20] M. J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, 36(5):570–580, May 1987.

[21] C.E. Bichot and P. Siarry. *Graph Partitioning.* ISTE. Wiley, 2013.

[22] Thomas Bläsius, Tobias Friedrich, Anton Krohmer, and Sören Laue. Efficient Embedding of Scale-Free Graphs in the Hyperbolic Plane. In *24th Annual European Symposium on Algorithms (ESA)*, pages 16:1–16:18, 2016.

[23] Michel Bode, Nikolaos Fountoulakis, and Tobias Müller. The probability of connectivity in a hyperbolic model of complex networks. *Random Structures & Algorithms*, 49(1):65–94, 2016.

[24] Michel Bode, Nikolaos Fountoulakis, and Tobias Müller. On the giant component of random hyperbolic graphs. In *7th European Conference on Combinatorics, Graph Theory and Applications*, volume 16, pages 425–429. Scuola Normale Superiore, 2013.

[25] János Bolyai. *The Science Absolute of Space: Independent of the Truth Or Falsity of Euclid's Axiom XI (which Can Never be Decided a Priori)...*, volume 3. Neomon, 1896.

[26] E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine. The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring. *Scientific Programming*, 20(2):129–150, 2012.

[27] Steffen Borgwardt, Andreas Brieden, and Peter Gritzmann. A balanced k-means algorithm for weighted point sets. *arXiv preprint arXiv:1308.4004*, 2013.

[28] Thomas Brandes, Eric Schricker, and Thomas Soddemann. *The LAMA Approach for Writing Portable Applications on Heterogeneous Architectures.* Springer, 2017.

[29] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. Maximizing modularity is hard. *arXiv preprint physics/0608255*, 2006.

[30] R. A. Bridges, J. P. Collins, E. M. Ferragut, J. A. Laska, and B. D. Sullivan. Multi-level anomaly detection on time-varying graph data. In *International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 579–583. IEEE, Aug 2015.

[31] Karl Bringmann, Ralph Keusch, and Johannes Lengler. Sampling geometric inhomogeneous random graphs in linear time. In *Proceedings of the 25th Annual European Symposium on Algorithms (ESA)*, volume 87 of *LIPIcs*, pages 20:1–20:15, 2017.

[32] Thang Nguyen Bui and Curt Jones. Finding good approximate vertex and edge partitions is np-hard. *Information Processing Letters*, 42(3):153–159, 1992.

[33] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. In *Algorithm Engineering - Selected Results and Surveys*, volume 9220 of *LNCS*, pages 117–158. Springer, 2016.

[34] F Cannavó and G Nunnari. Power laws in volcanic systems. In *19th European Conference on Mathematics for Industry*, page 346, 2016.

[35] Deepayan Chakrabarti and Christos Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Computing Surveys (CSUR)*, 38(1):2, 2006.

[36] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the 4th SIAM International Conference on Data Mining (SDM)*. SIAM, April 2004.

[37] A. Clauset, M.E.J. Newman, and C. Moore. Finding community structure in very large networks. *Physical Review E*, 70(6):66111, 2004.

[38] H. S. M. Coxeter. *The Trigonometry of Escher's Woodcut Circle Limit III*, chapter 29, pages 297–304. Springer, 2003.

[39] Christopher J. Cramer. *Essentials of Computational Chemistry.* Wiley, 2002.

[40] Pilu Crescenzi, Roberto Grossi, Michel Habib, Leonardo Lanzi, and Andrea Marino. On computing the diameter of real-world undirected graphs. *Theoretical Computer Science*, 514:84–95, 2013.

[41] S. Danilov, D. Sidorenko, Q. Wang, and T. Jung. The finite-volume sea ice–ocean model (fesom2). *Geoscientific Model Development*, 10(2):765–789, 2017.

[42] Daniel Delling, Daniel Fleischman, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. An exact combinatorial algorithm for minimum graph bisection. *Mathematical Programming*, 153(2):417–458, 2015.

[43] Mehmet Deveci, Sivasankaran Rajamanickam, Karen D. Devine, and Ümit V. Çatalyürek. Multi-jagged: A scalable parallel spatial partitioning algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 27:803–817, 2016.

[44] Inderjit S Dhillon and Dharmendra S Modha. A data-clustering algorithm on distributed memory multiprocessors. In *Large-Scale Parallel Data Mining*, pages 245–260. Springer, 2002.

[45] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw. Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM. *Parallel Computing*, 26:1555–1581, 2000.

[46] Chris Ding and Xiaofeng He. K-means clustering via principal component analysis. In *Proceedings of the 21st International Conference on Machine Learning (ICML)*, page 29. ACM, 2004.

[47] Sergei N Dorogovtsev and José FF Mendes. *Evolution of networks: From biological nets to the Internet and WWW.* Oxford University Press, 2003.

[48] Jonathan Drake and Greg Hamerly. Accelerated k-means with adaptive distance bounds. In *5th NIPS workshop on optimization for machine learning*, pages 42–53, 2012.

[49] Olive Jean Dunn. Multiple comparisons among means. *Journal of the American Statistical Association*, 56(293):52–64, 1961.

[50] Omar El-Daghar, Erik Lundberg, and Robert Bridges. Egbter: Capturing degree distribution, clustering coefficients, and community structure in a single random graph model. In *International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 282–289. IEEE, 2018.

[51] Charles Elkan. Using the triangle inequality to accelerate k-means. In *Proceedings of the 20th International Conference on Machine Learning (ICML)*, pages 147–153, 2003.

[52] P Erdős and A Rényi. On random graphs. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

[53] Euclid. *Elements.* Alexandria, ca. 300 BC.

[54] C. E. Ferreira, A. Martin, C. C. de Souza, R. Weismantel, and L. A. Wolsey. The node capacitated graph partitioning problem: A computational study. *Mathematical Programming*, 81(2):229–256, Apr 1998.

[55] C. Fiduccia and R. Mattheyses. A linear time heuristic for improving network partitions. In *Proceedings of the 19th ACM/IEEE Design Automation Conference*, pages 175–181, June 1982.

[56] Tobias Friedrich and Anton Krohmer. On the diameter of hyperbolic random graphs. *SIAM Journal on Discrete Mathematics*, 32(2):1314–1334, 2018.

[57] Lin Fu, Sergej Litvinov, Xiangyu Y. Hu, and Nikolaus A. Adams. A novel partitioning method for block-structured adaptive meshes. *Journal of Computational Physics*, 341:447–473, 2017.

[58] Daniel Funke, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free massively distributed graph generation. In *Proceedings of the 32nd International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018.

[59] A. Ganesh, L. Massoulie, and D. Towsley. The effect of network topology on the spread of epidemics. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 1455–1466. IEEE, 2005.

[60] Michael R Garey and David S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM Journal on Computing*, 4(4):397–411, 1975.

[61] Mark S. Gordon, Dmitri G. Fedorov, Spencer R. Pruitt, and Lyudmila V. Slipchenko. Fragmentation Methods: A Route to Accurate Calculations on Large Systems. *Chemical Revue*, 112(1):632–672, 2012.

[62] C Fonseca Guerra, JG Snijders, G te Velde, and E J. Baerends. Towards an order-n dft method. *Theoretical Chemistry Accounts*, 99(6):391–403, 1998.

[63] Luca Gugelmann, Konstantinos Panagiotou, and Ueli Peter. Random hyperbolic graphs: degree sequence and clustering. In *Proceedings of the 39th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 7392 of *LNCS*, pages 573–585. Springer, 2012.

[64] Greg Hamerly. Making k-means even faster. In *Proceedings of the 2010 SIAM International Conference on Data Mining (SDM)*, pages 130–140. SIAM, 2010.

[65] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[66] B. Hendrickson and R. Leland. A multi-level algorithm for partitioning graphs. In *Supercomputing '95 (SC)*, page 28 (CD). ACM Press, 1995.

[67] Bruce Hendrickson and Tamara G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26(12):1519–1534, 2000.

[68] Bruce Hendrickson and Robert Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*, 16(2):452–469, 1995.

[69] Herbert W Hethcote. The mathematics of infectious diseases. *SIAM review*, 42(4):599–653, 2000.

[70] David Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38:459–460, 1891.

[71] David Hilbert. Über Flächen von konstanter Gaußscher Krümmung. In *Algebra · Invariantentheorie · Geometrie*, pages 437–448. Springer, 1933.

[72] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[73] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. Engineering a scalable high quality graph partitioner. In *Proceedings of the 24th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2010.

[74] Xiaocheng Hu, Miao Qiao, and Yufei Tao. Independent range sampling. In *Proceedings of the 33rd Symposium on Principles of database systems (PODS)*, pages 246–255. ACM, 2014.

[75] Jan Hungershöfer and Jens-Michael Wierum. On the quality of partitions based on space-filling curves. In *Proceedings of the International Conference on Computational Science-Part III (ICCS)*, pages 36–45. Springer, 2002.

[76] Christoph R Jacob, S. Maya Beyhan, Rosa E Bulo, André Severo Pereira Gomes, Andreas W Götz, Karin Kiewisch, Jetze Sikkema, and Lucas Visscher. PyADF — A scripting framework for multiscale quantum chemistry. *Journal of Computational Chemistry*, 32:2328–2338, 2011.

[77] Christoph R Jacob and Johannes Neugebauer. Subsystem density-functional theory. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 4:325–362, 2014.

[78] Harold Jeffreys. *The theory of probability.* OUP Oxford, 1998.

[79] Frank Jensen. *Introduction to Computational Chemistry.* Wiley, 2nd edition, 2007.

[80] Ibrahim Kamel and Christos Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 500–509. Morgan Kaufmann Publishers, 1994.

[81] George Karypis. Multi-constraint mesh partitioning for contact/impact computations. In *Proceedings of the ACM/IEEE conference on Supercomputing (SC)*, page 56. ACM, 2003.

[82] George Karypis and Vipin Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, pages 28–28. IEEE, 1998.

[83] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.

[84] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell system technical journal*, 49(2):291–307, 1970.

[85] Brian Wilson Kernighan. *Some graph partitioning problems related to program segmentation.* PhD thesis, Princeton University, 1970.

[86] Karin Kiewisch, Christoph R. Jacob, and Lucas Visscher. Quantum-Chemical Electron Densities of Proteins and of Selected Protein Sites from Subsystem Density Functional Theory. *Journal of chemical theory and computation*, 9:2425–2440, 2013.

[87] J. Kim and T. Wilhelm. What is a complex graph? *Physica A: Statistical Mechanics and its Applications*, 387:2637–2652, April 2008.

[88] Shad Kirmani and Padma Raghavan. Scalable parallel graph partitioning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 51:1–51:10. ACM, 2013.

[89] Marcos Kiwi and Dieter Mitsche. A bound for the diameter of random hyperbolic graphs. In *Proceedings of the Meeting on Analytic Algorithmics and Combinatorics*, pages 26–39. SIAM, 2015.

[90] Robert Kleinberg. Geographic routing using hyperbolic space. In *26th IEEE International Conference on Computer Communications (INFOCOM)*, pages 1902–1909. IEEE, 2007.

[91] Tamara G. Kolda, Ali Pinar, Todd Plantenga, and C. Seshadhri. A scalable generative graph model with community structure. *SIAM Journal on Scientific Computing*, 36(5):C424–C452, Sep 2014.

[92] Hans-Peter Kriegel, Peter Kunath, and Matthias Renz. Probabilistic nearest-neighbor query on uncertain objects. In *Advances in databases: concepts, systems and applications*, pages 337–348. Springer, 2007.

[93] Dmitri Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguñá. Hyperbolic geometry of complex networks. *Physical Review E*, 82(3):036106, Sep 2010.

[94] Dmitri Krioukov, Fragkiskos Papadopoulos, Amin Vahdat, and Marián Boguñá. Curvature and temperature of complex networks. *Physical Review E*, 80(3):035101, 2009.

[95] Richard E Ladner and Michael J Fischer. Parallel prefix computation. *Journal of the ACM (JACM)*, 27(4):831–838, 1980.

[96] Sebastian Lamm. Communication efficient algorithms for generating massive networks. Master's thesis, Karlsruhe Institute of Technology (KIT), 2017.

[97] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Physical review E*, 78(4):046110, 2008.

[98] Janos K. Lanyi and Brigitte Schobert. Structural changes in the L photointermediate of bacteriorhodopsin. *Journal of Molecular Biology*, 365(5):1379 – 1392, 2007.

[99] John M Lee. *Riemannian manifolds: an introduction to curvature*, volume 176 of *Graduate Texts in Mathematics.* Springer, 2006.

[100] Juan Li and Chen-Ching Liu. Power system reconfiguration based on multilevel graph partitioning. In *PowerTech, IEEE Bucharest*, pages 1–5. IEEE, 2009.

[101] Stuart P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–136, 1982.

[102] H.P. Lovecraft. Dreams in the witch house. In *Weird Tales*. Rural Publishing Corporation, 1933.

[103] Bruno RC Magalhães, Farhan Tauheed, Thomas Heinis, Anastasia Ailamaki, and Felix Schürmann. An efficient parallel load-balancing framework for orthogonal decomposition of geometrical data. In *Proceedings of the 31st International Conference on High Performance Computing*, pages 81–97. Springer, 2016.

[104] Priya Mahadevan, Dmitri Krioukov, Kevin Fall, and Amin Vahdat. Systematic topology analysis and generation using degree correlations. In *SIGCOMM Computer Communication Review*, volume 36, pages 135–146. ACM, 2006.

[105] Meena Mahajan, Prajakta Nimbhorkar, and Kasturi Varadarajan. The planar k-means problem is np-hard. In *International Workshop on Algorithms and Computation (WALCOM)*, LNCS, pages 274–285. Springer, 2009.

[106] O. Marquardt and S. Schamberger. Open benchmarks for load balancing heuristics in parallel adaptive finite element computations. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 685–691. CSREA Press, 2005.

[107] Youssef M Marzouk and Ahmed F Ghoniem. K-means clustering for optimal partitioning and dynamic load balancing of parallel hierarchical n-body simulations. *Journal of Computational Physics*, 207(2):493–528, 2005.

[108] Henning Meyerhenke. Shape optimizing load balancing for mpi-parallel adaptive numerical simulations. In *Proceedings of the of the 10th DIMACS Implementation Challenge – Graph Partitioning and Graph Clustering*, Contemporary Mathematics. American Mathematical Society, 2013.

[109] Henning Meyerhenke, Burkhard Monien, and Thomas Sauerwald. A new diffusion-based multilevel algorithm for computing graph partitions. *Journal of Parallel and Distributed Computing*, 69(9):750–761, 2009. Best Paper Awards and Panel Summary: IPDPS 2008.

[110] Henning Meyerhenke, Burkhard Monien, and Stefan Schamberger. Graph partitioning and disturbed diffusion. *Parallel Computing*, 35(10–11):544–569, 2009.

[111] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Parallel graph partitioning for complex networks. *IEEE Transactions on Parallel and Distributed Systems*, 28(9):2625–2638, 2017.

[112] Henning Meyerhenke and Thomas Sauerwald. Beyond good partition shapes: An analysis of diffusive graph partitioning. *Algorithmica*, 64(3):329–361, 2012.

[113] Henning Meyerhenke and Stefan Schamberger. A parallel shape optimizing load balancer. In *Proceedings of the 12th International Euro-Par Conference*, volume 4128 of *LNCS*, pages 232–242. Springer, 2006.

[114] Joel C Miller and Aric Hagberg. Efficient generation of networks with given expected degrees. In *Algorithms and Models for the Web Graph*, pages 115–126. Springer, 2011.

[115] Sparsh Mittal and Jeffrey S Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):69, 2015.

[116] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis.* Cambridge University Press, 2005.

[117] K. W. Morton and D. F. Mayers. *Numerical Solution of Partial Differential Equations: An Introduction.* Cambridge University Press, 2005.

[118] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 19:45–74, 2010.

[119] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69:026113, Feb 2004.

[120] Mark Newman. *Networks: An Introduction.* Oxford University Press, 2010.

[121] Mark EJ Newman. Clustering and preferential attachment in growing networks. *Physical review E*, 64(2):025102, 2001.

[122] Maximillian Nickel and Douwe Kiela. Poincaré embeddings for learning hierarchical representations. In *Advances in neural information processing systems (NIPS)*, volume 30, pages 6338–6347, 2017.

[123] Christian Ochsenfeld, Jorg Kussmann, and D. S. Lambrecht. Linear-Scaling Methods in Quantum Chemistry. In *Reviews in Computational Chemistry*, volume 23, pages 1–82. Wiley, 2007.

[124] Johan Gotthardt Olsen, Claus Flensburg, Ole Olsen, Gerard Bricogne, and Anette Henriksen. Solving the structure of the bubble protein using the anomalous sulfur signal from single-crystal in-house Cu Kα diffraction data only. *Acta Crystallographica Section D*, 60(2):250–255, 2004.

[125] Mats Ormö, Andrew B. Cubitt, Karen Kallio, Larry A. Gross, Roger Y. Tsien, and S. James Remington. Crystal structure of the aequorea victoria green fluorescent protein. *Science*, 273(5280):1392–1395, 1996.

[126] Fragkiskos Papadopoulos, Maksim Kitsak, M Ángeles Serrano, Marián Boguná, and Dmitri Krioukov. Popularity versus similarity in growing networks. *Nature*, 489(7417):537–540, 2012.

[127] Fragkiskos Papadopoulos, Dmitri Krioukov, Marián Boguñá, and Amin Vahdat. Greedy forwarding in dynamic scale-free networks embedded in hyperbolic metric spaces. In *29th Conference on Computer Communications (INFOCOM)*, pages 1–9. IEEE, 2010.

[128] Jose Antonio Pascual, Javier Navaridas, and Jose Miguel-Alonso. Effects of topology-aware allocation policies on scheduling performance. In *Job Scheduling Strategies for Parallel Processing (JSSPP)*, volume 5798 of *LNCS*, pages 138–156. Springer, 2009.

[129] Georgios A. Pavlopoulos, Maria Secrier, Charalampos N. Moschopoulos, Theodoros G. Soldatos, Sophia Kossida, Jan Aerts, Reinhard Schneider, and Pantelis G. Bagos. Using graph theory to analyze biological networks. *BioData Mining*, 4(1):1–27, 2011.

[130] Jian Pei, Ming Hua, Yufei Tao, and Xuemin Lin. Query answering techniques on uncertain and probabilistic data: tutorial summary. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1357–1364. ACM, 2008.

[131] François Pellegrini. Scotch and PT-scotch graph partitioning software: An overview. In *Combinatorial Scientific Computing*, pages 373–406. CRC Press, 2012.

[132] Manuel Penschuck. Generating Practical Random Hyperbolic Graphs in Near-Linear Time and with Sub-Linear Memory. In *Proceedings of the 16th International Symposium on Experimental Algorithms (SEA)*, volume 75 of *LIPIcs*, pages 26:1–26:21, 2017.

[133] Pavel A Pevzner, Haixu Tang, and Michael S Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

[134] PRACE. Unified european applications benchmark suite, 2016. Accessed: 2017-09-30.

[135] Maria Predari and Aurélien Esnard. A k-way greedy graph partitioning with initial fixed vertices for parallel applications. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 280–287. IEEE, 2016.

[136] James P. Quirk and Rubin Saposnik. Admissibility and measurable utility functions. *The Review of Economic Studies*, 29(2):140–146, 1962.

[137] R. Ramage, J. Green, T. W. Muir, O. M. Ogunjobi, S. Love, and K. Shaw. Synthetic, structural and biological studies of the ubiquitin system: the total chemical synthesis of ubiquitin. *Biochemical Journal*, 299(1):151–158, 1994.

[138] Marko Robnik. An extremum property of the n-dimensional sphere. *Journal of Physics A: Mathematical and General*, 13(10):L349, 1980.

[139] Marti Rosas-Casals, Sergi Valverde, and Ricard V Solé. Topological vulnerability of the european power grid under errors and attacks. *International Journal of Bifurcation and Chaos*, 17(07):2465–2475, 2007.

[140] John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. Probabilistic programming in python using pymc3. *PeerJ Computer Science*, 2:e55, 2016.

[141] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers, 2005.

[142] Peter Sanders and Christian Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA)*, volume 7933 of *LNCS*, pages 164–175. Springer, 2013.

[143] V. Sarkar, W. Harrod, and A. E Snavely. Software challenges in extreme scale systems. *Journal of Physics: Conference Series*, 180(1):012045, 2009.

[144] K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high performance scientific simulations. In *The Sourcebook of Parallel Computing*, pages 491–541. Morgan Kaufmann Publishers, 2003.

[145] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14(3):219–240, 2002.

[146] David Sculley. Web-scale k-means clustering. In *Proceedings of the 19th international conference on World wide web*, pages 1177–1178. ACM, 2010.

[147] C Seshadhri, Tamara G Kolda, and Ali Pinar. Community structure and scale-free collections of Erdős-Rényi graphs. *Physical Review E*, 85(5):056109, 2012.

[148] C. Seshadhri, Ali Pinar, and Tamara G. Kolda. The similarity between stochastic Kronecker and Chung-Lu graph models. In *Proceedings of the SIAM International Conference on Data Mining (SDM)*, pages 1071–1082. SIAM, 2012.

[149] Yuval Shavitt and Tomer Tankel. Hyperbolic embedding of internet graph for distance estimation and overlay construction. *IEEE/ACM Transactions on Networking (TON)*, 16(1):25–36, 2008.

[150] Horst D Simon. Partitioning of unstructured problems for parallel processing. *Computing systems in engineering*, 2(2-3):135–148, 1991.

[151] George M Slota, Sivasankaran Rajamanickam, Karen Devine, and Kamesh Madduri. Partitioning trillion-edge graphs in minutes. In *Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS)*, pages 646–655. IEEE, 2007.

[152] Sorin Solomon and Peter Richmond. Power laws of wealth, market order volumes and market returns. *Physica A: Statistical Mechanics and its Applications*, 299(1-2):188–197, 2001.

[153] Christian L. Staudt, Michael Hamann, Ilya Safro, Alexander Gutfraind, and Henning Meyerhenke. Generating scaled replicas of real-world complex networks. In *Proceedings of the 5th International Workshop on Complex Networks and their Applications*, volume 693 of *SCI*, pages 17–28. Springer, 2016.

[154] Christian L Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. NetworKit: A tool suite for large-scale complex network analysis. *Network Science*, 4(4):508–530, 2016.

[155] Valerie E. Taylor and Bahram Nour-Omid. A study of the factorization fill-in for a parallel implementation of the finite element method. *International Journal for Numerical Methods in Engineering*, 37(22):3809–3823, 1994.

[156] G t Te Velde, F Matthias Bickelhaupt, Evert Jan Baerends, C Fonseca Guerra, Stan JA van Gisbergen, Jaap G Snijders, and Tom Ziegler. Chemistry with ADF. *Journal of Computational Chemistry*, 22(9):931–967, 2001.

[157] Matus Telgarsky and Andrea Vattani. Hartigan's method: k-means clustering without voronoi. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*, pages 820–827, 2010.

[158] Jeffrey Travers and Stanley Milgram. The small world problem. *Psychology Today*, 2(1):61–67, 1967.

[159] Dale E. Tronrud and James P. Allen. Reinterpretation of the electron density at the site of the eighth bacteriochlorophyll in the fmo protein from pelodictyon phaeum. *Photosynthesis Research*, 112(1):71–74, 2012.

[160] Abraham A Ungar. Einstein's special relativity: Unleashing the power of its hyperbolic geometry. *Computers & Mathematics with Applications*, 49(2-3):187–221, 2005.

[161] Moritz von Looz and Henning Meyerhenke. Querying probabilistic neighborhoods in spatial data sets efficiently. In *International Workshop on Combinatorial Algorithms (IWOCA)*, volume 9843 of *LNCS*, pages 449–460. Springer, 2016.

[162] Moritz von Looz and Henning Meyerhenke. Updating dynamic random hyperbolic graphs in sublinear time. *Journal of Experimental Algorithmics (JEA)*, 23(1):1–6, 2018.

[163] Moritz von Looz, Mustafa Safa Özdayi, Sören Laue, and Henning Meyerhenke. Generating massive complex networks with hyperbolic geometry faster in practice. In *High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2016. ©2016 IEEE. Reprinted with permission.

[164] Moritz von Looz, Roman Prutkin, and Henning Meyerhenke. Generating random hyperbolic graphs in subquadratic time. In *Proceedings of the 26th International Symposium on Algorithms and Computation (ISAAC)*, volume 9472 of *LNCS*, pages 467–478. Springer, 2015. Extended preliminary at `https://arxiv.org/abs/1501.03545`.

[165] Moritz von Looz, Charilaos Tzovas, and Henning Meyerhenke. Balanced k-means for parallel geometric partitioning. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP)*, pages 52:1–52:10. ACM, 2018.

[166] Moritz von Looz, Mario Wolter, Christoph R Jacob, and Henning Meyerhenke. Better partitions of protein graphs for subsystem quantum chemistry. In *International Symposium on Experimental Algorithms (SEA)*, volume 9685 of *LNCS*, pages 353–368. Springer, 2016.

[167] C. Walshaw and M. Cross. Jostle: Parallel multilevel graph-partitioning software – an overview. In *Mesh Partitioning Techniques and Domain Decomposition Techniques*, pages 27–58. Civil-Comp Ltd., 2007.

[168] Duncan J Watts and Steven H Strogatz. Collective dynamics of ‘small-world’ networks. *Nature*, 393(6684):440, 1998.

[169] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[170] Roy D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency: Practice and Experience*, 3(5):457–481, 1991.

List of Publications

Most of the results contained in this thesis were first published in peer-reviewed venues.

Journals

- Moritz von Looz and Henning Meyerhenke. Updating dynamic random hyperbolic graphs in sublinear time. *Journal of Experimental Algorithmics (JEA)*, 2018.

Conferences

- Moritz von Looz, Roman Prutkin, and Henning Meyerhenke. Generating random hyperbolic graphs in subquadratic time. In *Proceedings of the 26th International Symposium on Algorithms and Computation (ISAAC)*, volume 9472 of *LNCS*, pages 467–478. Springer, 2015. Extended preliminary at `https://arxiv.org/abs/1501.03545`
- Moritz von Looz, Mario Wolter, Christoph R. Jacob, and Henning Meyerhenke. Better partitions of protein graphs for subsystem quantum chemistry. In *Proceedings of the International Symposium on Experimental Algorithms (SEA)*, volume 9685 of *LNCS*, pages 353–368. Springer, 2016
- Moritz von Looz and Henning Meyerhenke. Querying probabilistic neighborhoods in spatial data sets efficiently. In *International Workshop on Combinatorial Algorithms (IWOCA)*, volume 9843 of *LNCS*, pages 449–460. Springer, 2016
- Moritz von Looz, Mustafa Safa Özdayi, Sören Laue, and Henning Meyerhenke. Generating massive complex networks with hyperbolic geometry faster in practice. In *High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2016
- Moritz von Looz, Charilaos Tzovas, and Henning Meyerhenke. Balanced k-means for parallel geometric partitioning. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP)*, pages 52:1–52:10. ACM, 2018.
- Daniel Funke, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free massively distributed graph generation. In *Proceedings of the 32nd International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018. Best Paper Award.