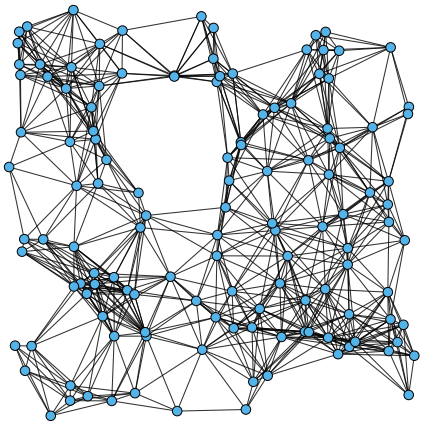
Karlsruher Institut für Technologie

Master thesis

Communication Efficient Algorithms for Generating Massive Networks

Sebastian Lamm

Date: 10. Januar 2017



Supervisors: Prof. Dr. Peter Sanders
Dr. rer. nat. Christian Schulz
Dr. Darren Strash

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

Abstract

Massive complex systems are prevalent throughout all of our lives, from various biological systems as the human genome to technological networks such as Facebook or Twitter. Rapid advances in technology allow us to gather more and more data that is connected to these systems. Analyzing and extracting this huge amount of information is a crucial task for a variety of scientific disciplines.

A common abstraction for handling complex systems are networks (graphs) made up of entities and their relationships. For example, we can represent wireless ad hoc networks in terms of nodes and their connections with each other. We then identify the nodes as vertices and their connections as edges between the vertices. This abstraction allows us to develop algorithms that are independent of the underlying domain.

Designing algorithms for massive networks is a challenging task that requires thorough analysis and experimental evaluation. A major hurdle for this task is the scarcity of publicly available large-scale datasets. To approach this issue, we can make use of network generators [21]. These generators allow us to produce synthetic instances that exhibit properties found in many real-world networks.

In this thesis we develop a set of novel graph generators that have a focus on scalability. In particular, we cover the classic Erdős-Rényi model, random geometric graphs and random hyperbolic graphs. These models represent different real-world systems, from the aforementioned wireless ad-hoc networks [40] to social networks [44]. We ensure scalability by making use of pseudorandomization via hash functions and redundant computations. The resulting network generators are *communication agnostic*, i.e. they require no communication. This allows us to generate massive instances of up to 2^{43} vertices and 2^{47} edges in less than 22 minutes on 32.768 processors.

In addition to proving theoretical bounds for each generator, we perform an extensive experimental evaluation. We cover both their sequential performance, as well as scaling behavior. We are able to show that our algorithms are competitive to state-of-the-art implementations found in network analysis libraries. Additionally, our generators exhibit near optimal scaling behavior for large instances. Finally, we show that pseudorandomization has little to no measurable impact on the quality of our generated instances.

Zusammenfassung

Stetig wachsende komplexe Systeme enormer Größe lassen sich in jedem Bereich unseres Lebens finden. Von unterschiedlichen biologischen Systemen wie dem menschlichen Genom bis hin zu technologischen Netzwerken wie Facebook oder Twitter. Rasche technologische Fortschritte erlauben es uns, die enormen Datenmengen, die mit diesen Systemen verbunden sind, zu sammeln. Das Analysieren und Auswerten der resultierenden Menge an Informationen ist eine bedeutende Aufgabe für unterschiedliche wissenschaftliche Gebiete.

Netzwerke, aufgebaut aus (abstrakten) Objekten und deren Beziehungen, sind eine häufig verwendete Abstraktion, um mit komplexen Systemen umzugehen. So lassen sich beispielsweise drahtlose Ad-Hoc Netze, bestehend aus einzelnen Endgeräte und deren Verbindungen, als Netzwerke modellieren. Hierfür werden die Endgeräte durch Knoten und deren Verbindungen durch Kanten repräsentieren, die die einzelnen Knoten miteinander verbinden. Diese Repräsentation, ermöglicht es uns Algorithmen zu entwickeln, die unabhängig von einer spezifischen Domäne sind.

Die Entwicklung von Algorithmen für riesige Netzwerke ist eine fordernde Aufgabe, die eine sorgfältige theoretische Analyse und experimentelle Evaluation erfordert. Ein bedeutendes Problem zur Bewältigung dieser Aufgabe ist die mangelhafte Verfügbarkeit von hinreichend großen Datensätzen. Um dieses Problem zu lösen, lassen sich sogenannte Netzwerkgeneratoren einsetzen [21]. Netzwerkgeneratoren erlauben es uns synthetische Instanzen von Netzwerken zu erzeugen, die Eigenschaften realer Daten aufweisen.

In dieser Masterarbeit entwickeln wir eine Reihe von Netzwerkgeneratoren, die einen besonderen Fokus auf Skalierbarkeit legen. Insbesondere behandeln wir das klassische Erdős-Rényi Modell, sowie unterschiedliche geometrische Modelle basierend auf Euklidischer und hyperbolischer Geometrie. Diese Modelle entsprechen verschiedenen Typen realer Netzwerke, von drahtlosen Ad-Hoc Netzen bis hin zu sozialen Netzwerken. Die Skalierbarkeit unserer Generatoren gewährleisten wir durch die Verwendung von Pseudozufall mittels Hash-Funktionen und redundanten Berechnungen. Die daraus resultierenden Algorithmen sind *kommunikationsagnostisch*, d. h. sie sind nicht auf Kommunikation angewiesen. Dies erlaubt es uns, riesige Instanzen mit bis zu 2^{43} Knoten und 2^{47} Kanten in weniger als 22 Minuten auf 32.768 Prozessoren zu erzeugen.

Zusätzlich zur theoretischen Analyse unserer Generatoren führen wir eine umfangreiche experimentelle Evaluation durch. Hierfür betrachten wir sowohl die sequentielle Leistungsfähigkeit als auch die Skalierbarkeit unserer Algorithmen. Hierdurch sind wir in der Lage, die Kompetitivität unserer Algorithmen gegenüber State-of-the-Art Implementierungen zu zeigen. Weiterhin weisen unsere Generatoren beinahe optimale Skalierbarkeit für große Eingaben auf. Schließlich zeigen wir, dass die Verwendung von Pseudozufall wenig bis kaum messbare Auswirkungen auf die Qualität der von uns erzeugten Instanzen hat.

Acknowledgments

I would like to give thanks to everyone who supported and encouraged me during my work on this thesis. Without all the people that cheered me up after the occasional setbacks, I wouldn't have gotten so far.

First of all, I owe special thanks to my supervisors Dr. Christian Schulz, Dr. Darren Strash and Prof. Dr. Peter Sanders for the chance of doing research on such an interesting topic and a lot of good advice. Furthermore, I would like to thank the Forschungszentrum Jülich for providing us with access to the JUQUEEN supercomputer. Last but not least, I would like to thank my friends for their support and the time spent together. My sincerest thanks to all of you!

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Ort, den Datum

Contents

Abstract vi

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Our Results . . . 2
- 1.3 Structure of Thesis . . . 3

2 Fundamentals 5
- 2.1 General Definitions . . . 5
  - 2.1.1 Probability Distributions . . . 5
  - 2.1.2 Graphs . . . 8
- 2.2 Network Models . . . 9
  - 2.2.1 Erdős-Rényi Model . . . 9
  - 2.2.2 Random Geometric Model . . . 10
  - 2.2.3 Random Hyperbolic Model . . . 10

3 Related Work 13
- 3.1 Sampling and Random Variables . . . 13
  - 3.1.1 Sampling Without Replacement . . . 13
  - 3.1.2 Sampling Probability Distributions . . . 14
- 3.2 Graph Generators . . . 15
  - 3.2.1 Erdős-Rényi Model . . . 15
  - 3.2.2 Random Geometric Model . . . 16
  - 3.2.3 Random Hyperbolic Model . . . 17
  - 3.2.4 Other Network Models . . . 17

4 Divide-and-Conquer Sampling 21
- 4.1 Sequential Sampling Algorithm . . . 21
- 4.2 Parallel Sampling Algorithm . . . 21
- 4.3 Generalizations . . . 23
  - 4.3.1 Sorted Sampling . . . 23
  - 4.3.2 Load Balancing . . . 24

5 Graph Generators **25**
5.1 Overview . . . 25
5.2 Erdős-Rényi Generator . . . 25
5.2.1 Directed $G(n, m)$ Generator . . . 25
5.2.2 Undirected $G(n, m)$ Generator . . . 27
5.2.3 Directed $G(n, p)$ Generator . . . 30
5.2.4 Undirected $G(n, p)$ Generator . . . 31
5.3 Random Geometric Generator . . . 33
5.3.1 2D Generator . . . 33
5.3.2 3D Generator . . . 37
5.4 Random Hyperbolic Generator . . . 41
5.4.1 Sequential Generator . . . 41
5.4.2 Parallel Generator . . . 46

6 Experimental Evaluation **51**
6.1 Implementation . . . 51
6.2 Experimental Setup . . . 53
6.2.1 Environment . . . 53
6.2.2 Experiment Design . . . 53
6.3 Running Time Comparison . . . 55
6.3.1 Sampling Without Replacement . . . 55
6.3.2 Erdős-Rényi Generators . . . 56
6.3.3 Random Geometric Generators . . . 56
6.3.4 Random Hyperbolic Generators . . . 59
6.4 Scaling Behavior . . . 60
6.4.1 Sampling Without Replacement . . . 60
6.4.2 Erdős-Rényi Generators . . . 60
6.4.3 Random Geometric Generators . . . 63
6.4.4 Random Hyperbolic Generators . . . 64
6.5 Pseudorandomization . . . 66

7 Discussion **69**
7.1 Conclusion . . . 69
7.2 Future Work . . . 70

A Pseudocode **71**

B Graph Properties **79**

Bibliography **93**

1 Introduction

1.1 Motivation

Emergence describes the process of structures, patterns and properties arising from the interactions of simpler entities in complex systems [31]. This process is prevalent throughout various disciplines including philosophy, science and art. A popular example of this phenomenon are the complex and diverse symmetrical patterns that arise in snowflakes through ever changing atmospheric conditions. But not only in non-living, physical systems can we observe the arising of large and complex entities, but also in our own society [71]. The World Wide Web is a highly decentralized system that is made up of roughly 45 billion pages [6] and just as many links that run between them. Even though links between pages are not created by any central organization, and therefore are subject to a certain degree of randomness, the resulting network exhibits a very distinct structure. For example, there is only a small set of pages that have a significantly higher amount of links pointing to them than the large majority of the web.

Rapid advances in technology as of 2016 have led these systems to expand and grow with unprecedented scale. The social network website Facebook alone has roughly 1.7 billion active users, each of which has an average of 350 friends [5]. The patterns and laws that govern the growth of such massive systems are intriguing for governments, companies and researchers alike. There are numerous applications that make use of this knowledge, e.g. information-sharing for law-enforcement [22], analyzing and predicting the spreading of diseases [57, 9] and structural packet routing strategies [76]. We can develop a mathematical understanding of these complex systems by analyzing them in terms of networks of related activities. For example, we may identify the Internet as a set of vertices that represent websites. A website is said to be connected to another if it has a link to that website. Using this information we can then compute the influence of a certain website on others. To study these concepts, different mathematical models of networks have been developed over the last decades [18]. These models focus on different metrics for characterization of complex networks such as their degree distribution or clustering coefficients. By using randomization and observing these metrics during the growth of a network, we can analyze their emergent properties [10]. In turn, we can use the observed properties to make assumptions on the growth of real-world networks and develop new algorithms to handle them efficiently. For example, vertices in social networks are likely to be connected by a small number of intermediate vertices (small-world phenomenon). By combining this

knowledge with location information we are able to develop new greedy routing algorithms specifically targeted towards such networks [41].

Algorithms for massive networks are important for extracting meaning from the sheer amount of data represented by them. Ensuring scalability, both in theory and practice, is one of our guiding principles for these algorithms. A major hurdle on the way to this goal is the scarcity of publicly available large-scale datasets to experimentally verify their scaling behavior in practice. To approach this problem, we can make use of network generators [21]. Network generators use mathematical models to generate instances that exhibit many of the same properties that are found in real-world networks, such as social networks. In theory, some of these generators allow us to generate massive networks that scale up indefinitely. Tough in practice, many generators are limited to generating moderately sized instances of up to a few thousand vertices in a reasonable amount of time. These limitations are often attributed to the apparently sequential nature of the mathematical models or hardware limitations [49, 56, 77].

1.2 Our Results

In this thesis we develop a set of novel network generators that focus on scalability. By doing so, we are able to generate massive networks that rival the current state-of-the-art. To be more specific, we are able to generate networks of up to 2^{43} vertices and 2^{47} edges in less than 22 minutes on 32.768 processors. In contrast, the largest instances commonly generated for supercomputer benchmarks consist of 2^{42} vertices and 2^{45} edges and come from a single graph family [4].

The generators we propose use different types of network models, from the classic Erdős-Rényi model to various geometric models. They all share a common goal of communication efficiency. In particular, they are are communication *agnostic*. This means that they require no communication at all, besides knowing their rank and the total number of processors involved. This is achieved by redundantly computing small parts of the resulting network. The amount of recomputations is kept at a bare minimum to achieve a good trade-off between communication efficiency and redundancy. Additionally, since all of the network models involve random choices, we have to ensure that different processors perform the same actions for redundant computations. This is achieved by making use of pseudorandomization. Pseudorandomness is a commonly used tool for applications that generate random-like behavior. An example are Monte-Carlo simulations [51] that are used in different areas such as radiation therapy [61] and VLSI design [39]. However, pseudorandomness is rarely exploited for communication efficiency in network generators. Instead, many generators use communication primitives to share vertices and/or edges between processors [35, 54]. By using pseudorandomization, different processors are able to come to the same random decisions, while the generated networks still exhibit statistical randomness.

We compare our generators against implementations found in state-of-the-art libraries (e.g. Boost[1] and NetworKit [70]). We consider both their sequential speed as well as scaling capabilities.

1.3 Structure of Thesis

In Section 2 we introduce the basic mathematical notations and definitions needed for understanding the concepts presented in this thesis. We also cover the different theoretical network models that form the basis of our generators. We present recent developments for these models and the current state-of-the-art in terms of efficiency and scalability in Section 3. In Section 4 we discuss the distributed sampling algorithm that serves as a foundation for most of our graph generators. We introduce the generators and perform an analysis on their running time and scalability in Section 5. In Section 6 we present our experimental evaluation of these generators and compare them to state-of-the-art implementations. We take a detailed look at their scalability, both in terms of weak and strong scaling. Finally, we discuss our findings and present possibilities for future work in this area in Section 7.

[1] http://www.boost.org

2 Fundamentals

2.1 General Definitions

2.1.1 Probability Distributions

A *probability distribution* describes the relationship between outcomes of a statistical experiment and their probability of occurrence. The set of possible outcomes of such an experiment is called the *sample space*. Based on the properties of the sample space, we can further divide probability distributions into discrete and continuous probability distributions. A *discrete probability distribution* can be described as the list of probabilities for the different outcomes. This list is also known as a probability mass function (PMF). For *continuous probability distributions* the probability of each individual outcome is 0. Only events that represent infinitely many outcomes, such as intervals, can have a positive probability. Therefore, we are not able to describe them using a probability mass function. Instead, we use a probability density function (PDF), which describes the infinitesimal probability for any single outcome.

As an example, the simplest probability distribution occurs when all outcomes of an experiment have an equal outcome. This is called a *uniform distribution*. We now cover different probability distributions, both discrete and continuous, that are relevant for our different network models.

Binomial Distribution. The binomial distribution $B(n, p)$ is a discrete probability distribution that was introduced by Johann Bernoulli [15]. It describes the number of successes in a sequence of $n \in \mathbb{N}$ independent yes or no experiments. Each of these experiments yields the answer yes with a probability of $p \in [0, 1]$. If a random variable X follows a binomial distribution with parameters n, p, we denote this as $X \sim B(n, p)$. The probability that the sequence of n experiments yields a yes answer k times is given by the PMF

$$f(k; n, p) = \Pr(X = k) = \binom{n}{k} p^k (1-p)^{n-k}. \tag{2.1.1}$$

Here, $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ $(k = 0, 1, 2, \ldots, n)$ is the *binomial coefficient*. In practice, the binomial distribution is often used to used to model the number of successes in a sample of size n drawn *with replacement* from a finite population of size N.

Multinomial Distribution. The multinomial distribution $M(n, p_1, \ldots, p_k)$ is a generalization of the binomial distribution to experiments that have $k \geq 2$ different outcomes. Each outcome is assigned a probability $p_i \geq 0$ such that $\sum_{i=0}^{k} p_i = 1$. The multinomial distribution then describes the number of times each outcome is obtained in a sequence of $n \in \mathbb{N}$ independent experiments. If a random variable $X = (X_1, \ldots, X_k)$ follows a multinomial distribution with parameters $n, p_1, \ldots, p_i$, we denote this as $X \sim M(n, p_1, \ldots, p_i)$. The probability that the sequence of n experiments yields each individual outcome x_i times is given by the PMF

$$f(x_1, \ldots, x_k; n, p_1, \ldots, p_k) = \Pr(X_1 = x_1, \ldots, X_k = x_k) \tag{2.1.2}$$

$$= \begin{cases} \frac{n!}{x_1! \cdots x_k!} p_1^{x_1} \cdots p_k^{x_k} & \text{when } \sum_{i=1}^{k} x_i = n \\ 0 & \text{otherwise} \end{cases}. \tag{2.1.3}$$

.

Geometric Distribution. The geometric distribution $\text{Geo}(p)$ is a discrete probability distribution which has two closely related variants. For our purposes it models the probability of having $k \in \{0, 1, 2, \ldots\}$ successive failures before a first success in a number of independent trials. The probability for a success is given by $0 < p \leq 1$. If a random variable X follows a geometric distribution with parameter p, we denote this as $X \sim \text{Geo}(p)$. The probability to draw k successive failures before a first success is given by the PMF

$$f(k; p) = \Pr(X = k) = (1 - p)^k p \tag{2.1.4}$$

Thus, the sequence of probabilities describes a geometric sequence. These geometric sequences were studied 2500 years ago by Euclid in his *Elements* [34].

In practice, the geometric distribution is commonly used in algorithms for sampling without replacement [72] to compute skip distances between sampled elements.

Hypergeometric Distribution. The hypergeometric distribution $H(N, K, n)$ is a discrete probability distribution. One of the its first appearances was as a solution to a problem found in *De ratiociniis in ludo aldae* [37]. In contrast to the binomial distribution, it describes the probability of having k successes in n draws, *without replacement*, from a finite population of size N. The likelihood of drawing a success is given by the total number of successes K within the population. If a random variable X follows a hypergeometric distribution with parameters N, K, n, we denote this as $X \sim H(N, K, n)$. The probability that the sequence of n draws yields a success k times is given by the PMF:

$$f(k; N, K, n) = \Pr(X = k) = \frac{\binom{K}{k}\binom{N-K}{n-k}}{\binom{N}{n}}. \tag{2.1.5}$$

Note that the hypergeometric distribution can be approximated by a binomial distribution under certain circumstances. To be more specific, let $p = K/N$. If N and K are large

compared to the sample size n, and p is not close to 0 or 1, then $H(N, K, n)$ can be approximated by a binomial distribution with parameters n and p. This is useful in practice since it is often easier to compute binomial random variables than hypergeometric ones due to the higher number of factorials needed for their computation [68].

Power-law Distribution. A power-law describes a functional relationship between two quantities where one quantity varies as a power of the other. In turn, power-law distributions describe distributions whose probability density function (or probability mass function in the discrete case) follows such a power-law. We say that a random variable X follows a power-law distribution with parameter γ if

$$f(k;\gamma) = \Pr(X = k) \sim k^{-\gamma}. \tag{2.1.6}$$

For the case of degree distribution of networks, the parameter γ is typically in the range $2 < \gamma < 3$ [23, 55].

Chernoff Bounds and Union Bound. Chernoff bounds [33] are used to derive exponentially decreasing bounds on tail distribution of sums of independent random variables. To be more specific, we let $X_1, \ldots, X_n$ be random indicator variables with $\Pr[X_i = 1] = p_i$ and $\Pr[X_i = 0] = (1 - p_i)$. Furthermore, we define $X = \sum_{i=0}^{n} X_i$ and $\mu = E[X]$. For any $\delta \in (0, 1]$, the Chernoff bounds are now defined as

$$\Pr[X < (1-\delta)\mu] < \Big(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\Big)^{\mu}$$

for the lower tail, and

$$\Pr[X > (1+\delta)\mu] < \Big(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\Big)^{\mu}$$

for the upper tail.

In case that the random variables X_i are not independent, we can additionally make use of the *union bound* [19]. The union bound for a set of countable events $X_1, \ldots, X_n$ is defined as

$$\Pr[X_1 \cup \ldots \cup X_n] \leq \sum_{i=1}^{n} \Pr[X_i].$$

We can then use Chernoff bounds and the union bound to derive a bound on the number of balls assigned to a bin in a n balls into m bins scenario. For this purpose, we assume that each ball is independently assigned to the ith bin with probability $Pr[X_i = 1] = p_i$. The p_i are chosen in such a way that $\sum_{i=0}^{m} p_i = 1$ and $p_i \in \mathcal{O}(1/m)$. Thus, the expected number of balls in the ith bin is $\mathcal{O}(n/m)$.

Lemma 2.1.1. *The number of balls assigned to the ith bin in a n balls into m bins scenario is $\mathcal{O}(n/m)$ with high probability*[1].

Proof. See Raab and Steger [59]. ∎

2.1.2 Graphs

A *graph* (network) is defined as a pair $G = (V, E)$ of *vertices* V and *edges* E. We denote the set of vertices of a graph G as $V(G) = \{1, \ldots, n\}$. For a directed (undirected) graph the set of edges consists of ordered (unordered) pairs $E(G) \subseteq V(G) \times V(G)$. We define $n = |V|$ to be the number of vertices and $m = |E|$ the number of edges. The two vertices that are part of an edge $e = (u, v)$ are said to be *adjacent*. For directed graphs the order of vertices of an edge is important: $e = (u, v)$ is different from $e' = (v, u)$. An edge $(u, u) \in E$ is called a *self-loop*. If not mentioned otherwise, we only consider graphs that contain no self-loops.

A graph that consists of n vertices can have at most $\binom{n}{2} = \frac{n(n-1)}{2}$ edges, i.e. when all vertices are pairwise adjacent. We define a graph to be *sparse* iff $m \in O(n)$, and *dense* iff $m \in \Theta(n^2)$.

Graphs are often represented by their *adjacency matrix* $\mathbf{A}$. The adjacency matrix is a $n \times n$ square matrix, whose entry $a_{i,j}$ ($i, j \in \{1, \ldots, n\}$) is one if the edge (i, j) exists , and zero otherwise.

A *subgraph* of a graph G is defined as a pair $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$. The vertex set V' of the subgraph has to include all endpoints of the corresponding edge set E', but may also contain additional vertices.

Node Degree and Degree Distribution

The set of *neighbors* for any vertex $v \in V$ is defined as $N(v) = \{u \in V(G) \mid (u, v) \in E(G)\}$. For an undirected graph, we define the *degree* of a vertex $v \in V$ as $\deg(v) = \Delta(v) = |N(v)|$. In the directed case, we have to separate between the *indegree* and *outdegree* of a vertex. The indegree is defined as $\Delta^-(v) = \{u \in V(G) \mid (u, v) \in E(G)\}$. Analogously, the outdegree is defined as $\Delta^+(v) = \{u \in V(G) \mid (v, u) \in E(G)\}$. Finally, the *total degree* $\Delta(v)$ of a vertex in a directed graph is the sum of its indegree and outdegree. The maximum degree of G is represented by Δ or $\Delta(G) = \max_{v \in V} \Delta(v)$.

An important property for graphs in theory and practice is the *degree distribution* $P(k)$. The degree distribution measures the fraction of nodes in the graph that have a degree of k. Again, for directed graphs, we have to separate between the inbound degree distribution $P(k_{\text{in}})$ and outgoing degree distribution $P(k_{\text{out}})$. Depending on the type of network model the degree distribution may vary significantly. For example, social networks tend to have a *power-law* degree distribution, while random graphs have a binomial degree distribution [53].

[1] i.e. with probability at least $1 - p^{-c}$ for any constant c

Connectivity

A vertex $u \in V$ is *connected* to a vertex $v \in V$ iff there exists a path of directed or undirected edges between the two vertices. In directed graphs, if u is connected to v and vice versa, they are *strongly connected*. Note that for an undirected graph connectedness implies strongly connectedness. A graph is called (strongly) connected iff each pair of vertices $u, v \in V$ is (strongly) connected. Undirected subgraphs in which any two vertices are connected to each other and are maximal are called *connected components*. In the same way we can define strongly connected components for directed subgraphs. The size and number of (strongly) connected components is a frequently used characteristic for analyzing network models.

Clustering Coefficient

Another important metric is the *clustering coefficient*. The clustering coefficient is a measurement of how much vertices tend to cluster together in the graph. If not mentioned otherwise, we use the *global* clustering coefficient which is based on triangles and triplets of vertices [53]. A triplet consists of three connected vertices. In turn, triangles are made of three closed triples, one for each triangle vertex. The global clustering coefficient is defined by the ratio of triangles to the total number of triplets. To be more specific

$$C = \frac{3 \times \text{\#triangles}}{\text{\#triplets}} = \frac{\text{\#closed triplets}}{\text{\#triplets}}. \tag{2.1.7}$$

Social networks usually have a larger clustering coefficient than random graphs with the same vertex set [52].

2.2 Network Models

2.2.1 Erdős-Rényi Model

The Erdős-Rényi (ER) model is one of the most commonly known models for generating random graphs. A random graph is obtained by uniformly sampling a graph from the set of all possible graphs with a set of n vertices. Random graphs created by the ER model can be both directed or undirected. We now briefly introduce the two closely related variants of the ER model.

The first version, proposed by Edgar Gilbert [30], is denoted as the $G(n, p)$ model. In this model we start from a set of n vertices and randomly add edges between them. Each of these edges is added independently with a probability $0 < p < 1$. As a result, all graphs with n vertices and m edges have an equal probability of $p^m(1-p)^{\binom{n}{2}-m}$. In particular, for $p = 0.5$ each of the $2^{\binom{n}{2}}$ possible graphs with n vertices is chosen with equal probability. The expected number of edges for a $G(n, p)$ random graph is $\binom{n}{2}p$. The probability that

a particular vertex v of a graph with n vertices has a certain degree k follows a binomial distribution:

$$\Pr[\deg(v) = k] = \binom{n-1}{k} p^k (1-p)^{n-1-k}. \tag{2.2.1}$$

The second version, proposed by Paul Erdős and Alfréd Rényi [24], is denoted as the $G(n, m)$ model. In the $G(n, m)$ model, we chose a graph uniformly at random from the set of all graph which have n vertices and m edges. This means that for $0 \leq m \leq N = \binom{n}{2}$ $G(n, m)$ has $\binom{N}{m}$ elements. Each of these elements occurs with an equal probability of $1/\binom{N}{m}$. The $G(n, m)$ model has very similar properties to the $G(n, p)$ model, but the latter is often easier to analyze because of the independence of edges.

2.2.2 Random Geometric Model

Random geometric graphs are spatial networks that place n vertices as points in a metric space using a specified probability distribution [58]. Two vertices are connected by an edge iff their distance in the metric space is within a given threshold. Since the distance between two vertices is a symmetric relationship, we are only interested in undirected graphs.

The metric space, its dimension and the distribution of vertices can vary depending on the use-case of the model. In this thesis we are looking at random geometric graphs in the two- and three-dimensional unit square $[0, 1)^{\{2,3\}}$. To measure distances between two vertices p, q, we use the Euclidean distance $\text{dist}(p, q) = \sqrt{\sum_{i=1}^{d\in\{2,3\}} (p_i - q_i)^2}$.

We use a radius $r > 0$ in order to add edges. If the distance between two vertices u and v is less than r, we add an undirected edge $\{u, v\}$ to the graph. Thus, this model can be described using the two parameters n and r.

To distribute vertices we sample the position of each vertex uniformly and independently at random in the unit square. This can be done by generating a set of uniform random variables on the interval $[0, 1)$ for each vertex. Following the construction algorithm of vertices and edges, the expected degree of any vertex that does not lie on the border is $n\pi r^2$ [58].

2.2.3 Random Hyperbolic Model

Random hyperbolic graphs are a different variant of spatial networks proposed by Krioukov et al. [44]. Instead of Euclidean space, which has flat curvature, they generate graphs using the hyperbolic plane, which has negative curvature. They show how this graph family naturally develops a power-law degree distribution (with $\gamma \geq 2$) and other features of complex real-world networks. These features are controlled by choosing the correct parameters for the average degree and vertex density. Vertices in this model are generated as points (ϕ, r) in polar coordinates on a disk of radius R in the hyperbolic plane with curvature $-\zeta^2$. This disk will be denoted as $\mathcal{D}_R$.

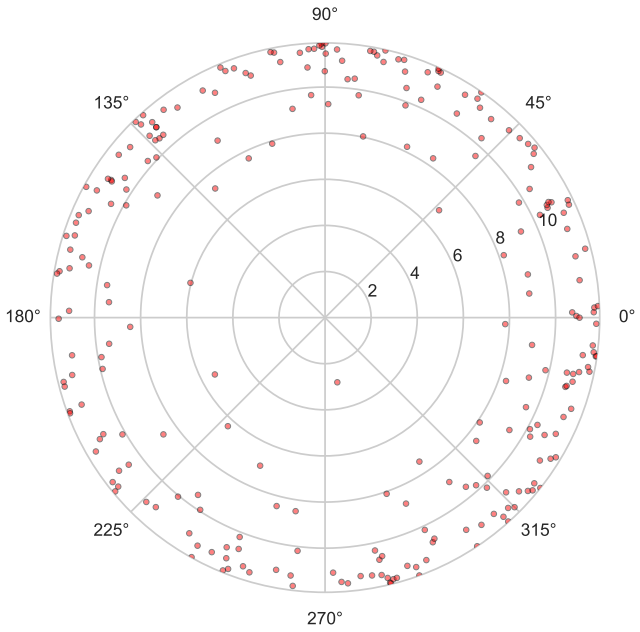


Figure 2.1: Point distribution on the disk $\mathcal{D}_R$ for $\alpha = 1$ and a set of 256 vertices with an average degree of 5.0. The disk radius R equals 11.91.

To generate points in the hyperbolic plane the angular coordinate ϕ is chosen uniformly at random from the interval $[0, 2\pi)$. The radial coordinate r is drawn using the probability density function

$$f(r) = \alpha \frac{\sinh(\alpha r)}{\cosh(\alpha R) - 1}. \tag{2.2.2}$$

The parameter α controls the growth of the random graph and determines the vertex density. For the case $\alpha = 1$ this results in a uniform random distribution on hyperbolic space in $\mathcal{D}_R$. Figure 2.1 shows an example of the vertex distribution on $\mathcal{D}_R$ for $\alpha = 1$ and 256 vertices with an average degree of 5.0. In general, the higher the value of α the more points tend to be on the border of $\mathcal{D}_R$ and vice versa. As with the random geometric model, we connect vertices p, q iff their hyperbolic distance

$$\text{dist}_H(p, q) = \cosh r_p \cosh r_q - \sinh r_p \sinh r_q \cos |\phi_p - \phi_q| \leq R. \tag{2.2.3}$$

There also exists a more general model where edges are added with a probability based on the hyperbolic distance between two vertices [44]. In this thesis we only focus on the more deterministic approach.

The neighborhood of a vertex in the deterministic model consists of all the vertices that are within a hyperbolic disk of radius R around it. The average degree is thus controlled via this radius. Krioukov et al. [44, 32] showed that for $\alpha/\zeta > \frac{1}{2}$ the degree distribution follows a power-law distribution with exponent $2\alpha/\zeta + 1$.

3 Related Work

We now present important advances related to the algorithms and generators proposed in this thesis. We cover various state-of-the-art algorithms for sampling without replacement from a finite population, as well as sampling from arbitrary probability distributions. We then highlight recent advances for generators for the graph models presented in the last section. This includes Erdős-Rényi, random geometric, and random hyperbolic graphs. Additionally, we cover other models and generators that are relevant for the design of our algorithms or their comparison.

3.1 Sampling and Random Variables

In this section we discuss the current state-of-the-art sampling algorithms. We first cover algorithms for generating a (sorted) sample from a fixed population without replacement. Additionally, we discuss algorithms for generating random variables from the distributions discussed in the previous section. Both of these topics are important for our distributed sampling algorithm as well as graph generators. For example, we use hypergeometric random variables to generate a distribution of edges that are then sampled without replacement. The section on sampling without replacement is an adaptation from Sanders et al. [64] which was written by Peter Sanders and Lorenz Hübschle-Schneider.

3.1.1 Sampling Without Replacement

We first consider the classic problem of sampling n numbers (elements) from a population $\{1, \dots, N\}$ *without replacement*. This is an important ingredient for many algorithms in data mining or statistics. The restriction that the population consists of integers from 1 to N is without loss of generality. If we want to sample from a general set M of size N, we can represent this set by an array of size N and use the array indices as our population. To further avoid special cases, we assume $n \leq N/2$. Otherwise, one can simple generate the $N - n < N/2$ elements that are *not* in the sample and take the remaining elements. We now discuss the current state-of-the-art of sampling algorithms.

Algorithm S. The algorithm performs a linear scan over the range $\{1, \dots, N\}$ and generates a uniformly random deviate for each element to decide whether it is sampled. Because of the linear scan this becomes increasingly slow for very large N. Nonetheless, this algorithm is widely used, including the GNU Scientific Library [3].

Algorithm H. This algorithm is a simple and efficient folklore algorithm that is very fast for small n [25, 43]. The sample is kept in a hash table T which is initially empty. To produce the next sample element, we generate uniformly random deviates X from the range $\{1, \ldots, N\}$. If $X \in T$, the element is rejected as it was already sampled, otherwise it is inserted. This algorithm has an expected running time $\mathcal{O}(n)$.

Algorithm D. Vitter [72] proposed an elegant sequential algorithm for generating a sorted sample without any additional auxiliary data structures. The sample is created by essentially generating appropriate random deviates. These deviates specify the number of positions to skip to the next sample element. Note that the distribution of the random deviates changes in each steps. Nonetheless, using a sophisticated technique based on the rejection method, this can be done in constant expected time. Therefore, the algorithm runs in expected time $\mathcal{O}(n)$.

Algorithm B. Ahrens and Dieter [7] proposed an algorithm based on Bernoulli samples. Each element of the range $\{1, \ldots, N\}$ is sampled with probability $p \approx n/N$. This yields a sample with $n' \approx n$ elements. If this sample is too big ($n' > n$) it can be repaired by removing $n' - n$ elements uniformly at random. The case $n' < n$ can be made highly unlikely by choosing p roughly larger than n/N. On the off chance that n' is still smaller than n one can simply restart the sampling process. The Bernoulli sampling itself can be implemented efficiently by generating geometrically distributed random deviates. These deviates describe the number of elements to skip in each step. The algorithm is faster than Algorithm D since generating geometric random deviates requires fewer arithmetic operations.

3.1.2 Sampling Probability Distributions

We now discuss the state-of-the art considering sampling from the various distributions presented in the previous section. Most of these approaches are based on the acceptance-rejection method [60, 75]. The idea behind this method is that the probability mass function (probability density function) $f(x)$ of the target distribution X, i.e. a binomial distribution, is approximated by another distribution Y with a probability function $g(x)$. We use this method if it is difficult to sample from the target distribution directly, i.e. if more simplistic approaches based on the inverse PMF (PDF) are impossible. Instead of sampling from X directly, we then sample from Y and accept the sample with probability $p(x) = \frac{f(x)}{k \cdot g(x)}$. If a sample is rejected with probability $1 - p(x)$, the process is repeated using a newly generated sample. This process is repeated until a sample was successfully accepted. The constant k is chosen such that $k \cdot g(x) \geq f(x)$ for all possibilities of x. Acceptance-rejection sampling works for any distribution in $\mathbb{R}^m$ that has a density.

The main advantage of this sampling method over others is that the running time does not grow depending on the value of x. This is crucial for designing efficient algorithms that

require sampling from various probability distributions. Nonetheless, the running time of acceptance-rejection sampling is heavily influenced by the rejection rate. Therefore, one has to find an approximation function $g(x)$, that is both easy to compute and has a low rejection rate.

Various improvements can be made to further raise the acceptance rate and therefore lower the running time of the sampling algorithm. Most of these improvements make use of sophisticated approximation functions [68, 69].

3.2 Graph Generators

We now discuss the state-of-the-art for the graph generators presented in the last section. We also highlight additional graph models when they cover relevant aspects for our own generators or serve as competitors.

3.2.1 Erdős-Rényi Model

Batagelj and Brandes [13] present sequential algorithms for the $G(n, p)$ as well as $G(n, m)$ model that have an optimal running time. Both of their algorithms are adaptations of different sampling routines.

For their $G(n, p)$ generator they make use of an adaptation of Algorithm D by Vitter [72]. This means that they use geometric random deviates to skip edges that are not sampled. The probability of sampling an edge after k tries is $(1-p)^{k-1}p$. Therefore, one can assign an interval $I_k \subseteq [0, 1)$ of length $(1-p)^{k-1}p$ to each positive integer k. Skip distances are then sampled by selecting the smallest k for which I_k ends after a random $r \in [0, 1)$. Their algorithm is able to generate a $G(n, p)$ graph in time $\mathcal{O}(n + m)$, which is optimal.

This approach would also be feasible for building a $G(n, m)$ generator. The main problem in this case is that skip distances are not independent of the current algorithm state. To be more specific, if $t - 1$ candidate edges have been checked and l of them have been sampled, the probability of skipping $k - 1$ edges is

$$\prod_{i=1}^{t+k-1} (1 - \frac{m-l}{\binom{n}{2} - i + 1}) \frac{m-l}{\binom{n}{2} - t + k}. \tag{3.2.1}$$

The resulting algorithm would have an expected running time $\mathcal{O}(n+m)$. However, the time per edge is not always constant. In turn, they propose two different versions of $G(n, m)$ generators. The first one is an adaptation of Algorithm H where edges are sampled uniformly at random using a hash table. If an edge was already picked, one can simply retry with a different sample. They then continue to show that their algorithm has an expected linear running time $O(m)$. Their second version of the $G(n, m)$ generator is based on a virtual Fisher-Yates shuffle [28] which eliminates the uncertainty in the number of iterations.

Nobari et al. [54] proposed a data parallel generator for both the directed and undirected $G(n, p)$ model. Their generators are designed for graphics processing units (GPUs). They first develop a sequential algorithm that makes use of a geometric distribution to compute the number of edges to skip between samples, similar to the algorithm of Batagelj and Brandes [13]. Additionally, they use precomputations to avoid costly evaluations of logarithms during the evaluation of the geometric distribution. To adapt their algorithm to a data parallel setting, they first create random numbers using a parallel pseudorandom number generator. They then use these random numbers to concurrently compute skip values. Finally, they compute absolute edge indices by using a parallel prefix sum.

3.2.2 Random Geometric Model

Generating random geometric graphs with n vertices and radius r can be done naively by comparing all n vertices in $\Theta(n)$ time. This bound can be improved if the vertices are known to be generated uniformly at random [35]. To this end, a tiling (overlay) of the unit square into squares (cells) is created. Each of these cells has a side length of r. Thus, the number of cells in each row and column is $k = \lceil \frac{1}{r} \rceil$. Each vertex is then associated with a cell which can be retrieved in constant time. Since the vertices are generated uniformly at random, each cell contains an expected number of $\mathcal{O}(n/k^2)$ vertices. Thus, the time complexity of generating the grid data structure is $\mathcal{O}(k^2 + n)$.

To find the neighbors of each vertex, we consider each cell C and its neighbors C' (side length of r). We then compute the distance for each pair of vertices in C and C' and add edges accordingly. Since there are k^2 cells, this step takes expected time $\mathcal{O}(n^2/k^2)$. In total, the generator has expected time complexity $\mathcal{O}(k^2 + n + n^2/k^2) = \mathcal{O}(n + m)$ (see Lemma 5.3.1).

Holtgrewe and Sanders [35] proposed a distributed memory parallelization of this algorithm. Their algorithm assumes a number of $P = p^2$ processes, each of which owns a square of $k/p \times k/p$ cells. Each of the processes starts by independently generating $\frac{n}{p}$ vertices. These vertices are then distributed to their owners which are able to sort them by their cell number. Afterwards, a global index is created for each local vertex, and border cells are exchanged with neighboring processes. Finally, each processor can generate the edges for his local vertices independently.

Since they sort local vertices using Quicksort, the expected time complexity for local computation is $\mathcal{O}(n/p \log(n/p))$ Additionally, the expected time needed for communication is bounded by $T_{\text{all-to-all}}(n/p, p) + T_{\text{all-to-all}}(1, p) + 4T_{\text{point-to-point}}(n/(k \cdot p) + 2)$. $T_{\text{all-to-all}}(l, c)$ is the time needed for an all-to-all communication step with messages of length l between c communication partners. $T_{\text{point-to-point}}(l)$ is the time needed for a point-to-point communication step between two communication partners and messages of length l.

3.2.3 Random Hyperbolic Model

The naive construction of random hyperbolic graphs takes time $\Theta(n^2)$. Von Looz et al. [73, 74] improved this bound to $\mathcal{O}((n^{3/2} + m) \log n)$ and $\mathcal{O}(n \log n + m)$ (empirical observation) respectively. Bringmann et al. [20] proposed a theoretical algorithm with an optimal expected linear time complexity that is based on a generalization of random hyperbolic graphs. We now briefly discuss each of these approaches.

For their first algorithm von Looz et al. [73] relate the hyperbolic space to Euclidean geometry using the Poincaré disk model. This model uses a n dimensional hypersphere to represent an n-dimensional hyperbolic space. In particular, one can use the Euclidean unit disk $U_1(0)$ to represent the hyperbolic plane. One important property of this representation is that hyperbolic circles are mapped onto Euclidean circles. They use this fact to generate a polar quadtree on the Poincaré disk to answer neighbor queries. The polar quadtree itself can be generated in time $\mathcal{O}(n \log n)$. To compute the neighbors for all vertices their algorithm then needs time $\mathcal{O}((n^{3/2} + m) \log n)$.

In contrast to the polar quadtree algorithm, their second approach [74] generates random hyperbolic graphs directly in the hyperbolic plane. They do so by partitioning the hyperbolic plane into concentric ring-shaped slabs. The slabs are chosen in such a way, that each slab contains an equal expected amount of vertices. One can then use these slabs to limit the number of distance calculations necessary during the edge insertion. This is done by computing angular boundaries for neighborhood queries. To find boundary vertices quickly, vertices are stored in sorted order within each slab. The resulting generator suggest a time complexity of $\mathcal{O}(n \log n + m)$, but no explicit proof of this bound is given.

The approach by Bringmann et al. [20] uses a generalization of random hyperbolic graphs called *Geometric Inhomogeneous Random Graphs* (GIRGs). Their model promises to make theoretical studies of random hyperbolic graphs easier by ignoring constant factors while maintaining their qualitative behavior. Additionally, they propose an optimal sampling algorithm for GIRGS with expected linear time. Their algorithm works by performing a sophisticated partitioning of the underlying space into cells. The geometric data structure build on this partitioning allows traversing nodes in close proximity in expected amortized constant time. The first implementation of their algorithm was given by Bläsius et al. [16] as part of their embedding algorithm for scale-free graphs in the hyperbolic plane.

3.2.4 Other Network Models

We now discuss recent advances for network models that are not a main focus of this thesis. We do so because some of these advances are important for understanding the context and contribution of this thesis.

Barabasi-Albert Model

Recently, Sanders and Schulz [65] proposed an algorithm for generating massive scale-free networks. Their algorithm is based on the popular preferential attachment model by Barabasi and Albert [12] (BA model). In this model a graph is generated one vertex at a time and a fixed number of d edges is added to existing vertices. The probability of a vertex being selected for an edge insertion is proportional to its current degree. This process naturally results in graphs that exhibit a power-law distribution.

Batagelj and Brandes [13] proposed an optimal sequential algorithm for this model with time complexity $\mathcal{O}(n + m)$. Their algorithm works by generating one edge at a time and writing them into an edge array E of size $2dn - 1$. In this array, edge i is represented by its corresponding endpoints which are stored at positions $2i$ and $2i + 1$ respectively. Therefore, $E[2i] = \lfloor i/d \rfloor$. The key observation is that we get the same degree distribution as the original BA model by sampling the target vertex uniformly at random from the already existing edges [65]. To this end, the second endpoint $E[2i+1]$ is set to $E[x]$ where x is chosen uniformly at random from $\{0, \ldots, 2i\}$.

Apparently, the main problem for the scalable execution of this algorithm is its inherently sequential nature. Sanders and Schulz [65] solve this issue by making clever use of pseudo-randomization and edge recomputations. In particular, they use pseudorandomness to reproduce random behavior when generating edges. In a parallel setting this can be enabled by using hash functions that map array positions to pseudorandom numbers. This trick allows them to compute edges independently from one another. Thus, their network generator is able to generate scale-free graphs in an embarrassingly parallel fashion. The concepts of pseudo-randomization and recomputations used in their algorithm will frequently occur in our own graph generators.

Meyer and Penschuck [50] also proposed two I/O-efficient BA model generators for the external memory model. Additionally, they extend one of their generators to a massively parallel setting. Like the previous approach, their work is based upon the sequential algorithm of Batagelj and Brandes [13]. In the external memory model, this algorithm would produce $\Omega(m)$ I/Os with high probability, because of the highly randomized access pattern. To alleviate this fact, they propose an algorithm called *TFP-BA* which uses time-forward processing [11] to delay the generation of edges. Therefore, they use tokens that represent the creation of an edge and queries to it. The dependencies between these tokens form an acyclic graph. They then use an external memory priority queue to process the tokens in the correct order.

Their algorithm needs $\mathcal{O}(\mathit{scan}(m_0) + \mathit{sort}(m))$ [1] I/Os and has a time complexity of $\mathcal{O}(m_0 + m \log m)$ where m_0 is the number of edges of a seed graph $G_0 = (V_0, E_0)$. In order to extend their algorithm for efficient parallelization, they use a custom sorted edge list and two parallelization schemes based on tree-composition and token-wise parallelism.

[1] *scan(N)* $= \Theta(N/B)$ is the number of I/Os needed to read/write N contiguous items with block size B. *sort(N)* $= \Theta(N/B) \cdot \log_{M/B}(N/B)$ is the number of I/Os needed to sort N contiguous items with block size B using a fast internal memory with capacity M.

Stochastic Kronecker Graphs

Stochastic Kronecker graphs were first proposed in 2005 by Leskovec et al. [46]. They showed that these graphs match many of the properties found in large real-world networks, such as scale-free degree distributions or small diameters. The main idea behind this model is to recursively create self similar graphs using the Kronecker product. Given a $n \times m$ matrix $\mathbf{U} = [u_{i,j}]$ and a $n' \times m'$ matrix $\mathbf{V}$ the Kronecker product is defined as

$$\mathbf{S} = \mathbf{U} \otimes \mathbf{V} = \begin{bmatrix} u_{1,1}\mathbf{V} & u_{1,2}\mathbf{V} & \cdots & u_{1,m}\mathbf{V} \\ u_{2,1}\mathbf{V} & u_{2,2}\mathbf{V} & \cdots & u_{2,m}\mathbf{V} \\ \vdots & \vdots & \ddots & \vdots \\ u_{n,1}\mathbf{V} & u_{n,2}\mathbf{V} & \cdots & u_{n,m}\mathbf{V} \end{bmatrix} \in \mathbb{R}^{(n \cdot n') \times (m \cdot m')} \tag{3.2.2}$$

This product is then used to successively multiply a seed graph G_0 with a corresponding $N_0 \times N_0$ adjacency matrix to create graphs of increasing size. By using a probability matrix U instead of the adjacency matrix we get a *stochastic* Kronecker graph. An entry $u_{i,j}$ of this probability matrix represents the probability that an edge between the vertices i and j is present. Fitting the matrix $\mathbf{U}$ to correspond to real-world networks can be done in linear time in the number of edges.

A special case of the stochastic Kronecker graph model is the recursive matrix model (R-MAT) by Chakrabarti et al. [21]. This model is well known for its usage in the popular Graph 500 benchmark [4]. To generate a graph G with n vertices and m edges its adjacency matrix A is subdivided into four equal sized parts. Edges are then added to the adjacency matrix by assigning each of the four partitions a probability a, b, c, d respectively. The probabilities are chosen such that $a + b + c + d = 1$. If a partition was chosen for an edge, it is again subdivided recursively using the same probabilities. This step is repeated until a 1×1 partition is encountered, in which case we add the corresponding edge to the graph. Using this procedure, adding a single edge to graph takes time $\mathcal{O}(\log n)$. Therefore the time complexity of the generator is $\mathcal{O}(m \log n)$.

The R-MAT model can be parallelized trivially since an edge can be added independently of other edges. Therefore, it is commonly used in large scale benchmarks such as the Graph 500 benchmark [4]. This benchmarks uses a variety of different graph sizes ranging from 2^{26} vertices and 2^{30} edges up to 2^{42} vertices and 2^{46} edges. The largest graph instances use 10^{15} bytes or roughly 1 PB of memory. The concept of subdividing the adjacency matrix of a graph into partitions is also an important part of our own ER generators.

4 Divide-and-Conquer Sampling

We now discuss our new divide-and-conquer algorithm for sampling n elements without replacement from a population of size N. We start by presenting the sequential variant and then proceed with its parallelization. Additionaly, we highlight some possible extensions and implementation details. Most of the theory presented in this section is an taken from Sanders [64] and was developed by Peter Sanders.

4.1 Sequential Sampling Algorithm

The sequential algorithm is based on the observation that the number of samples L up to an arbitrary splitting position $l \in \{1, \ldots, N\}$ follows a hypergeometric distribution. This distribution is parameterized by the number of samples n, the number of successes l, and the universe size N. Consequently, the number of samples starting at $l+1$ up to N is given by $n - L$. By generating a random deviate from this distribution ($L \sim H(N, l, n)$), we can divide the original sample step into two, one for $\{1, \ldots, l\}$ and one for $\{l+1, \ldots, N\}$. In order to distribute the samples evenly between the two subproblems, we choose $l = \lfloor N/2 \rfloor$.

Generating a random deviate from the hypergeometric distribution $H(N, l, n)$ can be done in expected constant time by using one of the acceptance-rejection algorithms presented in Section 3.1.2 (e.g. [69]). We then continue dividing the sample recursively until the number of samples is below a threshold n_0. Once the number of samples is below n_0, we can use one of the linear time sampling algorithm presented in Section 3.1.1. Since the recursion tree that is created has size of at most $2n/n_0$, the overall expected running time of our algorithm is $\mathcal{O}(n)$. The pseudocode for the sequential sampling algorithm is given in Algorithm 2.

4.2 Parallel Sampling Algorithm

We now consider the parallelization of the sequential sampling algorithm for P processors. Our goal is to partition the range $\{1, \ldots, N\}$ into P pieces, one for each processor, that can be computed individually. For this purpose, let N_i denote the last element in the range associated with processor i, i.e. processor i generates the sample elements that lie in the range $\{N_{i-1} + 1, \ldots, N_i\}$ ($N_0 = 0$). The main idea of the parallelization is to adapt the sequential sampling algorithm in such a way that the original range $\{1, \ldots, N\}$ is split into the subranges N_i for each processor i. This can be done by using $\lceil \log P \rceil$ levels of

recursion. Each processor then follows a single recursive call – the one which contains its local subrange.

Once each processor is left with its local subrange it can use any of the sequential sampling algorithms (e.g. D or H) to compute its local sample. However, there are some issues that we have to be aware of. The first issue is that in order to get consistent results we have to ensure that the hypergeometric random deviates generated by processors that follow the same path in the recursion tree are the same. Additionally, random deviates generated in two different subtrees have to be independent of each other. One way to achieve this would be using true randomness, e.g. by using a hardware random number generator [38]. The problem with this approach is that it requires communication to distribute the according random deviates to the processors. Since each recursion level would thus induce communication, this would limit the scalability of the parallel algorithm. Therefore, we make use of pseudorandomness. The main benefit of using pseudorandomness is that it exhibits statistical randomness while being much easier to produce than true randomness. For our purpose, we use a (high quality) hash function h as source of pseudorandomness. We then use the result of the hash function as a seed for generating hypergeometric random deviates in each recursion level. In particular, we use $h((j, k, t))$ to generate the t-th random deviates in the subproblem for processors $\{j, \ldots, k\}$. This allows us to achieve the desired effect without any communication. The pseudocode for Algorithm P is presented in Algorithm 1.

For the local sampling procedure we are still able to use any ordinary generator of pseudorandomness. This way, we might have a better trade-off between speed and quality than with hashing. Nonetheless, we can seed this generator with $h(i)$ on processor i in order to break the symmetry between processors.

The second issue is that processors need to know the global sample numbers N_j to generate the correct hypergeometric random deviates within their current subrange. In the case of an evenly distributed sampling universe, i.e. each processor has an equally sized subrange, this is easy. We simply compute $N_j = j\lceil n/P \rceil$ for $j < P$ and set $N_p = N$. These ranges still hold true, even if the number of processors P does not divide n, in which case the last processor gets a smaller subrange.

Finally, we obtain the following running time for our parallel algorithm.

Lemma 4.2.1. *If $max_i(N_i - N_{i-1}) = \mathcal{O}(N/P)$ then Algorithm P runs in time $\mathcal{O}(n/P + \log P)$ with high probability[1].*

Proof. If we only calculate with expectations, each processor generates at most $\lceil \log P \rceil$ hypergeometric random deviates and $\mathcal{O}(n/P)$ samples. Since the random deviates can be generated in expected constant time and the local sampling algorithms require expected linear time, the time complexity of Algorithm P follows easily. Nonetheless, we have to be careful to rule out rare cases that could lead to a slow down in computation, and thus in a larger overall execution time. We consider three issues: deviations in the number of

[1] i.e. with probability at least $1 - p^{-c}$ for any constant c

samples per processor, deviations in the time needed to generate hypergeometric random deviates, and deviations in the running time of the local sampling algorithm. We now briefly discuss each of these issues.

First, we show that the number of samples for each processor is $\mathcal{O}(n/P)$ with high probability. Since the number of samples per processor has a hypergeometric distribution, which spreads the elements more evenly than a binomial distribution [63], we can simplify the analysis. We do so by treating each sample j individually and assigning it to processor i with probability $p_{i,j} = N_i - N_{i-1} + 1 = \mathcal{O}(1/P)$. The total number of samples that get assigned to processor i is then represented by a random variable X_i with mean $\mu = \mathcal{O}(n/P)$. We can retrieve error bounds on the probability that we are close to μ by using Chernoff bounds (Lemma 2.1.1). By choosing $n = O(P \log P)$ and $n = \Omega(P \log P)$ these bounds yield $\mathcal{O}(n/P)$ samples per processor with high probability.

The second issue can be analyzed by looking at the structure of acceptance-rejection methods. As mentioned in Section 3.1.2, they sample from an approximation distribution and accept the result with a certain probability. The running time of these methods can thus be bound by a constant times a geometrically distributed random variable. It is easy to see that the sum of $\mathcal{O}(\log P)$ of these random variables is in $\mathcal{O}(\log P)$ with high probability.

The third issue depends on the local sampling algorithm that is used. For Algorithm D and the sequential divide-and-conquer algorithm the running time distribution can be bound using the same method as for the second issue. When using Algorithm H we have to make sure that the hash table has a sufficient size in order to get short tails for its running time distribution. This can be achieved by choosing a size of $\mathcal{O}(n/P + \log P)$. ∎

4.3 Generalizations

We now discuss further generalizations for our sequential and distributed sampling routines. In particular, we are covering sorted sampling and load balancing for heterogeneous hardware resources.

4.3.1 Sorted Sampling

The sequential sampling algorithm is able to easily generate a sorted sample given a base case sampling routine that generates samples in sorted order. This is due to the recursion tree being traversed *in-order*, i.e. the left subtree is recursively traversed before the right subtree. This poses no problem if we decide to use Algorithm D as our base case sampling routine.

Algorithm H can also be adapted to generate samples in sorted order. There are mainly two different variants to establish sortedness. The first one is to ignore the keys order during insertion and sort the hash table afterwards. This is done by first performing a condense operation on the hash table to remove gaps between clusters of elements. The resulting

condensed hash table of size n (the sample size) can then be sorted using a state-of-the-art sorting algorithm, e.g.insertion sort.

The second variant is highly implementation specific and requires a hash function that stores clusters of elements roughly in order. The main idea is to maintain the invariant that the samples in the hash table are sorted. This can be done by inserting elements at the correct positions within the clusters. We give a detailed explanation of this scheme in Section 6.1

4.3.2 Load Balancing

We now consider an execution environment where processors are not equally fast, e.g. when using heterogeneous hardware resources. Since the version of the sequential sampling algorithm presented previously implicitly assumes equally fast processors, this can lead to a slow-down in computation time. To solve this issue, we split our sampling population $\{1, \ldots, N\}$ into $P' \gg P$ jobs or subranges of roughly equal size. We can then assign these jobs dynamically to the processors by using standard load balancing techniques.

One possibility is to use a centralized master processors that assigns jobs to processors (workers). Once a processor has finished its jobs he sends a request for an additional job to the master. This is continued until all jobs have been processed. The main downside of this approach is the necessary communication between master and workers. The additional communication effort leads to a running time of $\mathcal{O}(n/P + P)$ opposed to $\mathcal{O}(n/P + \log P)$ (Theorem 4.2.1). A more scalable approach that does not impede the original running time uses *work stealing* [27, 17].

5 Graph Generators

5.1 Overview

In this section we present our novel distributed graph generators. All of these generators focus on communication-efficiency through redundant computations and pseudo-randomization. Infact, they are all communication *agnostic*, i.e. they require no communication at all. We begin our discussion by presenting the generators for the different Erdos-Renyi models. This includes directed and undirected versions of the $G(n, m)$ and $G(n, p)$ models. We then continue with two random geometric models that make use of Euclidean geometry (RGG) as well as hyperbolic geometry (RHG).

5.2 Erdős-Rényi Generator

We now discuss our generators for the two different Erdős-Rényi models, $G(n, m)$ and $G(n, p)$, introduced in Section 2.2.1. We begin by presenting our distributed generators for the directed and undirected $G(n, m)$ model. Afterwards, we discuss how to adapt the resulting algorithms for the $G(n, p)$ model. All of these algorithms are communication *agnostic*, i.e. they do not require any communication between individual processors (PEs). We also provide theoretical bounds for their running time and scalability.

5.2.1 Directed $G(n, m)$ Generator

Generating a directed graph in the $G(n, m)$ model can be reduced to sampling a graph from the set of all possible graphs with n vertices and m vertices. One way of doing so is to sample m edges uniformly at random from all possible $M = 2 \cdot \binom{n}{2} = n(n-1)$ directed edges $\{1, \ldots, M\}$. Since we are not interested in graphs with multi-edges, the sampling has to be performed *without* replacement. Therefore, our problem reduces to the efficient sampling of m elements from a population $\{1, \ldots, M\}$ without replacement. We already discussed different algorithms that are able to solve this problem in linear time $\mathcal{O}(m)$. For our generator, we use an adaptation of the distributed sampling algorithm presented in the previous section. For the rest of this section we assume we are given a set of P processors. In addition to P, each processor also knows its id $0 \leq i < P$. We also assume n is a multiple of P without loss of generality.

Our algorithm starts by dividing the set of possible edges into P *chunks*. Each chunk represents a set of rows of the adjacency matrix of our graph, which we call its *range*.

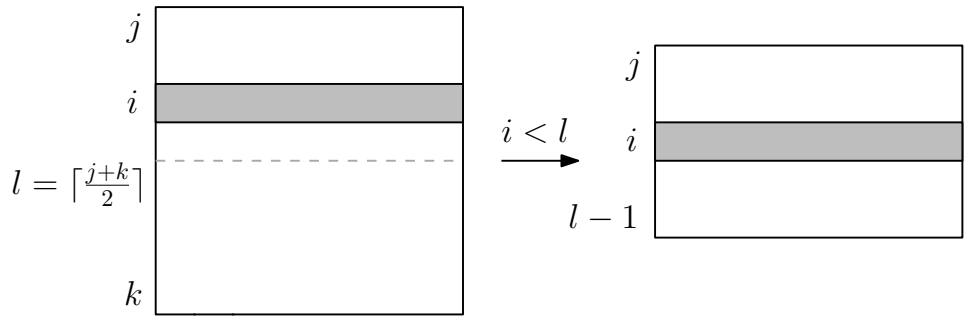


Figure 5.1: Recursion for the directed $G(n, m)$ generator for a set of processors $\{j, \ldots, k\}$. The chunk for processor i is highlighted. Since the chunk is in the upper half of the adjacency matrix, recursion only follow the upper subtree.

We then assign each chunk to its corresponding processor using its id i. Processor i then is responsible for generating the sample (set of edges) for its chunk. Note, that the distribution of vertices and edges to each processor is predetermined. Therefore, we let N_i (M_i) denote the last vertex (edge) in the range of chunk i.

In order to compute the correct sample size for each chunk, we use the same divide-and-conquer technique as with the distributed sampling algorithm. In each recursion step of this algorithm, we divide the set of chunks into two equal sized subsets $\{1, \ldots, l\}$ and $\{l + 1, \ldots, P\}$. We choose $l = \lfloor P/2 \rfloor$ on each level to ensure evenly divided ranges for each subtree. Each processor then continues the recursion, following its local chunk based on its id as seen in Figure 5.1. The recursion is stopped once we are left with a single chunk, which is then sampled locally using one of the algorithms discussed in Section 3.1.1. Note that the resulting recursion tree has at most $\lceil \log P \rceil$ levels and size $\mathcal{O}(P)$.

The base case of the recursion consists of sampling m' elements (edges) from a population of $M_P = \frac{n(n-1)}{P}$ (edges per processor) possible edges without replacement. To create an edge from a sampled element $s \in \{1, \ldots, M_P\}$ we set the source vertex to $v_s = \lfloor \frac{s-1}{n-1} \rfloor$. Analogously, we set the target vertex to $v_t = (s - 1) \mod (n - 1)$. We additionally increase the target id by one if it is larger or equal to the source id to avoid self-loops. To be more specific, if we compute the target id as described above, it holds that $v_t \in \{0, \ldots, v_s, \ldots, n - 2\}$. By increasing its value by one iff $v_t \geq v_s$, we shift its range to $\{0, \ldots, v_s - 1, v_s + 1, \ldots, n - 1\}$, which results in the desired edge distribution. Note, that the resulting edges have ids that are local to a certain processor. In order to get the correct global vertex ids, we add the offset N_{i-1} to both source and target. Thus, we finally insert the directed edge $(v_s + N_{i-1}, v_t + N_{i-1})$.

Lemma 5.2.1. *The directed $G(n, m)$ generator runs in time $\mathcal{O}(\frac{m+n}{P} + \log P)$ with high probability.*

Proof. Our algorithm is an adaptation of the distributed sampling algorithm that evenly divides the set of vertices, and therefore the set of potential edges, between P processors.

Thus, the population per processor has size $\mathcal{O}(N/P)$ and the running time directly follows from Theorem 4.2.1. ∎

5.2.2 Undirected $G(n, m)$ Generator

We now discuss how to adapt the divide-and-conquer scheme for undirected $G(n, m)$ graphs. Again, we want to evenly divide the set of vertices, and their corresponding edges, via chunks between the different processors. Additionally, we want each processor to generate all incident edges for its set of vertices. This is a common assumption for processing massive graphs on distributed systems where graphs are partitioned between computation nodes [14]. However, there is a major issue if we simply try to use the same approach we do for the directed case. To be more specific, if there is an undirected edge $\{i, j\}$, we have to make sure that *both* processors, the one that is assigned i and the one that is assigned j, sample this edge. Because both processors are assigned different chunks, they follow different paths in the recursion tree. Since the random variables generated in each subtree are independent of each other, it is highly unlikely that they both sample the edge $\{i, j\}$. To solve this issue, we only distribute the edges of the lower triangular adjacency matrix between the processors. However, we still want a processor to sample all the incident edges for its corresponding set of vertices. We now discuss how to distribute the chunks while maintaining this invariant. To this end, we introduce a different type of chunks.

We begin by dividing each dimension of the adjacency matrix into P sections of size n/P. A chunk is then defined as a set of edges that correspond to a $n/P \times n/P$ submatrix of the lower triangular adjacency matrix. Thus, we have a total of $\frac{P(P+1)}{2}$ chunks that can be arranged into a triangular $P \times P$ matrix as seen in Figure 5.2.

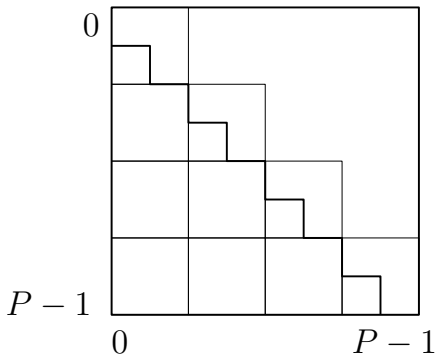


Figure 5.2: Example of an adjacency matrix subdivided into a $P \times P$ chunk matrix.

To evenly distribute the chunks, processor $i < P$ is assigned the $(i+1)$th row and column of this matrix. It therefore is responsible for generating the P chunks $\{(i, 0), \ldots, (i, i)\}$ and $\{(i, i), \ldots, (P, i)\}$. This corresponds to the set of vertices $\{(i-1) \cdot n/P + 1, \ldots, i \cdot n/P\}$ and all their incident edges. Again, we use N_i to identify the last vertex in the $(i+1)$th

row (or column) of chunks. By generating rectangular chunks instead of whole rows or columns, we can make sure that both processor i and processor $j \leq i$ redundantly generate chunk (i, j) using the same set of random values. Thus, they both sample the same set of edges independently from each other.

To generate our new chunks, we again use a divide-and-conquer approach. Our algorithm starts by dividing the $P \times P$ chunk matrix into equal sized quadrants, as seen in Figure 5.3. To do so, we split the rows (and columns) into two equal sized sections $\{1, \ldots, l\}$ and $\{l + 1, \ldots, P\}$. We choose $l = \lceil P/2 \rceil$ as our splitting value. The quadrants are numbered from one to four in counter-clockwise order starting at the upper right quadrant.

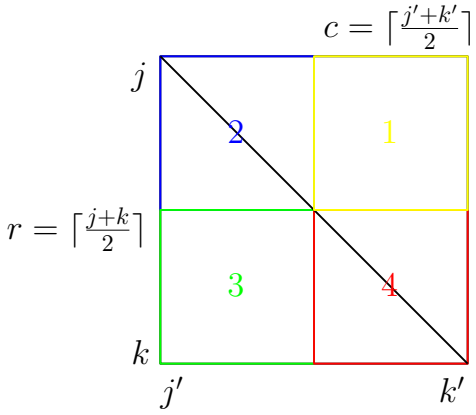


Figure 5.3: Splitting process for the undirected $G(n, m)$ generator for a set of chunk rows $\{j, \ldots, k\}$ and columns $\{j', \ldots, k'\}$. Quadrants are colored individually and are enumerated from top right in counter-clockwise order.

We then compute the number of elements of the adjacency matrix within each of the quadrants. Since we are only concerned with the lower triangular adjacency matrix, there are two different types of quadrants: triangular and rectangular. The second and fourth quadrant are triangular matrices with l and $P - l$ rows (and columns) respectively. The number of edges (without self-loops) in a $n' \times n'$ triangular submatrix of chunks can be computed by using the triangular number $\frac{n'(n'-1)}{2}$ [29]. In contrast, the third quadrant is a rectangular $l \times (P - l)$ matrix. The first quadrant is also a rectangular matrix but never contains any edges so it can be omitted from further analysis. The number of edges in $n' \times m'$ rectangular submatrix of chunks is given by $n' \cdot m'$.

We then generate a set of three hypergeometric random deviates to determine the number of samples (edges) in each quadrant. The first of these deviates is used to determine the number of samples in the upper half of the adjacency matrix. We then use this first deviate to compute the number of samples in the second and third quadrant. The fourth quadrant can be handled by subtracting the random deviate for the third quadrant from the number of samples in the lower half of the adjacency matrix.

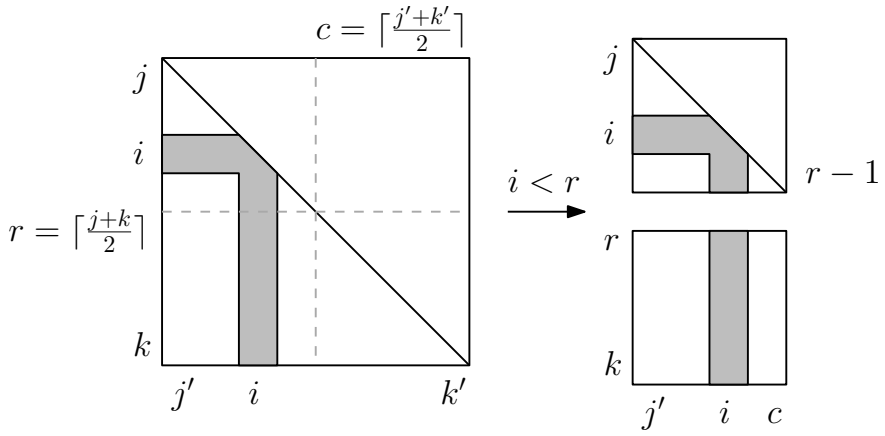


Figure 5.4: Recursion for the undirected $G(n, m)$ for a set of chunk rows $\{j, \ldots, k\}$ and columns $\{j', \ldots, k'\}$. The chunk row and column for processor i are highlighted. Since the chunk row is in the upper half of the triangular adjacency matrix, recursion only follows the second and third quadrant.

To decide which quadrants to handle recursively, each processor then checks if its assigned row of chunks is in the upper or lower half of the adjacency matrix. If it is in the upper half, we can be certain that its column of chunks will be in the left half of the adjacency matrix. It therefore only has to perform recursion on the second and third quadrants as illustrated in Figure 5.4. Additionally, we can be sure that the third quadrant only contains a column of chunks. Otherwise we follow the subtrees for the third and fourth quadrant. This time the third quadrant only contains a row of chunks. Again, we stop the recursion if only a single chunk is remaining. The resulting recursion trees has at most $\lceil \log P \rceil$ levels and size $\frac{4P^2-1}{3}$.

Depending on the type of chunk we then determine the resulting edges using a local sampling algorithm. The pseudocode for the undirected $G(n, m)$ generator is provided in Algorithm 3 and Algorithm 4.

Triangular submatrix. To generate edges for a $n' \times n'$ triangular submatrix, we first compute the total number of edges M using its triangular number. To determine the source vertex of a sampled element $s \in \{1, \ldots, M\}$ we first need to compute the corresponding row in the adjacency matrix. This can be done by finding the largest number x whose triangular number $\frac{x(x+1)}{2}$ is smaller than or equal to $s - 1$. We can find this number by computing and rounding the triangular root of $s-1$, which is defined as $\frac{\sqrt{8(s-1)+1}-1}{2}$. Thus, the source vertex of our edge is $v_s = \lfloor \frac{\sqrt{8(s-1)+1}-1}{2} \rfloor$. Afterwards, we add N_{i-1} as an offset to get the correct global vertex id. The target vertex can then be computed by subtracting the triangular number of the source vertex from the sample, i.e. $v_t = (s-1) - \frac{v_s(v_s+1)}{2}$. Again, we add the offset N_{i-1}.

Rectangular submatrix. Generating edges in a $n' \times m'$ rectangular submatrix is very similar to sampling edges in the directed case. The source vertex of a sampled element $s \in \{1, \ldots, N\}$ ($N = n' \cdot m'$) is $v_s = \lfloor \frac{s-1}{n'} \rfloor + N_{i-1}$. Analogously, the target vertex can be computed using $v_t = (s-1) \mod m' + N_{i-1}$

Lemma 5.2.2. *The undirected $G(n, m)$ generator runs in time $\mathcal{O}(\frac{m+n}{P} + P)$ with high probability*[1].

Proof. Each processor P has to generate a total of P chunks consisting of a single triangular submatrix and $P - 1$ rectangular submatrices. Additionally, each edge $\{i, j\}$ has to be generated twice (except when P equals one), once by the processor that is assigned vertex i, and once by the processor that is assigned vertex j. Thus, we have to sample a total of $2m$ edges. We first show that the time spent for recursion is linear in P. We then follow the arguments presented in the proof of Theorem 4.2.1 and show that the time complexity for sampling the $2m$ edges is $\mathcal{O}(m/P)$ with high probability.

At every level during the recursion we have to follow the subtrees for two of the four quadrants. This means that we have to process 2^l quadrants at level l. At every level we only need to split the quadrants and therefore compute three hypergeometric random deviates. Thus, the time spent at every level only takes expected linear time as shown in Theorem 4.2.1. Since there are at most $\lceil \log P \rceil$ levels until each processor reaches its P chunks, the total time spent on recursion is $\sum_{i=0}^{\log P} 2^i = 2(P-1) = \mathcal{O}(P)$ with high probability.

Following the proof of Theorem 4.2.1, we can use Chernoff bounds (Lemma 2.1.1) to show that the total number of samples (edges) that is assigned to any processor will be in $\mathcal{O}(m/P)$ with high probability. In turn, sampling these elements can be performed in expected linear time with high probability. Thus, the directed $G(n, m)$ generator has a running time of $\mathcal{O}(m/P + P)$ with high probability. ∎

5.2.3 Directed $G(n, p)$ Generator

We now discuss how to adapt the directed $G(n, m)$ generator for the $G(n, p)$ model. The main difference with the $G(n, m)$ model is that we do not have to recursively compute hypergeometric random deviates in order to derive the correct number of edges for each chunk. Since the distribution of vertices for each individual chunk is predetermined, we can instead generate a binomial random deviate m' (e.g. $m' \sim B(m = n(n-1), \frac{m}{P} \cdot p)$) to get the correct sample size. For this purpose, we let M_i denote the last edge in the range of the ith chunk. Afterwards, we continue to perform the sampling and generation of edges as seen in the $G(n, m)$ generator.

To seed the binomial random generator we can compute a hash value based on the number of the given chunk. Since we can omit the recursion and a binomial random deviate can be generated in expected constant time, the resulting algorithm runs in time $\mathcal{O}(m/P)$

[1] i.e. with probability at least $1 - p^{-c}$ for any constant c

with high probability. The pseudocode for the directed $G(n, p)$ generator is presented in Algorithm 5.

5.2.4 Undirected $G(n, p)$ Generator

As for the directed case, the difference when using the undirected $G(n, p)$ generator lies in the fact that we do not have to compute hypergeometric random deviates. Instead, we can skip recursion and directly compute the number of edges to sample by generating binomial random deviates. Since the distribution of chunks to processors is again predetermined, we can compute the number of vertices and therefore potential edges within each chunk. To be more specific, we can compute the position of each chunk within the $P \times P$ chunk matrix, and thus the number of potential edges, by using its triangular root. We then seed a binomial random generator with the chunk row i and column j to generate the number of edges m' (e.g. $m' \sim B(m, \frac{n(n-1)}{2P} \cdot p)$) and sample edges locally. However, because we still have to iterate over P chunks for ever processor, the running time of this algorithm is $\mathcal{O}(m/P + P)$ with high probability as with the $G(n, m)$ generator. The pseudocode for the undirected $G(n, p)$ generator is presented in Algorithm 6.

5.3 Random Geometric Generator

We now discuss the generators for the random geometric model (RGG) introduced in Section 2.2.2. To be more specific, we provide generators for the two- and three-dimensional unit square model. We begin by discussing our two dimensional generator as well as its parallelization. We then adapt the resulting algorithm for the three dimensional case. Additionally, we provide proofs on the time complexity and scalability of both generators.

5.3.1 2D Generator

Generating a two dimensional random geometric graph can be done naively in $\Theta(n^2)$ time. To do so, we first distribute the set of n vertices uniformly at random in the two dimensional unit square $[0, 1)^2$. The position (x, y) of each vertex is determined by computing two uniform random variables from the unit interval $[0, 1)$. This can be done using a pseudorandom number generator, e.g. a Mersenne Twister [48]. Afterwards, we compare all pairs of vertices and check if their distance is smaller than or equal to a given radius $r > 0$.

We now discuss several improvements to reduce the running time of the naive algorithm. To reduce the number of comparisons that need to be performed between vertices, we introduce a two dimensional grid data structure similar to the one used by Holtgrewe et al. [35]. The grid data structure partitions the unit square into equal sized cells. The partition is performed in such a way that each individual cell has a side length of at least r. In particular, we have $g = \lfloor 1/r \rfloor$ grid cells in each row and column, as seen in Figure 5.5a. Thus, the total number of cells is g^2. Cells are numbered from 0 to $g^2 - 1$ row-wise from top-left to bottom-right. We then store each vertex in the corresponding grid cell, which we can determine in constant time using simple arithmetic operations. Afterwards, we compute a prefix sum over the number of vertices in each cell to retrieve the correct global vertex ids. If we are not interested in consecutive global vertex ids, we can omit this step and only determine the number of vertices for the local chunk.

Since the vertices are uniformly distributed within the unit square, we have a n balls into g^2 bins situation. The expected number of vertices in each cell therefore is $\mathcal{O}(nr^2)$. By using Chernoff bounds (Lemma 2.1.1), we can show that this happens with high probability. Additionally, we are able to show that by using this grid data structure the number of vertex comparisons that need to be performed is $\mathcal{O}(m)$ with high probability. Figure 5.5b shows an example for a two dimensional RGG graph with 256 vertices and a radius of 0.1 using our grid data structure.

Lemma 5.3.1. *The number of vertex comparisons for generating a RGG graph using a two dimensional grid data structure with side length r is $\mathcal{O}(m)$ with high probability.*

Proof. To compute the edges of our graph, we start by iterating over the individual cells. Since each cell has a side length of at least r, we only have to perform vertex comparison with the vertices stored in neighboring cells. In the two dimensional case, there are at most eight neighbors (and the cell itself) that we have to consider. Since each cell

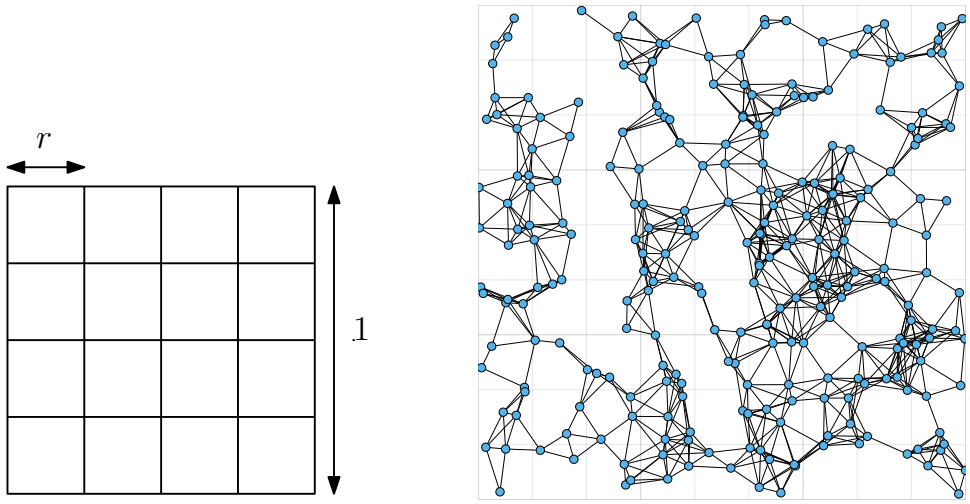


Figure 5.5: (a) Partitioning of the unit plane into set of cells with radius r. (b) Example of a two dimensional random geometric graph with 256 vertices and a radius of 0.1. The gray lines represent our grid data structure.

contains $\mathcal{O}(nr^2)$ vertices with high probability the number of comparison to perform between neighboring cells is $\mathcal{O}((nr^2)^2)$. As explained above, there are $\lfloor 1/r \rfloor^2$ cells in total. We therefore get $\mathcal{O}(\frac{1}{r^2} \cdot (nr^2)^2) = \mathcal{O}(n^2r^2))$ vertex comparisons with high probability. The expected average degree of a vertex outside the border in the RGG model is $n\pi r^2$, as presented in Section 2.2.2. Vertices on the border have fewer edges than the ones in the center. Therefore, we can omit them from our analysis. We then have to generate $n \cdot n\pi r^2 = \mathcal{O}(n^2r^2) = \mathcal{O}(m)$ edges in total. Thus the total expected number of vertex comparisons is $\mathcal{O}(m)$ with high probability. ∎

Using this approach, each undirected edge $\{i, j\}$ that runs between two different cells, is generated twice. Thus, we are able to reduce the number of vertex comparisons by a constant factor if we only compare a cell to cells that have a higher id.

We now discuss how to parallelize our grid approach in a communication agnostic way. We therefore assume we are given a set of $P = p^2$ processors. In addition to P, each processor also knows its own id $0 \leq i < P$.

Our goal is to distribute the vertices between processors in such a way that the amount of recomputations is minimized. To this end, we again use the notion of chunks. In the case of the two dimensional RGG generator a chunk represents a rectangular section of the unit square. We then partition the unit square into P disjoint chunks and assign one of them

to each processor. To be more specific, processor i is assigned the chunk that spans the rectangle

$$\Big[(i \mod p)/p, (i \mod p)/p + \frac{1}{p}\Big) \times \Big[\lfloor \frac{i}{p} \rfloor /p, \lfloor \frac{i}{p} \rfloor /p + \frac{1}{p}\Big).$$

Each processor is then responsible for generating the vertices in its chunk, as well as all their incident edges. For this purpose, we again use a divide-and-conquer approach similar to the undirected $G(n, m)$ generator.

Recursion begins by splitting the chunks $\{j, \ldots, k\}$ and $\{j', \ldots, k'\}$ in each dimension into equally-sized subsets. The splitting values are $l = \lfloor \frac{j+k}{2} \rfloor$ and $l' = \lfloor \frac{j'+k'}{2} \rfloor$, respectively. This results in a partitioning of the underlying space into quadrants. The possibility for each vertex to be assigned to a individual quadrant is the ratio of the area of the quadrant to the area of the complete rectangle. To this end, we generate three binomial random deviates to compute the number of vertices within each of the quadrants. Since we have to assign global vertex ids to each chunk, each processor then continues recursion for all subtrees. We repeat this procedure until each processor is left with the complete set of chunks and knows the correct number of vertices for each of them. Note that the resulting recursion tree has at most $\lceil \log P \rceil$ levels and size $\frac{4P-1}{3}$.

In the case of non-local chunks, we simply store the number of vertices to compute a prefix sum for the global vertex ids. For the local chunk of a processor, we generate the vertices and assigning them to correct grid cells. This can be done using the same approach as with the sequential version of the algorithm. Additionally, since we want each processor to generate *all* incident edges for its local vertices, we have to make sure that the cells of neighboring chunks that are within the radius of local vertices are also generated. Because each cell has a side length of at least r and each chunk contains roughly $\lfloor \frac{1}{pr} \rfloor^2$ cells, there are at most $4 \cdot \lfloor \frac{1}{pr} \rfloor + 4$ of these cells. Due to the communication agnostic design of our algorithm, the generation of these cells is done through recomputations. We therefore repeat the vertex generation process for the eight neighboring chunks, as seen in Figure 5.6a.

To reduce the memory overhead of our program, each processor only stores the vertices of cells that are adjacent to its local chunk. The pseudocode for the two dimensional RGG generator is presented in Algorithm 7. An example of the subgraph that a single processor generates is given in Figure 5.6b

Lemma 5.3.2. *The two dimensional RGG generator has a running time of $\mathcal{O}(\frac{m+n}{P} + P)$ with high probability.*

Proof. We first show that our divide-and-conquer algorithm assigns each processor n/P vertices in time $\mathcal{O}(P)$ with high probability. As stated above, the recursion tree has at most $\lceil \log P \rceil$ levels and size $\mathcal{O}(P)$. At each level of the recursion we generate three binomial

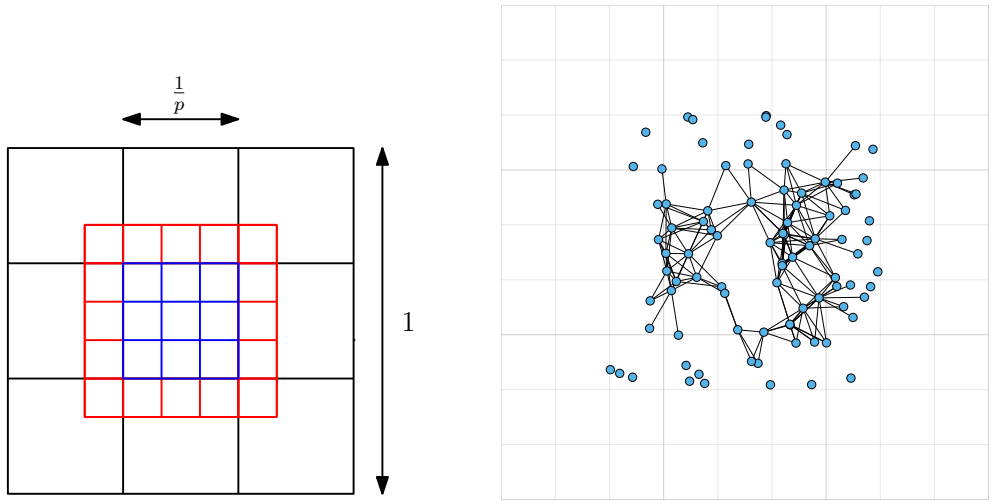


Figure 5.6: (a) Unit square and the local chunk (blue) of a processor partitioned into cells including neighboring cells (red) that are computed redundantly. Each chunk has size $\frac{1}{p}$ and each cell has size r. (b) Section of a two dimensional random geometric graph with 256 vertices and a radius of 0.1 that is generated on a single processor. The section consists of a single chunk and all its surrounding grid cells. Edges are only added if at least one endpoint is a local vertex.

random deviates, which takes expected constant time. We then use the binomial random deviates to assign each quadrant its corresponding number of vertices. Since the tree has a size of $\mathcal{O}(P)$, we therefore have a running time of $\mathcal{O}(P)$ with high probability to determine the number of vertices for each chunk. The result of this process is a uniform distribution of vertices into chunks. As with the sequential algorithm, we therefore can simplify the analysis on the number of vertices that is distributed to each chunk to a n balls into P bins situation. Thus, we have an expected number of $\mathcal{O}(n/P)$ vertices per processors. Again, by using Chernoff bounds (Lemma 2.1.1) we can show that this happens with high probability. Generating these vertices and putting them into the correct grid cell can be done in linear time in the number of points per chunk.

Generating the vertices for the neighbors of the local chunk can also be done in $\mathcal{O}(n/P)$ time with high probability, since each chunk only has a constant number of neighbors. Thus, generating and partitioning all vertices necessary for each individual processor has a time complexity of $\mathcal{O}(n/P + P)$ with high probability.

Following the proof of Lemma 5.3.1, we are able to show that the number of edges that each processor has to generate is $\mathcal{O}(n/P \cdot n\pi r^2)$ with high probability. This is equal to $\mathcal{O}(m/P)$ edges per processor with high probability. The running time bound for the RGG generator then follows by combining the time required to generate the vertices as well as

edges. If we do not require consecutive vertex ids, the running time can be decreased to $\mathcal{O}(\frac{m+n}{P} + \log P)$ with high probability. ∎

For a sufficiently large number of vertices and a fixed expected number of edges, the additional workload introduced through parallelization is negligible. To see this, we compare the workload per processor in the sequential and parallel case. In the sequential case, we only have a single chunk of n vertices. In the parallel case, each processor additionally recomputes two rows and columns of cells, as well as four single cells, for its neighbors. This adds up to an additional amount of $4 \cdot \frac{nr}{p} + 4 \cdot nr^2$ vertices with high probability. Since the number of comparisons for each vertex remains the same in both cases, we can use the amount of additional vertices as a measurement for the additional workload. Therefore, we divide the number of vertices generated in the parallel case by the number of vertices generated in the sequential case. This results in a factor of $1 + 4rp + 4r^2p^2$.

Since we keep the expected number of edges fixed, the radius for an increasing amount of vertices decreases. To be more specific, the average degree of a vertex outside the border is $n\pi r^2$. Thus, the expected number of edges can be bound by $m = n^2\pi r^2$. If we then keep m fixed, we get $\lim_{n\to\infty} r = \lim_{n\to\infty} \frac{\sqrt{m\pi}}{n} = 0$. In turn, $\lim_{n\to\infty} 1 + 4rp + 4r^2p^2 = 1$. This shows that the efficiency of our generator increases depending on the sparsity of the graph.

5.3.2 3D Generator

We now discuss how to generalize our RGG generator from two to three dimensions. Again, generating a three dimensional random geometric graph can be done naively in $\Theta(n^2)$ time by comparing all pairs of vertices. Vertices are connected if their distance is less or equal to $r > 0$.

To improve this bound we build a three dimensional grid data structure to partition the unit cube $[0, 1)^3$. Each dimension of this grid spans $g = \lfloor 1/r \rfloor$ cells, resulting in g^3 cells in total. We assign each cell a triple (x, y, z) representing its grid-coordinate in each dimension. We then number the triples from 0 to $g^3 - 1$ by their lexicographic ordering, as seen in Figure 5.7. Similar to the two dimensional case, we can use a balls into bins argument to show that the number of vertices in each grid cell is $\mathcal{O}(nr^3)$ with high probability. Figure 5.8 shows an example for a three dimensional RGG graph with 256 vertices and a radius of 0.15 using our grid data structure.

Lemma 5.3.3. *The number of vertex comparisons for generating a RGG graph using a three dimensional grid data structure with side length r is $\mathcal{O}(m)$ with high probability.*

Proof. By having cells with a side length of r, we can limit the vertex comparison to the 26 neighbors of each cell. Following the balls into bins argument, the expected number of vertices in each cell is $\mathcal{O}(nr^3)$ with high probability. Therefore, computing the edges between a pair of cells takes expected time $\mathcal{O}((nr^3)^2)$. As there are $\lfloor 1/r \rfloor^3$ cells in total, we need to perform $\mathcal{O}(n^2r^3)$ vertex comparisons with high probability to compute the edges between all eligible pairs of cells.

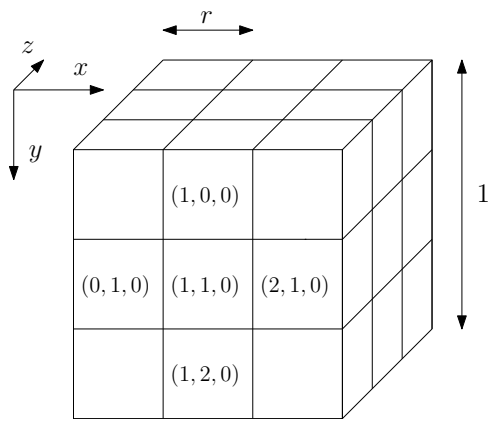


Figure 5.7: Unit cube and the resulting grid data structure with cell size r.

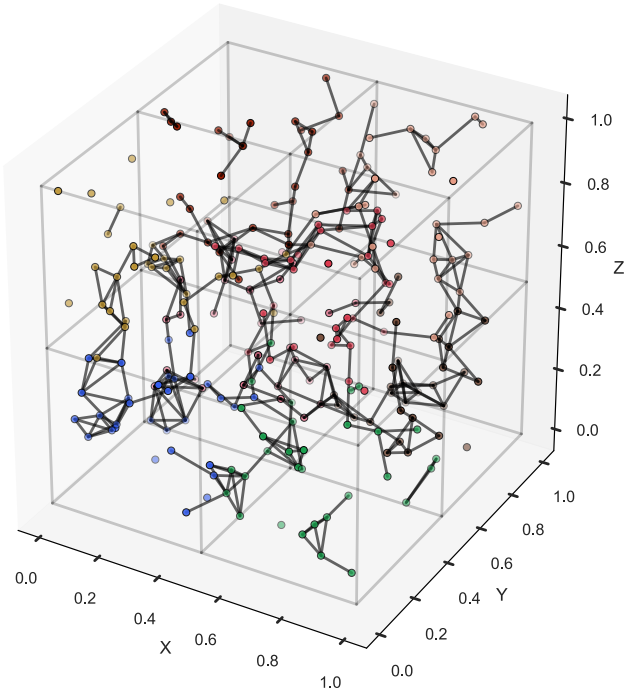


Figure 5.8: Three dimensional random geometric graph with 256 vertices and a radius of 0.15. The grid data structure is omitted for visual clarity.

By an extension of the two dimensional case, the expected average degree of a vertex outside the border in the unit cube is $\frac{4}{3}n\pi r^3$. Vertices that lie on the border can be omitted from the analysis, as they have fewer edges than the ones in the center. This results in an

expected number of edges of $n \cdot \frac{4}{3}n\pi r^3 = \mathcal{O}(n^2 r^3) = \mathcal{O}(m)$. Therefore, the expected number of vertex comparisons is linear in m with high probability. ∎

To parallelize this algorithm we extend our divide-and-conquer approach to three dimensions. We therefore assume we are given a set of $P = p^3$ processors. Each processor is assigned an equal sized chunk that spans a cuboid. To be more specific, processor i is assigned the cuboid that spans

$$\begin{aligned}&\Big[(i \mod p)/p, (i \mod p)/p + \frac{1}{p}\Big)\\ \times &\Big[\Big\lfloor\frac{i}{p}\Big\rfloor/p, \Big\lfloor\frac{i}{p}\Big\rfloor/p + \frac{1}{p}\Big)\\ \times &\Big[\Big\lfloor\frac{i}{p^2}\Big\rfloor/p, \Big\lfloor\frac{i}{p^2}\Big\rfloor/p + \frac{1}{p}\Big).\end{aligned}$$

Recursion now includes three sets of chunks that equally partition the current cuboid into octants. The possibility for a vertex to be assigned a individual octant is the ratio of the area of the octant to the area of the remaining cuboid. We then determine the number of vertices in each octant, by generating seven binomial random deviates: one for the x-dimension, two for the y-dimension and four for the z-dimension. By comparing its id i with the splitting values of the partitioning, each processor then follows the subtree for one octant. The resulting recursion tree has at most $\lceil \log P \rceil$ levels and size $\frac{8P-1}{7}$.

The base case consists of sampling the vertices using three uniform random deviates for each vertex. Assigning them to the correct grid cell can be done in constant time using simple arithmetic. Additionally, we have to redundantly compute the vertices for all neighboring cells using our divide-and-conquer approach. There are at most $6 \cdot \lfloor\frac{1}{pr}\rfloor^2 + 12 \cdot \lfloor\frac{1}{pr}\rfloor + 8$ of these cells.

Lemma 5.3.4. *The three dimensional RGG generator has a running time of $\mathcal{O}(\frac{m+n}{P} + P)$ with high probability.*

Proof. We can use the same arguments as in the proof of the two dimensional RGG generator, to show that each processor is assigned $\mathcal{O}(n/P)$ vertices in time $\mathcal{O}(P)$ with high probability. In turn, we can generate these vertices and assign them to the corresponding grid cells in time $\mathcal{O}(n/P)$. Thus, the total time spent on generating the local chunk is $\mathcal{O}(n/P + P)$ with high probability.

Since each chunk only has a constant number of neighbors, the same bound applies for recomputing the neighboring cells using the divide-and-conquer algorithm. The time needed for generating the local edges stays the same as for the sequential case. Thus, the three dimensional RGG generator runs in time $\mathcal{O}(\frac{m+n}{P} + P)$ with high probability. If we do not require consecutive vertex ids, the running time can be decreased to $\mathcal{O}(\frac{m+n}{P} + \log P)$ with high probability. ∎

Again, as we increase the number of vertices to a sufficiently large value and keep the expected number of edges fixed, the additional workload introduced by the parallelization is negligible. In the sequential case, we only have a single chunk consisting of n vertices. In the parallel case, each processor additionally recomputes six sides, twelve rows and columns, as wells as eight single cells, for its neighbors. Therefore, $6 \cdot \frac{nr}{p^2} + 12 \cdot \frac{nr^2}{p} + 8 \cdot nr^3$ additional vertices have to be generated with high probability. Thus, the quotient of the parallel workload and the sequential workload is $1 + 8 \cdot r^3p^3 + 12 \cdot r^2p^2 + 6 \cdot rp$. As with the two dimensional case, we can see that this quotient approaches one if $\lim_{n\to\infty} r = 0$.

5.4 Random Hyperbolic Generator

We now present our generator for the random hyperbolic model discussed in Section 2.2.3. Our generator is designed for the two dimensional native representation. Thus, each vertex is modeled as a point (ϕ, r) in polar coordinates on a disk of radius R. We only consider the deterministic model where vertices are connected iff their hyperbolic distance is less or equal to the target radius R. We describe two different versions of our algorithm, one that requires communication and one that is fully communication agnostic. In addition, we provide proofs for the time complexity of the generators in both the sequential and parallel case.

5.4.1 Sequential Generator

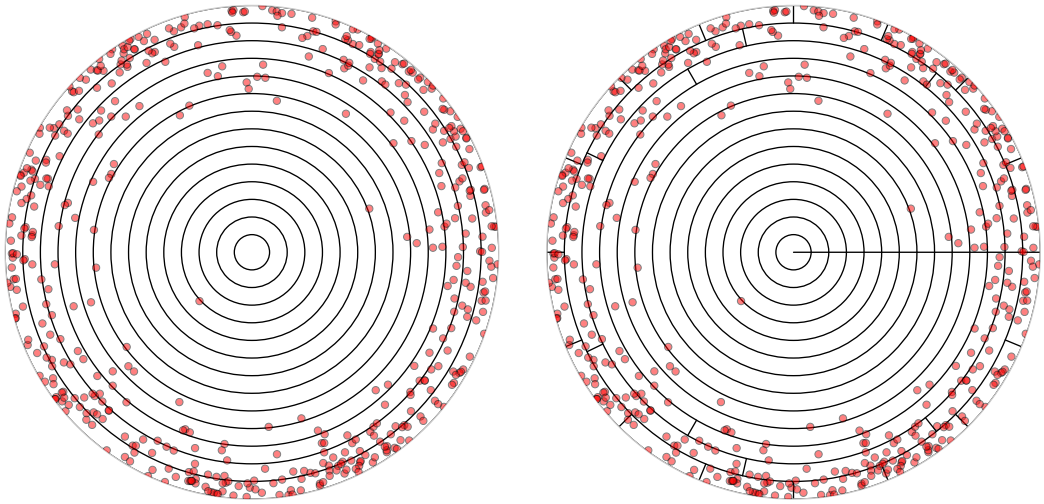
Our generator assumes we are given the number of vertices n, their average degree $\bar{k}$, as well as a power-law exponent γ. We then use γ to compute the growth parameter $\alpha = \frac{\gamma-1}{2}$. To compute the target radius R, we use the approximation presented by von Looz et al. [73]. Afterwards, we can naively create a random hyperbolic graph in $\Theta(n^2)$ by comparing all pairs of vertices, checking if their distance is less than the target radius R, and adding edges accordingly. As with the RGG generator, we can improve this bounded by introducing a partitioning of the hyperbolic plane [20, 73, 74].

Grid Data Structure. We begin by dividing the hyperbolic plane into $\lfloor \frac{\alpha R}{\ln(2)} \rfloor$ concentric annuli, similar to the approach of von Looz et al. [74]. Since R is proportional to $\ln(n)$ this results in $\mathcal{O}(\ln(n))$ annuli [44]. We number the annuli from 0 to $\lfloor \frac{\alpha R}{\ln(2)} \rfloor - 1$ in increasing order starting from the innermost annulus. Each annulus is defined by two radii r_i and r_{i+1}, with $r_0 = 0$ and $r_{\text{max}} = R$. The annulus then contains the area of the hyperbolic plane between these two radii, i.e. a vertex (ϕ, r) is contained within annulus i iff $r_i \leq r < r_{i+1}$. Therefore, the set of annuli fully partitions the hyperbolic plane. The radial boundaries for the annuli are chosen in such a way that they are equidistant, e.g. annulus i is assigned the radii $r_i = i \cdot \lfloor \frac{\ln(2)}{\alpha} \rfloor$ and $r_{i+1} = (i+1) \cdot \lfloor \frac{\ln(2)}{\alpha} \rfloor$.

To determine the number of vertices in each annulus, we compute a multinomial random deviate with $\lfloor \frac{\alpha R}{\ln(2)} \rfloor$ outcomes. To compute a multinomial random deviate for k outcomes, we can compute $k-1$ dependent binomial random deviates. As for the other generators, we use pseudorandomization based on a hash functions to seed the multinomial random generator. The probability that an individual vertex is assigned to annulus i is given by

$$p_i = \int_{r_i}^{r_{i+1}} f(r)dr = \frac{\cosh(\alpha \cdot r_{i+1}) - \cosh(\alpha \cdot r_i)}{\cosh(R) - 1}. \tag{5.4.1}$$

As discussed in Section 2.2.3, $f(r)$ is the probability density function for the radial coordinate of a vertex. By computing the integral of $f(r)$ over the interval $[r_i, r_{i+1}]$, we thus get



(a) (b)

Figure 5.9: Partitioning of the hyperbolic plane into set of equidistant annuli (a) as well as cells (b). The number of vertices is $n = 512$ with an average degree of $\bar{k} = 4$ and a power-law exponent of $\gamma = 2.6$. The expected number of vertices per cell for (b) is set to 2^4.

the probability that a vertex is assigned to annulus i. Because the vertices are distributed multinomially, the expected value for the number of vertices in annulus i is $n \cdot p_i$.

To actually compute the vertices within an annulus i, we need to generate two uniform random variates for each vertex (ϕ_v, r_v). This is done using a Mersenne Twister (e.g. [48]) that is initialized with the hash value of the annulus id. We use the first variate to draw ϕ_v uniformly at random from the interval $[0, 2\pi)$. The second variate is used to draw r_v with probability density $f(r)$ from the interval $[r_i, r_{i+1}]$. Figure 5.9a shows an example of the resulting partitioning and vertex distribution.

We then continue to further partition each annulus into a set of angular cells. Each cell within an annulus occupies an equal portion of the hyperbolic plane and is defined by two angular boundaries ϕ_j and ϕ_{j+1} with $\phi_0 = 0$ and $\phi_{\text{max}} = 2\pi$. The number of cells is chosen in such a way that each cell contains an expected constant number of vertices k with high probability. Thus, we let number of cells in annulus i be $\lceil \frac{i \cdot p_i}{k} \rceil$. In turn, $\phi_j = j \cdot 2\pi / \lceil \frac{i \cdot p_i}{k} \rceil$. Cells are numbered from 0 to $\lceil \frac{i \cdot p_i}{k} \rceil - 1$ by increasing order of their smaller angular boundary. During the computation of the vertices within an annulus, we then assign each vertex v to its respective grid cell $j = \left\lfloor \phi_v / \lceil \frac{i \cdot p_i}{k} \rceil \right\rfloor$. Figure 5.9b shows the resulting partitioning of vertices in the hyperbolic plane into cells and annuli. In order to ensure we have the correct vertex ids when processing cells of our grid data structure, we make use of two prefix sums. One prefix sum is computed over the number of vertices within each annulus, and a second one is computed over the vertices in each cell. Thus, we are able to assign a set of global vertex ids to each cell.

Edge Generation. We now describe how we use our partitioning to efficiently reduce the number of vertex comparisons. For this purpose, we begin by iterating over the annuli in increasing order, starting from the innermost annulus. Within each annulus, we further iterate over the cells by increasing angular coordinate. We then perform a search query for each vertex in the corresponding cell to gather its incident edges.

The query begins by determining how far the angular coordinate of a potential neighbor $u = (\phi_u, r_u)$ is allowed to deviate from the angular coordinate of our query vertex $v = (\phi_v, r_u)$. If we assume that u lies in annulus i with radial boundaries r_i and r_{i+1}, we can use the distance inequality

$$\cosh(R) \geq \cosh(r_v)\cosh(r_i) - \sinh(r_v)\sinh(r_i)\cos(|\phi_u - \phi_v|) \Leftrightarrow \tag{5.4.2}$$

$$|\phi_u - \phi_v| \leq \cos^{-1}\Big(\frac{\cosh(r_v)\cosh(r_i) - \cosh(R)}{\sinh(r_v)\sinh(r_i)}\Big) \tag{5.4.3}$$

to determine this deviation. We then gather the set of cells that lie within the resulting boundary coordinates. To do so, we start from the cell that intersects the angular coordinate of our query vertex and then continue in both angular directions until we find a cell that lies outside the boundary. For each cell that we encounter, we iterate over the corresponding vertices and perform distance comparisons with our query vertex. Respectively, we add an edge between v and u iff $\text{dist}_H(v, u) \leq R$. We then continue this process with the next annulus, until all annuli have been processed.

Note that we only have to perform queries in one radial direction to gather all edges, since edges can be found from both direction. Therefore, we only consider annuli that have an equal or larger radial boundary than our starting annulus.

To avoid duplicate edges within an annulus, we skip cells in the same annulus that have a higher id than the current one. Additionally, if two vertices are within the same cell, we only add an edge if the angular coordinate of the first vertex is smaller than that of the second vertex. If they have the same angular coordinate, we only add an edge if the radial coordinate of the first vertex is smaller than that of the second vertex. The pseudocode for the sequential RHG generator is given in Algorithm 8.

Lemma 5.4.1. *The expected time complexity of the sequential RHG generator for n vertices with an average degree of $\bar{k}$ and a power-law exponent $\gamma > 2$ is $\mathcal{O}(n + m)$.*

Proof. We begin by giving a bound on the number of vertex comparisons that we have to perform. For each vertex we have to perform a single query outwards through all annuli with a larger radius. Within each annulus we step through the set of cells that lie within the target radius of our query vertex. We now show that the time required for each query vertex with radius r is linear in the average degree

$$\bar{k}(r) = n\big(\frac{2}{\pi}\xi e^{-\zeta r/2} - (\frac{2}{\pi} - 1)e^{-\alpha r}\big)$$

for a vertex with radius r and $\xi = (\alpha/\zeta)/(\alpha/\zeta - 1/2))$ [44]. Note that we can assume that $\zeta = 1$ while retaining all degrees of freedom [74]. Additionally, since we assume that $\gamma > 2$ and therefore $\alpha > 1/2$, the average degree is equal to $\bar{k}(r) = n \cdot \frac{2}{\pi} \cdot \xi \cdot e^{-r/2}$ [44].

We first compute the expected number of vertices in annulus i, i.e. the number of vertices with a radius $r_i \leq r < r_{i+1}$. Note that the total number of annuli is $\frac{\alpha R}{\ln(2)}$ and annulus i is bounded by the radius $r_i = i \cdot \frac{\ln(2)}{\alpha}$. The probability that a vertex v has radius $r_v \leq r$ is given by $f(r) = \alpha \frac{\sinh(\alpha r)}{\cosh(\alpha R)-1} \approx \alpha e^{\alpha(r-R)}$ [44]. We then get the probability of a vertex being assigned to annulus i by

$$\int_{r_i}^{r_{i+1}} f(r)dr \approx \int_{r_i}^{r_{i+1}} \alpha e^{\alpha(r-R)}dr = \frac{e^{\alpha r_{i+1}} - e^{\alpha r_i}}{e^{\alpha R}}.$$

Therefore, the expected number of vertices in annulus i is $n_i = n \cdot \frac{e^{\alpha r_{i+1}} - e^{\alpha r_i}}{e^{\alpha R}}$. In particular, the number of vertices in the innermost annulus is $n_0 = \frac{n}{e^{\alpha R}}$. If we now compute the ratio of the expected number of vertices in annulus $i+1$ and annulus i, we get a growth factor for the number of vertices in each annulus:

$$\begin{aligned}
\frac{n_{i+1}}{n_i} &= \frac{e^{\alpha r_{i+2}} - e^{\alpha r_{i+1}}}{e^{\alpha r_{i+1}} - e^{\alpha r_i}} \\
&\overset{r_i = i \cdot \frac{\ln(2)}{\alpha}}{=} \frac{e^{\alpha r_{i+2}} - e^{\alpha r_{i+1}}}{e^{\alpha r_{i+1}} - e^{\alpha r_i}} \\
&= \frac{e^{(i+1)\ln(2)}}{e^{i\ln(2)}} \\
&= 2.
\end{aligned}$$

As a result, the expected number of vertices in annulus i can be written as $n_i = n_0 \cdot 2^i$.

Next, we bound the number of vertices that we have to examine during a query for a vertex with radius r. As mentioned previously, we compute two boundary angles for each annulus i to determine the set of cells that we have to look at. We do so by using the distance inequality (5.4.2). For large r, r_i, and R we can approximate the difference between the angles of our vertex and annulus with $2e^{(R-r_i-r)/2}$ [44]. In order to compute the number of points that we examine for each annulus, we then multiply this approximation by two (one boundary for each direction) and divide it by 2π. This gives us the probability that a point in annulus i is within our query. We then multiply it by the expected number of points $n_i = n_0 \cdot 2^i$ in annulus i. In turn, this gives us the expected number of points that we have to examine in annulus i. The additional number of points that we have to check due to our grid data structure is constant, since each cell only contains a constant number of points.

To determine the total number of points that we process for a vertex with radius r, we then sum up the expected number of points over all annuli.

$$\begin{aligned}
\sum_{i=0}^{\frac{\alpha R}{\ln(2)}} n_0 \cdot 2^i \cdot \frac{2(2e^{(R-r-r_i)/2})}{2\pi} &= n_0 \cdot \frac{2}{\pi} \cdot \sum_{i=0}^{\frac{\alpha R}{\ln(2)}} 2^i \cdot e^{(R-r-r_i)/2} \\
&= n_0 \cdot \frac{2}{\pi} \cdot \frac{1}{2^{1/(2\alpha)}-2} \cdot e^{-r/2} \cdot (2^{1/(2\alpha)} e^{R/2} - 2e^{\alpha R}) \\
&= n \cdot \frac{2}{\pi} \cdot \frac{1}{2^{1/(2\alpha)}-2} \cdot e^{-r/2-\alpha R} \cdot (2^{1/(2\alpha)} e^{R/2} - 2e^{\alpha R}) \\
&= n \cdot \frac{2}{\pi} \cdot \frac{1}{2^{1/(2\alpha)}-2} \cdot (2^{1/(2\alpha)} e^{-r/2-\alpha R+R/2} - 2e^{-r/2})
\end{aligned}$$

Since we assume that $\gamma > 2$ and therefore $\alpha > 1/2$ this equation can be approximated by

$$\sum_{i=0}^{\frac{\alpha R}{\ln(2)}} n_0 \cdot 2^i \cdot \frac{2(2e^{(R-r-r_i)/2})}{2\pi} \approx n \cdot \frac{2}{\pi} \cdot \frac{2}{2-2^{1/(2\alpha)}} \cdot e^{-r/2}.$$

Therefore, the number of vertices that we examine for each query only differs by a constant factor from the average degree $\bar{k}(r) = n \cdot \frac{2}{\pi} \cdot \xi \cdot e^{-r/2}$ presented by Krioukov et al. [44]. In turn, the total amount of vertex comparisons that we perform over all vertices is in $\mathcal{O}(m)$.

Next, we show that the expected time needed to step through the set of annuli for all vertices is bounded by $\mathcal{O}(n)$. For each vertex in annulus i that we process, we perform an outward search and therefore step through the remaining $\frac{\alpha R}{\ln(2)} - i$ annuli ($\frac{\alpha R}{\ln(2)}$ total annuli). Thus, the number of annuli that we have to process for all vertices in annulus i is $n_0 \cdot 2^i \cdot (\frac{\alpha R}{\ln(2)} - i)$. We can then bound the total number of annuli that we process by a summation over all annuli.

$$\begin{aligned}
\sum_{i=0}^{\frac{\alpha R}{\ln(2)}} n_0 \cdot 2^i \cdot (\frac{\alpha R}{\ln(2)} - i) &= n_0 \cdot \Big(\sum_{i=0}^{\frac{\alpha R}{\ln(2)}} 2^i \cdot \frac{\alpha R}{\ln(2)} - \sum_{i=0}^{\frac{\alpha R}{\ln(2)}} 2^i \cdot i \Big) \\
&= n_0 \cdot \Big(\frac{\alpha R(2e^{\alpha R}-1)}{\ln(2)} - (e^{\alpha R}(\frac{2\alpha R}{\ln(2)} - 2) + 2) \Big) \\
&= n_0 \cdot \big(2e^{\alpha R} - \frac{\alpha R}{\ln(2)} - 2 \big) \\
&= 2n - \frac{2n}{e^{\alpha R}} - \frac{\alpha n R}{\ln(2) e^{\alpha R}} = \mathcal{O}(n)
\end{aligned}$$

The last equation holds true since R is proportional to $\ln n$. ∎

5.4.2 Parallel Generator

To parallelize the sequential approach, we begin by evenly subdividing the hyperbolic plane into P angular intervals. Processor $0 \leq i < P$ is then assigned the angular interval $[i \cdot \frac{2\pi}{P}, (i+1) \cdot \frac{2\pi}{P})$. For this purpose, we introduce an additional subdivision step to our grid data structure. We now discuss how each processor generates this new grid data structure.

As with the sequential approach, each processor starts be computing the number of vertices in each annulus using a multinomial random variate. We then use a recursive algorithm to subdivide each annulus into P chunks that represent the angular intervals for the corresponding processors. At each level of the recursion, we maintain the set of remaining chunks $\{j, \ldots, k\}$, as well as the number of vertices n'. Recursion begins by splitting the chunks $\{j, \ldots, k\}$ into equally-sized subsets $\{j, \ldots, l\}$ and $\{l+1, \ldots, k\}$. We choose a splitting value of $l = \lfloor \frac{j+k}{2} \rfloor$. The two resulting sets then correspond to the angular intervals $[j \cdot \frac{2\pi}{P}, l \cdot \frac{2\pi}{P})$ and $[(l+1) \cdot \frac{2\pi}{P}, k \cdot \frac{2\pi}{P})$.

To determine the number of vertices within each set of chunks, we generate a binomial random deviate (e.g.$x \sim B(n', \frac{l-j}{k-j})$). Each processor then continues recursion for both subtrees until we are left with a single chunk. At this point, each processor is able to generate the vertices within its own local chunk as with the sequential approach. This includes generating a set of cells depending on the expected number of vertices within its current chunk and annulus. For the remaining non-local chunks we only store their respective number of vertices. We do so, because the vertices within these chunks are only generated on demand during the edge generation process. Figure 5.10 shows an example of the resulting partitioning of the hyperbolic plane into annuli, chunks and cells.

Note that the resulting recursion tree within a single annulus has a size of at most $2P-1$ and a height of $\lceil \log P \rceil$. As with the sequential algorithm, we use a prefix sum over the number of vertices in each chunk to determine global vertex ids.

Lemma 5.4.2. *Generating the grid data structure for a set of P processors takes time $\mathcal{O}(P \log n + \frac{n}{P})$ and assigns each processor $\mathcal{O}(\frac{n}{P})$ vertices with high probability.*

Proof. Chunks are chosen in such a way that they assign each processor i an equal angular interval of the hyperbolic plane $[i \cdot \frac{2\pi}{P}, (i+1) \cdot \frac{2\pi}{P})$. The number of vertices per chunk in an annulus is generated through a set of binomial random variates. This result in a uniform distribution of the vertices in the interval $[0, 2\pi]$ with respect to their angular coordinate. Thus, each processor is assigned $\mathcal{O}(\frac{n}{P})$ vertices with high probability (Lemma 2.1.1).

The time spent during the chunk creation per annulus is $\mathcal{O}(P)$ with high probability, since the size of the recursion tree is at most $2P-1$ and we only spent expected constant time per level for generating the binomial random deviate. We have to repeat this recursion for each of the $\mathcal{O}(\ln n)$ annuli. Thus, the total time spent for the recursion over all annuli is $\mathcal{O}(P \ln n) = \mathcal{O}(P \log n)$.

As stated above, the chunk for each processor contains $\mathcal{O}(\frac{n}{P})$ vertices with high probability. We can generate these vertices and distribute them to their respective grid cells in

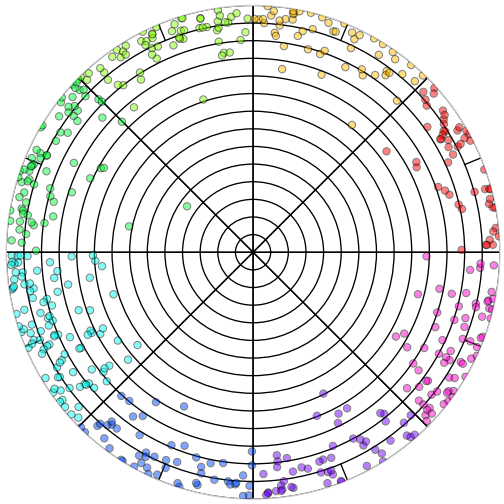


Figure 5.10: Partitioning of the hyperbolic plane into set of equidistant annuli and chunks. Each chunk is distributed to one processor and further subdivided into cells. The number of vertices is $n = 512$ with an average degree of $\bar{k} = 4$ and a power-law exponent of $\gamma = 2.6$. The expected number of vertices per cell for is set to 2^4. The local vertices for each processor are highlighted in different colors.

linear time. The running time bound then follows by combining the time spent on vertex creation as well as recursion. ∎

After generating the grid data structure, each processor is responsible for gathering all incident edges for its local set of vertices. We now present two different approaches for adapting the search queries used in the sequential algorithm for parallelization.

Communication Agnostic Approach. Our first approach generates all incident edges of a vertex without the need of communication. To do so, we need to recompute all non-local vertices that lie within the hyperbolic circle (of radius R) of any of our local vertices. To find these vertices, we perform two searches for each vertex, one outwards (as with the sequential algorithm) and one inwards.

Each of the searches behaves similarly to the sequential search, with the addition that any non-local chunks that we step through during the search are recomputed. To be more specific, if we examine a non-local cell during any of the angular searches that we perform within in an annulus, we redundantly generate the vertices for its whole chunk. These vertices are then assigned their respective cells and stored for future searches. An example of this process is presented in Figure 5.11a.

One issue with this approach is that the innermost annuli, which contain only a small number of vertices with high probability, are divided into P chunks. Since the innermost

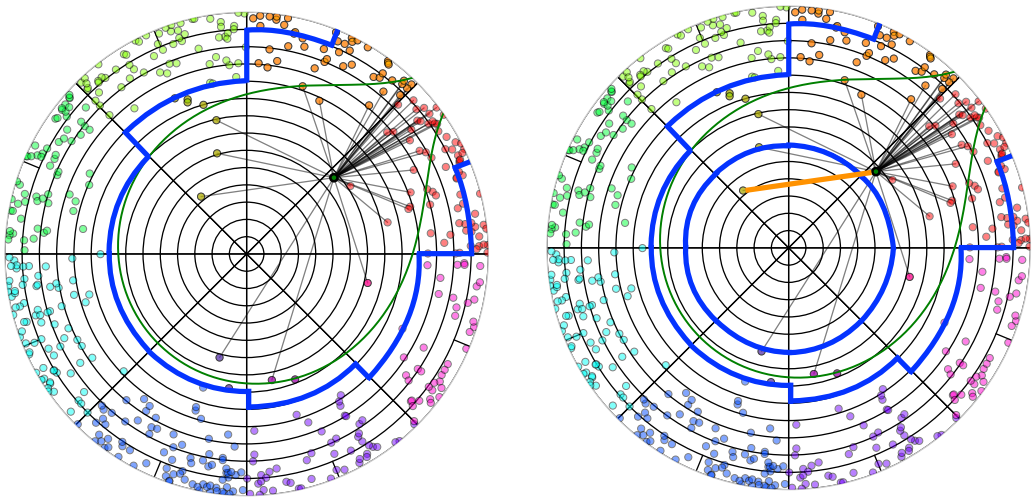


Figure 5.11: Example of search queries for the parallel RHG generator. The query vertex and its hyperbolic circle is highlighted in green. (a) The communication agnostic approach performs queries in both directions, inward and outward. (b) The communication efficient variant only performs a outward search. In this case, the highlighted edge has to be received from a different processor during the all-to-all step. In both variants, the detected cells are highlighted in blue.

annuli are almost always contained within the search radius for any given vertex, we have to iterate over P chunks for each of these levels. This severely impacts the running time for each individual search for a large number of processors. To alleviate this, we use a threshold on the number of annuli that are computed sequentially. In particular, if the number of cells in an annuli is below a given threshold, we store the whole set of vertices for this annuli within each individual chunk. Therefore, when performing an inward search, we only have to process one chunk instead of P, and perform comparisons with all its vertices. The pseudocode for the communication agnostic RHG generator is given in Algorithm 9.

Although we provide no proof for the running time of our parallel generator, our experimental analysis in Section 6.4.4 suggest an expected running time of $\mathcal{O}(\frac{n+m}{P} + P \log n)$.

Communication Based Approach. Our second approach only performs the outward search for each vertex as with the sequential algorithm. Again, we recompute any chunks that are detected during a search and store the resulting vertices for further iterations. The main problem with only performing an outward search for a vertex $v = (\phi_v, r_v)$ in annulus $i > 0$ is that we do not generate incident edges for its non-local neighbors that are within an annulus $j < i$. To fix this issue, we store the set of edges that run from local vertices to non-local vertices within an annulus with a smaller radial boundary. In addition to the edge itself, we also store the target processor for the non-local endpoint. We then distribute the

edges to their respective processors by using an all-to-all communication step. This all-to-all communication step can severly limit the scaling capabilities, especially for a very large number of processors. An example of the set of edges that need to be distributed to other processors is given in Figure 5.11b.

6 Experimental Evaluation

We now present the experimental evaluation of our sampling algorithm as well as graph generators. For each algorithm, we perform a sequential running time comparison to existing state-of-the-art implementations. We then continue to examine their strong and weak scaling behavior. Additionally, we show that pseudorandomization does not impede the quality of our generated graphs when examining common graph properties, e.g. degree distributions or clustering coefficients. The descriptions and experimental results for the sampling algorithms are extracts from Sanders et al. [64] which were developed as part of this thesis.

6.1 Implementation

We implemented our sampling algorithms as well as graph generators in C++[1]. We use Spooky Hash[2] as a hash function for pseudorandomization. The generated hash values are then used for initializing a Mersenne Twister [48] pseudorandom number generator for uniform deviates. Non-uniform random deviates are generated using the `stocc` library[3], which also uses a Mersenne Twister internally. If the size of our inputs (e.g. the number of vertices) exceeds 2^{64} bit, we use the multiple-precision floating points library `GMP`[4] and a reimplementation of the code from the `stocc` library. We now cover some implementation specifics for the individual algorithms that we use in our evaluation. If not mentioned otherwise, we compiled all algorithms and libraries using g++ version 6.2 using optimization level `fast` and `-march=native`. For the parallel versions of our algorithms, we use the MPI implementation MPICH 1.5[5] compiled with g++ version 4.9.3. The Python interface for our competitors is run with Python version 3.5.2.

Sampling Algorithms. We implemented Algorithm D in C++ from the descriptions in Vitter [72].

Algorithm H uses hashing with linear probing [42]. We can use a very simple and efficient hash function, such as extracting the most significant $\log n + O(1)$ bits from the key, since the elements inserted in the table are uniform random deviates. To further speed

[1] `https://github.com/sebalamm/GraphGen`
[2] `http://www.burtleburtle.net/bob/hash/spooky.html`
[3] `http://www.agner.org/random/`
[4] `http://www.mpfr.org`
[5] `http://www.mpich.org`

up table accesses, we choose a power of two as the table size. Depending on the desired output, sorted or unsorted, we use a different algorithm to obtain table entries and empty them. In the unsorted case, we use a stack to store the position of inserted elements. This way, we are able to empty the table without considering empty table entries. If we want to output table entries in sorted order, we are not able to use a stack, and instead scan the table at the end. Additionally, we allocate an additional set of n table entries to the right in order to avoid wrapping around the table during probing. Otherwise, the wrapping procedure is able to destroy the global sorted order between clusters of elements.

Our sequential divide-and-conquer algorithm uses Algorithm H as its base case sampler during recursion. We choose Algorithm H over Algorithm D because the former is faster for small subproblems, where the hash table fits completely into the cache [64]. Since we are able to freely choose the base case size for the divide-and-conquer algorithm (e.g. for $n_0 = 2^9$), this is always the case.

The parallel sampling algorithm uses the divide-and-conquer algorithm with parameters $n_0 = 2^8$ and $m = 2^{11}$ as its local subroutine.

Erdős-Rényi Generators. For the evaluation of our $G(n, m)$ generator, we compare it with implementations found in the Boost[6] and NetworkX[7] [66] libraries. NetworkX is a Python library that we use for validating the properties of our generated graphs. Their $G(n, m)$ graph generator is inspired by the selection sampling algorithm (Algorithm S) by Knuth [43]. The Boost implementation serves as a baseline for the sequential running time of our directed and undirected graph generators. They use a sampling procedure similar to Algorithm D, that only samples the edges that are actually present in the resulting graph.

In the case of the $G(n, p)$ model, we use the C++ implementation (and Python interface) of the NetworKit library[8] [70]. We use them both for running time comparisons and property validation. Their algorithm is an implementation of the random network generator proposed by Batagelj and Brandes [13]. During preliminary experiments, the Boost implementation had a very high running time for the $G(n, p)$ model even for moderately sized instances and thus is omitted from the comparison.

Our own generators for both the $G(n, m)$ and $G(n, p)$ model use our divide-and-conquer sampling algorithms with parameters $n_0 = 2^9$ and $m = 2^{12}$ as their local sampling routines.

Random Geometric Generators. Implementations of RGG generators are provided by the NetworkX library, as well as Holtgrewe et al. [35]. NetworkX uses a Python implementation of the naive $\Theta(n^2)$ algorithm that compares all pairs of points and adds edges iff their distance is less than the radius r. Their algorithm works for any arbitrary dimension d. The algorithm by Holtgrewe et al. [35] described in Section 3.2.2 is implemented

[6] `http://www.boost.org/doc/libs/1_62_0/libs/graph/doc/erdos_renyi_generator.html`

[7] `https://networkx.readthedocs.io/en/stable/reference/generators.html`

[8] `https://networkit.iti.kit.edu/api/generators.html`

using MPI and serves as our competitor for the running time comparison. Due to the slow running time of the RGG generator of NetworkX, we additionally use this algorithm for our property validation. It is also noteworthy that their implementation only features a two dimensional generator. Both Boost and NetworKit do not provide any RGG generators.

Random Hyperbolic Generators. For the running time comparison and validation of our hyperbolic generator, we again use the NetworKit library. Their hyperbolic generator features two different algorithms depending on the temperature of the statistical model. Since we are only interested in the case where the temperature $T = 0$, we compare our generator with their implementation of von Looz et al. [74].

For our own generator, we performed preliminary experiments to determine a good value for the expected number of points within each cell. Our experiments indicate that our algorithm is faster if we choose a relatively small number of points per cell, e.g. 2^4.

6.2 Experimental Setup

We now present our experimental setup including the specifications of our test hardware, as well as the design of our sequential and parallel experiments.

6.2.1 Environment

We conduct our experiments on two different types of machines. The sequential comparisons and property analyses are performed on a single core of a dual-socket Intel Xeon E4-2670 v3 system with 128 GB of DDR4-2133 memory, running Ubuntu 14.04 with kernel version 3.13.0-91-generic. For the scaling experiments we use the IBM Blue Gene/Q machine JUQUEEN. The JUQUEEN consists of 28 racks and 28,672 nodes which are connected via a 5D torus with 40 GBps and 2.5μsec latency. Each computation node of the JUQUEEN has 16 cores of an IBM PowerPC A2 system with 1.6 GHz, as well as 16 GB of DDR3-SDRAM. Each node runs the CNK (Compute Node Kernel) operating system which is a lightweight proprietary kernel. We use the maximum number of cores per node for our scaling experiments.

6.2.2 Experiment Design

We begin by performing a comparison of state-of-the-art implementations with our algorithms in terms of sequential running time. To account for the randomness during the generation process we are mainly interested in the time per generated sample/edge. Additionally, we provide the corresponding number of vertices and/or edges to compute the total running time. If not mentioned otherwise, we average the resulting running times over ten iterations with different seeds for varying input sizes. We then compare the reported

running times with our theoretical results presented in the previous section. For this set of experiments we use an edge list as the output format for all our generators.

Afterwards, we examine the scaling behavior of our algorithms. To this end, we measure both their strong and weak scaling behavior. Strong scaling is a measurement of how the running time varies with the number of processors for a fixed problem size. To be more specific, if the amount of time needed one a single processor is t_1, and the amount of time needed on P processors it t_P, then the strong scaling efficiency is given as

$$s_{\text{strong}} = \frac{t_1}{P \cdot t_P}.$$

Weak scaling is a measurement of how the running time varies with the number of processors for a fixed problem size *per processor*. If the amount of time needed one a single processor is t_1, and the amount of time needed on P processors is t_P, then the weak scaling efficiency is given as

$$s_{\text{weak}} = \frac{t_1}{t_P}.$$

Again, we average our results over ten iterations with different seeds. Additionally, we compare the experimental results with our theoretical running times. Since the memory restrictions of the JUQUEEN do not allow us to store a complete edge list for the local subgraph of each processor, we only store the degree distribution for a fixed amount of k nodes on each processor.

Finally, we validate the correctness of our generated graphs and measure the impact of pseudorandomization on the generation process, by performing an analysis of various graph properties. For this purpose, we compare the degree distribution, clustering coefficient, as well as the size of connected components of the graphs generated by our algorithm with the implementations presented in Section 6.1. Each property is averaged over a set of 100 iterations with different seeds for graphs of varying sizes.

6.3 Running Time Comparison

We now compare the running time of our divide-and-conquer sampling algorithm, as well as graph generators, to existing state-of-the-art implementations. If not mentioned otherwise, we are only concerned with the sequential running times of each algorithm. The experimental results for the sequential sampling algorithm are extracts from Sanders et al. [64] which were developed as part of this thesis.

6.3.1 Sampling Without Replacement

We begin by comparing the performance of our divide-and-conquer sampling algorithm (Algorithm R) to Algorithm D and H. Since Algorithm D returns samples in sorted order, we additionally test a variant of Algorithm R that also returns samples in sorted order (Algorithm SR). We use a population size of $N = 2^{50}$ and an increasing amount of samples n. The number of iterations for each n is set to $2^{30}/n$. This way the workload stays the same for each value of n. Figure 6.1 shows the performance for each algorithm.

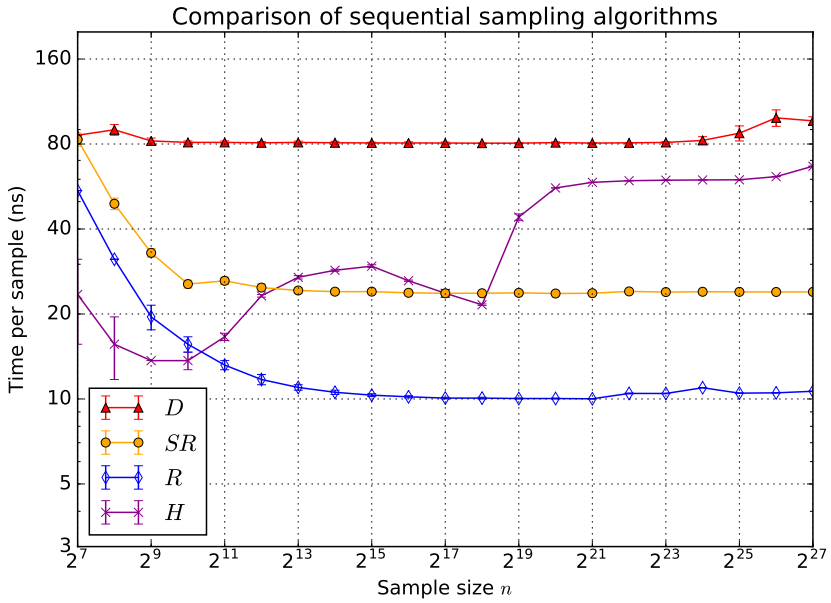


Figure 6.1: Running time per sample for the sequential algorithms H, D, R. The bars show the standard deviation. The number of repetitions for each algorithm is $2^{30}/n$. For Algorithm R, we use $n_0 = 2^{10}$. *SR* is Algorithm AR with sorted output.

Algorithm H is very fast for small n, but starts to degrade as the size for the hash table exceeds the cache size. Our divide-and-conquer algorithm (Algorithm R) has a similar performance for small n but the time needed for each sample remains constant for growing n. If we compare it to Algorithm H for large n, we can see that it is up to 5 times

faster. Algorithm D is the slowest of the four algorithms tested, but its time per sample is still independent of n for growing sample sizes. Compared to Algorithm R it is up to 7 times slower for large n. The variant of Algorithm R that returns samples in sorted order (Algorithm SR) is still 3.4 times faster than Algorithm D and the time per sample remains constant for growing n.

6.3.2 Erdős-Rényi Generators

We now compare the performance of our Erdős-Rényi generators to existing implementations found in the Boost and NetworKit libraries. Note that when executed on a single processor, our generators are equivalent to a variant of sampling without replacement. This is because no recursion has to be performed in order to distribute the workload in the sequential case. As pointed out above, both the $G(n, m)$ and $G(n, p)$ generators use the sequential divide-and-conquer algorithm with $n_0 = 2^9$ for their base case.

We evaluate the performance of each generator for different numbers of vertices n and a growing (expected) number of edges m. To be more specific, we set the number of vertices from 2^{18} to 2^{24} and the (expected) number of edges from 2^{16} to 2^{28}. We compute the edge probability from the expected number of edges as $p = \frac{m}{n(n-1)}$ and $p = \frac{m}{n(n-1)/2}$ for the directed and undirected case respectively. Figure 6.2 shows the results of our experiments for the smallest and largest set of vertices.

First, we note that all of the Erdős-Rényi generators have a linear increase in running time (constant time per edge) with increasing m. However, both the Boost and NetworKit implementation also have an increasing time per edge for growing numbers of vertices n. Especially for small m and large n, the running time of our competitors is dominated by the number of vertices. In contrast, the running time of our generator is independent of n. Since we use a simple edge list as our output format this is not surprising. All in all, the results are consistent with the optimal theoretical running time of $\mathcal{O}(n + m)$ with high probability for all competitors.

We now take a detailed look at the performance of each generator. For the directed $G(n, m)$ model, our generator is roughly 10 times faster than Boost for the largest value of $m = 2^{28}$. If we compare the directed $G(n, p)$ generators, we can see that KaGen is up to 4 times faster than the NetworKit generator for larger inputs. In the undirected case, the running time of Boost increases slightly compared to its directed generator. In turn, our $G(n, m)$ generator is roughly 21 times faster than Boost and has an equally lower time per edge. For the undirected $G(n, p)$ generator NetworKit is roughly 3.7 times faster compared to its directed counterpart for large n. However, our algorithm is still more than 9 times faster for large values of m.

6.3.3 Random Geometric Generators

In this section we evaluate the performance of our two random geometric graph generators. In particular, we compare them to the implementation of Holtgrewe et al. [35, 36]. Since

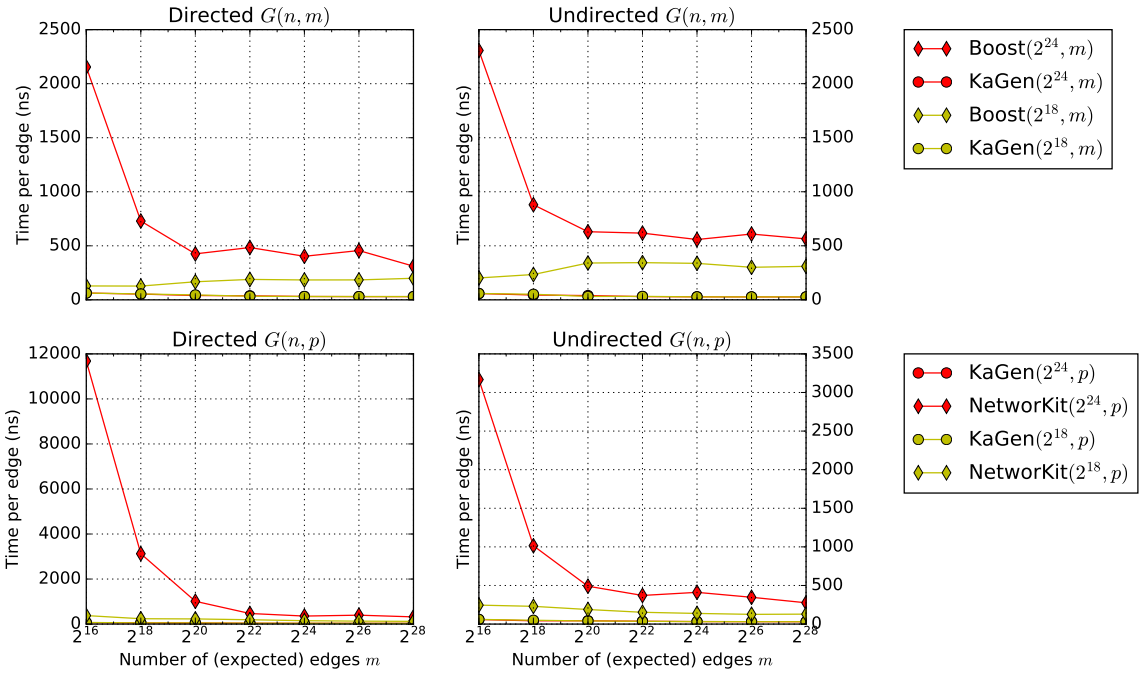


Figure 6.2: Time per edge for the Erdős-Rényi generators for 2^{18} and 2^{24} vertices and 2^{16} to 2^{28} (expected) edges. For the $G(n, p)$ generators the edge probability is set to $p = \frac{m}{n(n-1)}$ (directed) and $p = \frac{m}{n(n-1)/2}$ (undirected) respectively.

both competitors are nearly identical for the sequential case, we are mainly interested in their parallel running time behavior. For this purpose, we compare them in terms of their parallel running time for instances of growing sizes. For the sequential case, we only examine our own generators.

The results for the sequential experiments are presented in Figure 6.3. We test our generators for growing numbers of vertices from 2^{16} to 2^{26}. The radius is set to $r = 0.55\sqrt{\frac{\ln n}{n}}$ for the two dimensional case, and $r = 0.55\sqrt[3]{\frac{\ln n}{n}}$ for the three dimensional case. We first notice that both generators have a constant time per edge, and thus linear running time increase with a growing number of vertices. To be more specific, the time per edge for both generators quickly reduces, but remains roughly constant for larger n. The theory for our generators suggests a running time of $\mathcal{O}(n \log n)$ with high probability when choosing a radius of $r \in \mathcal{O}(\sqrt{\frac{\ln n}{n}})$. Thus, experiments with even more nodes and/or different radii might be required to validate this bound.

Next, we compare our algorithms to the implementation of Holtgrewe et al. in terms of their parallel running time. It should be noted that Holtgrewe et al. only support two dimensional random geometric graphs. Thus, the three dimensional generator is excluded from our evaluation. Figure 6.4 shows the running time of both competitors for a growing

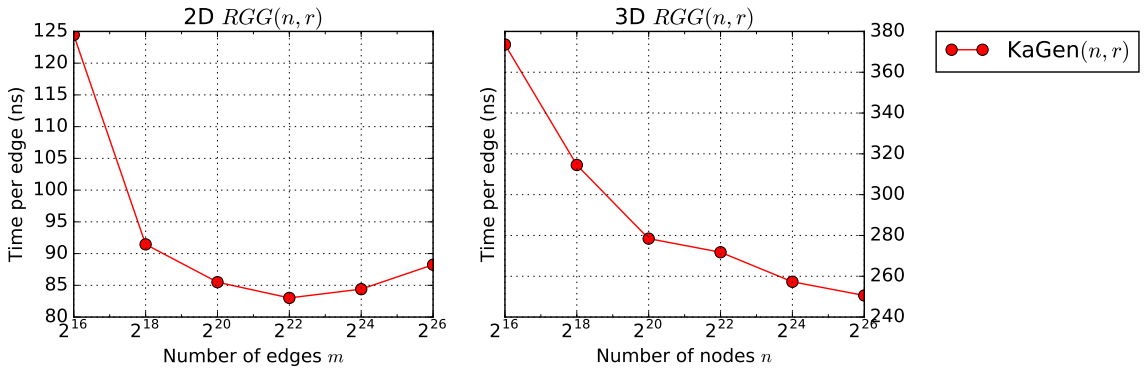


Figure 6.3: Time per edge for the random geometric graph generators for 2^{16} to 2^{26} vertices. The radius is set to $r = 0.55\sqrt{\frac{\ln n}{n}}$ (2D) and $r = 0.55\sqrt[3]{\frac{\ln n}{n}}$ (3D) respectively.

number of $P = p^2$ processors. The input size per processor is set to $n_0 = n/P$ and varies from 2^{16} to 2^{20}. To maintain an equal workload per processor the radius is set to $r = 0.55\sqrt{\frac{\ln n_0}{n_0}}/\sqrt{P}$.

We are able to see that the running time for both generators behaves very similarly. Nonetheless, our algorithms is roughly a factor of 2 faster than our competitor. Additionally, we notice a slightly increasing gap between the two algorithms for $n_0 = 2^{20}$ vertices. This can be attributed to the increased communication effort to exchange vertices during the algorithm of Holtgrewe et al. [35]. In contrast, our algorithm maintains a a roughly constant running time even for a larger number of processors.

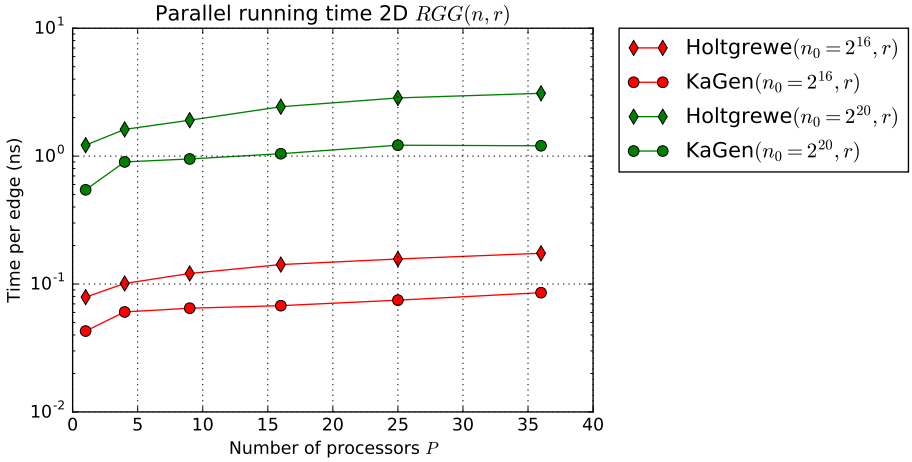


Figure 6.4: Running time for the two dimensional random geometric graph generators for and a growing number of processors $P = p^2$ and a constant input size $n_0 = n/P$ per processor. The radius is set to $r = 0.55\sqrt{\frac{\ln n_0}{n_0}}/\sqrt{P}$ to maintain a constant workload per processor.

6.3.4 Random Hyperbolic Generators

We now examine the performance of our random hyperbolic generator and compare it to the implementation given in the NetworKit library. The NetworKit implementation, which is capable of multi-threaded execution, only uses a single thread to ensure a fair comparison. We test both generators for numbers of vertices from 2^{18} to 2^{24} and average degrees from 4 to 256. These are average degrees found in various real-world networks [73]. Figure 6.5 shows the results of our experiments in terms of time per edge for two common values of the power-law exponent γ [23, 55].

We can see that the NetworKit generator is superior to our implementation for small average degrees, independent of the power-law exponent. For the smallest average degree of 4, NetworKit is up to 3.5 times faster than our generator. However, for larger average degrees the speedup of NetworKit diminishes. Starting at an average degree of 32 our algorithm becomes faster than NetworKit. For the largest input sizes that we test, our algorithm is roughly 1.4 times faster than our competitor.

Concerning the asymptotic behavior of both algorithms, NetworKit seems to exhibit a steeper increase in running time for growing average degrees than our own generator. This behavior is expected, as NetworKit has an experimentally observed running time of $\mathcal{O}(n \log n + m)$ [74].

Our own generator has a roughly constant time per edge (linear running time) for large n and growing averages degrees. This is in line with the expected theoretical running time of $\mathcal{O}(n+m)$ given in Lemma 5.4.1. The slight increase in running time for a growing number of edges can most likely be attributed to lower-order terms.

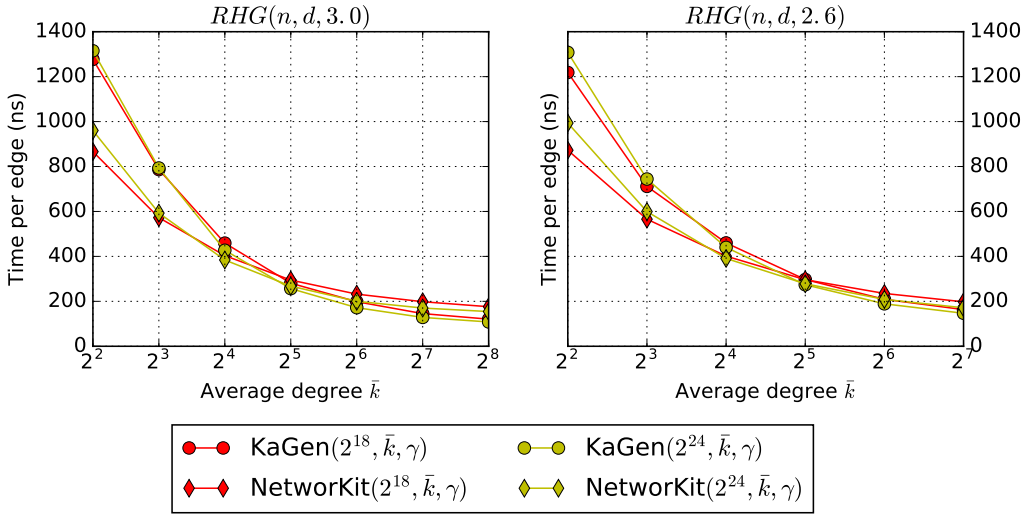


Figure 6.5: Time per edge for the random hyperbolic graph generators for 2^{18} and 2^{24} vertices and average degrees between 4 and 256. The power-law exponent γ is fixed to 3.0 (uniform distribution) and 2.6 respectively. The NetworKit generator is executed on a single thread.

6.4 Scaling Behavior

In this section we discuss the scaling behavior of our communication agnostic sampling routine and graph generators. We analyze both their weak and strong scaling behavior. We then compare the experimental results with our theoretical findings. Each experiment uses the full 16 cores available for each computation node of the JUQUEEN. Note that the performance of a single core of a computation node is an order of magnitude lower than the Intel CPUs used for the sequential experiments [64]. We can partially attribute this to the lower clock frequency and lower number of transistors used. Our algorithms also use a SIMD-oriented Mersenne Twister, which contains optimizations that make use of the SSE 2 units of Intel CPUs. Similar optimizations for the QPX instructions of Blue Gene/Q are not available. Additionally, there is a lack of autovectorization for Blue Gene/Q in g++. The experimental results for the parallel sampling algorithm are extracts from Sanders et al. [64] which were developed as part of this thesis.

6.4.1 Sampling Without Replacement

We begin by examining the performance of our parallel sampling algorithm (Algorithm P). Figure 6.6 shows the weak scaling behavior of our algorithm, i.e. we keep the local input size n/p per processor constant while increasing the number of total processors. We then use different values for the ratio n/P and set the number of iterations to $2^{30} \cdot P/n$ to keep the same amount workload for increasing P. We can see that our parallel algorithm scales almost perfectly for sufficiently large values of n/P. Only for the smallest local input size of $n/P = 4096$, we see a linear increase in running time with an exponential increase in P. Thus, for smaller inputs, the recursion step of our algorithm has a significant impact on the running time. All of these results are in line with the asymptotic running time of $\mathcal{O}(n/P + \log P)$ presented in Section 4.2.

6.4.2 Erdős-Rényi Generators

We now discuss the strong and weak scaling behavior of our Erdős-Rényi generators. For the weak scaling experiments each processor is assigned an equal number of n/P nodes and m/P edges to sample. For the strong scaling experiments we have a constant number of n nodes and m edges that are distributed over all processors. We set $n/P = (m/P)/2^4$ for weak scaling experiments and $n = m/2^4$ for strong scaling experiments. The number of edges to sample per processor (over all processors) ranges from 2^{20} up to 2^{28}. The edge probability for the $G(n, p)$ generators is computed by dividing the expected number of edges by the number of possible edges, as with the sequential experiments. We additionally tested the capabilities of our algorithms by generating directed Erdős-Rényi graphs with 2^{47} edges and 2^{43} vertices on 32.768 cores. We were able to generate these instances in less than 22 minutes.

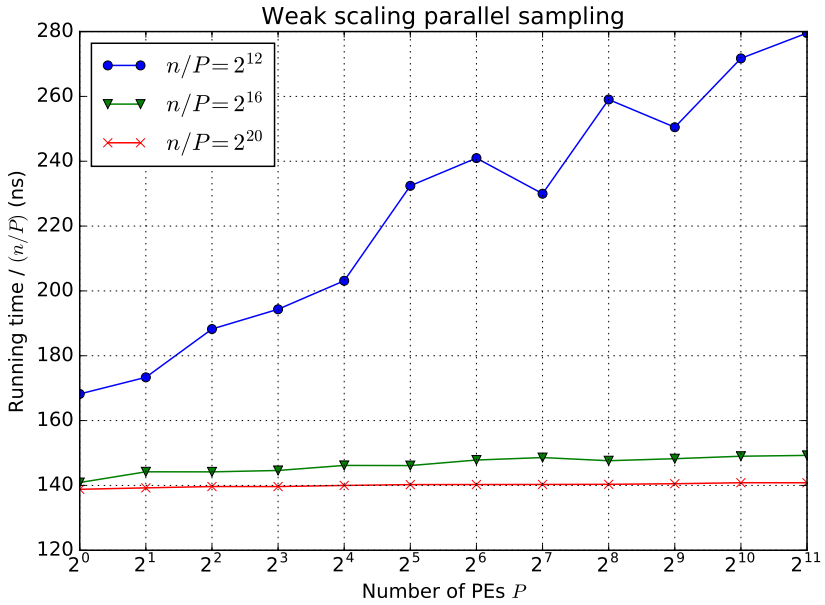


Figure 6.6: Running time for generating n samples on P processors for different values of n/P using Algorithm P with Algorithm R as local sampler, using $n_0 = 2^8$.

$G(n, m)$ generators. The results for the $G(n, m)$ generators are presented in Figure 6.7. We first discuss the scaling behavior of our directed generator. Since the directed Erdős-Rényi generators are an adaptation of the parallel sampling algorithm, they behave very similarly to it. We can see that our algorithm has an almost perfect strong and weak scaling behavior. Only for the smallest input sizes and more than 2^{10} processors, the logarithmic term of our running time becomes noticeable in the strong scaling results. Nonetheless, this is consistent with the asymptotic running time $\mathcal{O}(\frac{n+m}{P} + \log P)$ that we give in Lemma 5.2.1.

Next, we examine the scaling behavior of our undirected $G(n, m)$ generator. If we look at its weak scaling behavior, we can see that for a growing number of processors the running time starts to increase and then remains constant up until larger numbers of processor. This is due to the fact that as the number of processors/chunks increases the number of redundantly generated edges also increases up to twice the number of sequentially sampled edges. Afterwards, the running time stays constant for larger values of m/P. For smaller values of m/P we see an exponential increase in running time with an exponential increase in P. We can attribute this to the linear time in the number of processors needed to locate the correct chunks for each processor. The same effects occurs for the strong scaling experiments. Thus, we can experimentally validate the asymptotic running time of $\mathcal{O}(\frac{n+m}{P} + P)$ given in Lemma 5.2.2.

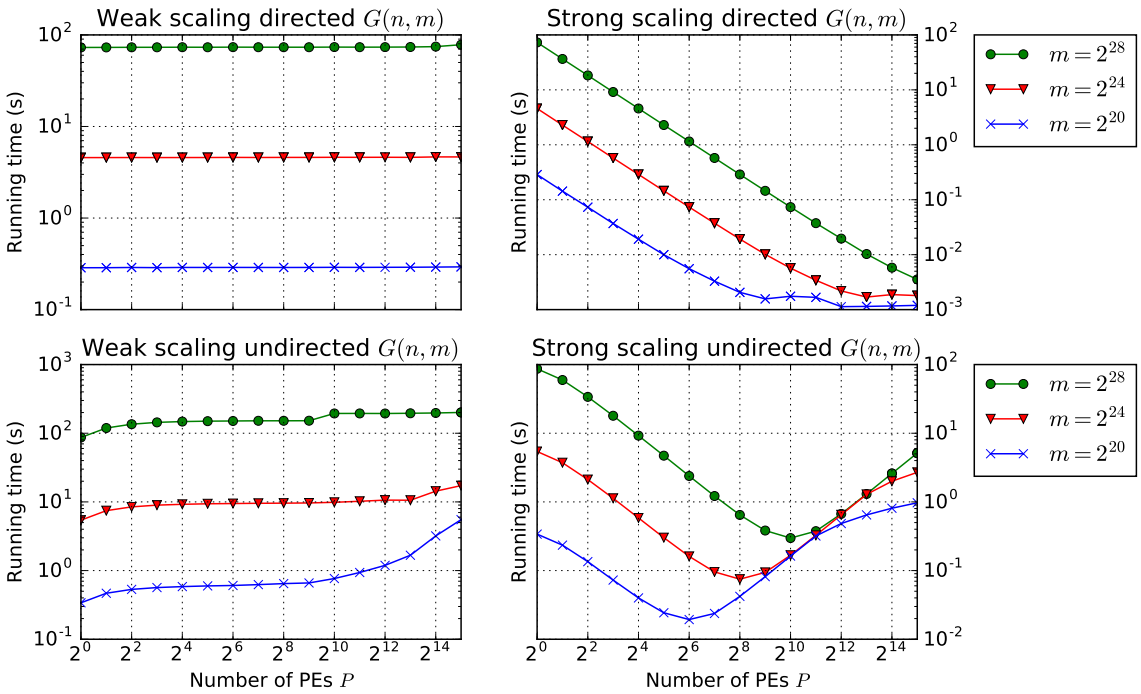


Figure 6.7: Running time for generating m edges and $n = m/2^4$ vertices on P processors using the $G(n, m)$ generators. For the weak scaling experiments, m is the number of edges per processor.

$G(n, p)$ generators. The scaling behavior for the $G(n, p)$ generators is given in Figure 6.8. As we mentioned in Section 5.2, these generators are simplifications of the $G(n, m)$ generators. To be more specific, the directed $G(n, p)$ generator works in the same way as the $G(n, m)$ generator but without the additional recursion overhead, since the chunk distribution can be computed directly. The undirected $G(n, p)$ generator has the same asymptotic running time as the undirected $G(n, m)$ generator, but locating the correct chunks for each processor is faster. This is because we do not have to compute several hypergeometric random deviates. Thus, both generators have a very similar scaling behavior to their $G(n, m)$ counterparts.

We can see that even for the smallest input sizes that we tested, our directed generator has an almost perfect weak and strong scaling behavior. In Section 5.2.3 we stated that the running time of the directed $G(n, p)$ generator is $\mathcal{O}(\frac{m+n}{P})$ with high probability for an expected number of m edges. Therefore, we are able to validate this bound with our experiments.

Compared to the undirected $G(n, m)$ generator, the running time of the undirected $G(n, p)$ generator is roughly 10% lower. The strong scaling behavior is once again dominated by the additive $\mathcal{O}(P)$ term for growing numbers of processor, especially for smaller input

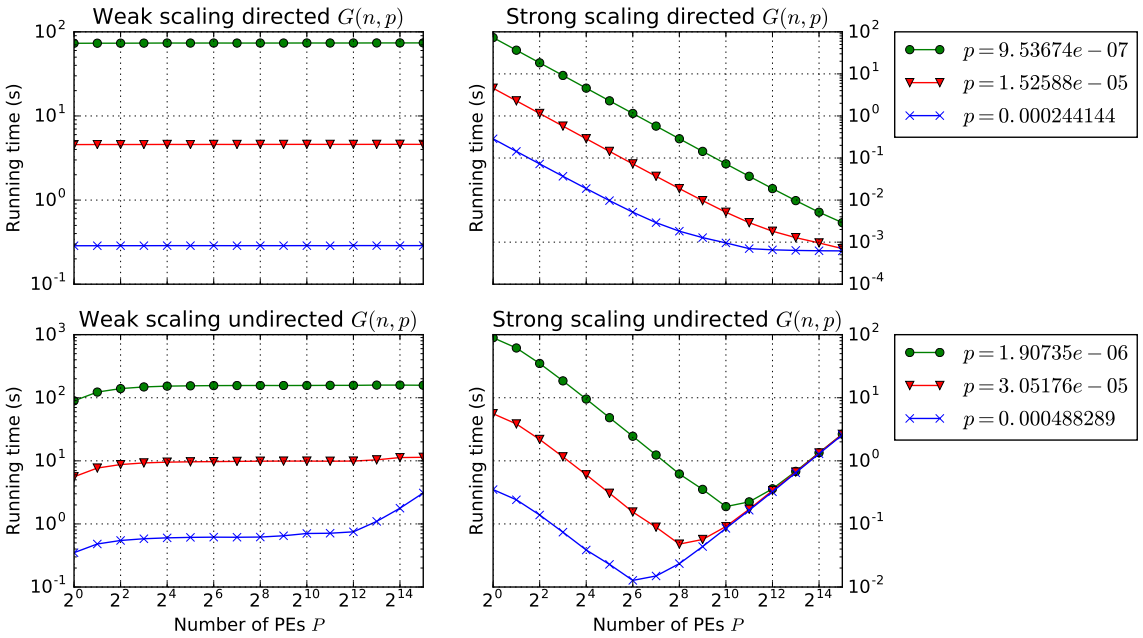


Figure 6.8: Running time for generating an expected number of m edges and $n = m/2^4$ vertices on P processors using the $G(n,p)$ generators. For the weak scaling experiments, m is the number of edges per processor.

sizes. Nonetheless, all results are consistent with the theoretical running times given in Section 5.2.4.

6.4.3 Random Geometric Generators

We now discuss the results of our scaling experiments for the two and three dimensional random geometric generators presented in Figure 6.9. For the weak scaling experiments each processor is assigned an equal number of n/P nodes. For the strong scaling experiments we have a constant number of n nodes that are distributed over all processors. The number of processors P is set to a square (cubic) number p^2 (p^3) as explained in Section 5.3.1 (Section 5.3.2).

In the two dimensional case, the number of nodes per processor (over all processors) is 2^{18} to 2^{24}. The radius r is set to $0.55\sqrt{\frac{\ln n/P}{n/P}}/\sqrt{P}$ (weak scaling) and $0.55\sqrt{\frac{\ln n}{n}}$ (strong scaling) respectively.

For the three dimensional generator, the number of nodes per processor (over all processors) is 2^{16} to 2^{20}. Analogously, we set r to $0.55\sqrt[3]{\frac{\ln n/P}{n/P}}/\sqrt[3]{P}$ (weak scaling) and $0.55\sqrt[3]{\frac{\ln n}{n}}$ (strong scaling).

We first consider the weak scaling behavior for the two dimensional generator. We see that the running time for all input sizes quickly increases over one, four and nine proces-

sors. It then remains constant until a large number of processors is reached. This is due to the fact that the number of neighbors that we have to generate redundantly increases from zero for one processor up to eight neighbors for more than four processors. The increase in running time can be bound by computing the additional amount of vertices created through redundant computations (end of Section 5.3.1) and then multiplying it by the average degree $n\pi r^2$. This bound yields roughly twice the running time needed for the sequential computation, which is consistent with the experimental results. We are also able to see this effect for the strong scaling experiments.

We also notice that for the smallest input size and very large numbers of processors there is a exponential increase in running time with an exponentially increasing number of processors. This is attributed to the linear time $\mathcal{O}(P)$ needed to determine the correct number of vertices for each chunk and compute the corresponding prefix sum. The effect becomes more noticeable for the strong scaling experiments, since the number of nodes per processor is steadily decreasing.

Next, we examine the scaling experiments for the three dimensional generator. Again, we are able to observe both of the effects mentioned above. Therefore, the scaling behavior of both random geometric generators is in line with the theoretical running times of $\mathcal{O}(\frac{m+n}{P} + P)$ given in Lemma 5.3.2 and Lemma 5.3.4.

6.4.4 Random Hyperbolic Generators

Lastly, we present the results of our scaling experiments for the random hyperbolic generators. For the weak scaling experiments each processor is again assigned an equal number of n/P nodes. For the strong scaling experiments we have a constant number of n nodes that are distributed over all processors. The number of nodes per processor (over all processors) is 2^{16} to 2^{20}. The power-law exponent γ is fixed to 3.0.

Figure 6.10 shows the scaling behavior of the RHG generator with an average degree of $\bar{k} = 4$. Looking at the weak scaling behavior, we are able to see that there is a slight increase in running time, especially for a small number of processors. Afterwards, the running time remains roughly constant until a large number of processors is reached. We can attribute this behavior to the redundant computations that are introduced through parallelization, similar to the RGG generators. Since each chunk contains a constant expected number of points, this additional workload remains roughly constant for larger numbers of processors. In the strong scaling experiments, where the number of points in each chunk is steadily decreasing, this effect is less noticeable.

Once our generator reaches a larger number of processors, we see an exponential increase in running time with an exponential increase in the number of processors. This especially holds true for the smallest inputs. If we take a look at the strong scaling experiments, we also see that there is a difference of roughly $2 - 3$ seconds between the individual runs. We attribute this behavior to the time $\mathcal{O}(P \log n)$ necessary to build the grid data structure as presented in Lemma 5.4.2. Overall, the scaling behavior of our random hyperbolic generator indicates a running time of $\mathcal{O}(\frac{m+n}{P} + P \log n)$.

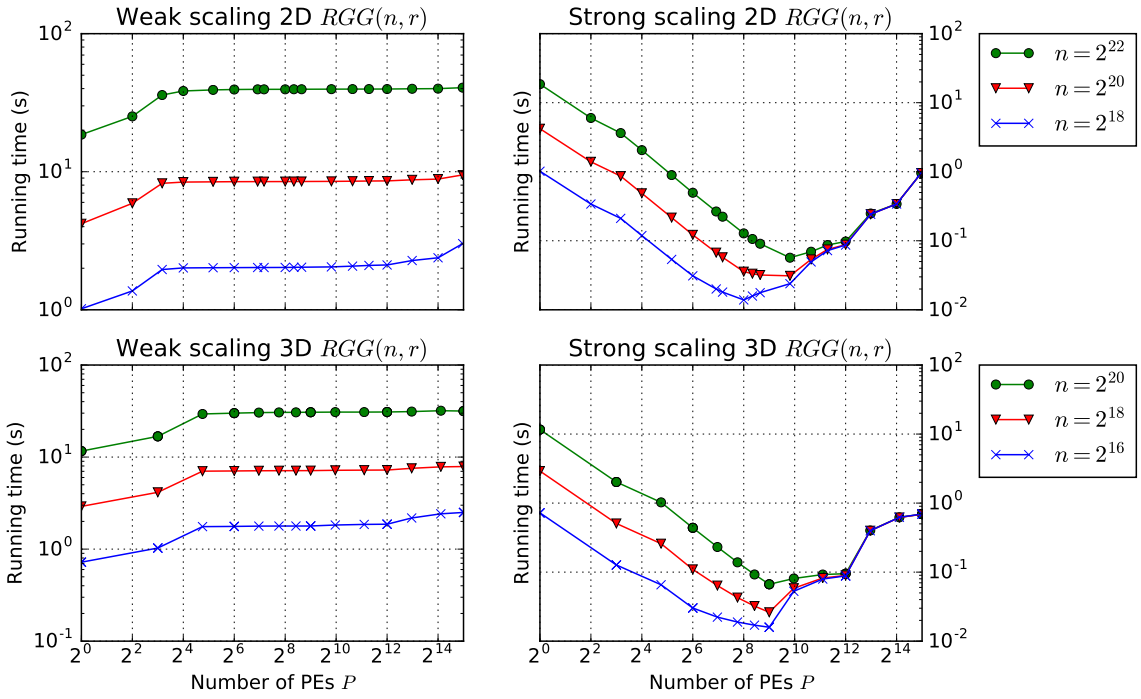


Figure 6.9: Running time for generating a graph with n vertices on P processors using the RGG generators. The radius r is set to $0.55\sqrt[\{2,3\}]{\frac{\ln n/P}{n/P}}/\sqrt{P}$ (weak scaling) and $0.55\sqrt[\{2,3\}]{\frac{\ln n}{n}}$ (strong scaling) respectively. For the weak scaling experiments, n is the number of vertices per processor.

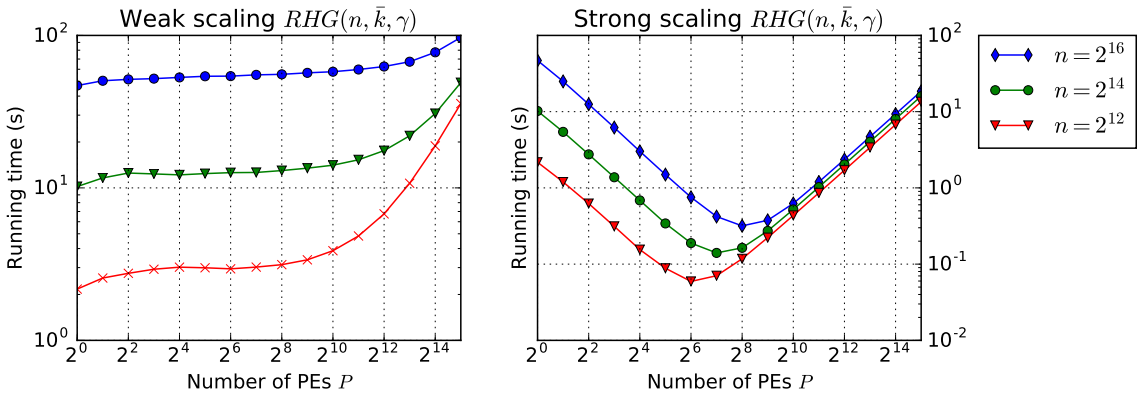


Figure 6.10: Running time for generating a graph with n vertices, average degree $\bar{k} = 4$ and $\gamma = 3.0$ on P processors using the RHG generator. For the weak scaling experiments, n is the number of vertices per processor.

6.5 Pseudorandomization

We now cover the impact of pseudorandomization on our generators and the resulting output graphs. For this purpose, we first discuss the quality of the pseudorandom number generators (PRNG) used for our graph generators. Afterwards, we compare the graphs produced by our generators with state-of-the-art implementations. In particular, we examine their degree distributions, clustering coefficients, as well as the size of their largest connected component.

Quality of PRNGs. As mentioned in Section 6.1, we use Spooky Hash as a source for our pseudorandom number generators. Spooky Hash is a 128-bit noncryptographic hash function proposed by Bob Jenkins[9] that is able to produce 128-bit, 64-bit and 32-bit hash values. Short keys are hashed in about 1 byte per cycle, whereas long keys are processed in 3 bytes per cycle. Additionally, there is a 30 cycle startup overhead. Even though Spooky Hash is noncryptographic, it achieves avalanche [26] for 1-bit and 2-bit inputs. When hashing roughly 42 million unique keys, the fraction of keys hashed without collision is 1.0 [1]. Additionally, Spooky Hash shows good behavior when analyzing the random uniformity of the hash values using a chi-squared test that places the hash values into a set of one million bins [2].

Our pseudorandom number generator is a double precision floating point SIMD-oriented Mersenne Twister (dSFMT) by Matsumoto et al. [62]. This generator is more than twice as fast as the original Mersenne Twister proposed by Matsumoto et al. [48]. Additionally, it provides a better equidistribution than the original implementation. It supports periods from $2^{521}-1$ to $2^{216091}-1$. For our experiments we use a period of $2^{19337}-1$. The dSFMT passes both the DIEHARD tests [47] and TestU01 tests [45]. Thus, it exhibits very good statistical properties.

Graph Properties. We now briefly discuss the results of our property evaluation. More detailed results are presented in Appendix B. All graph properties were evaluated using the NetworKit [70] library.

For the directed and undirected Erdős-Rényi generators, we tested graph sizes of 2^{12} to 2^{18} vertices and 2^{14} to 2^{20} (expected) edges. Again, the edge probability for the $G(n,p)$ generators is computed by dividing the expected number of edges by the total number of edges possible. Due to hardware limitations, we could only examine the size of the strongly connected components of the directed generators for instances of up to 2^{16} vertices.

The properties of the RGG generators were examined on instances of 2^{14} to 2^{18} vertices for radii varying from 0.001 to 0.01. Finally, we evaluated our RHG generator for instances of 2^{14} to 2^{18} vertices with average degrees between 4 and 128 as well as different values of $\gamma \in (2,7]$. If not mentioned otherwise, all results are averaged over 100 iterations with different seeds.

[9] `http://www.burtleburtle.net/bob/hash/spooky.html`

Figure 6.11 and Figure 6.12 show the degree distributions, clustering coefficients and the size of the largest connected components for different graph generators. We can see that our undirected $G(n, m)$ generators are able to nearly exactly match the properties of its competitor. This includes the clustering coefficients, connected component sizes, as well as degree distributions. Due to their similar behavior, the same holds true for the properties of the $G(n, p)$ generators shown in Appendix B.

For the RGG and RHG generators, there are some minor differences of the clustering coefficients, especially for smaller instances. However, since the standard deviation for these instances is fairly large, more iterations might be required to obtain conclusive results. Nonetheless, when examining the size of the largest components for both generators, they both behave very similarly. The same holds true for the degree distributions of both generators which are given in Appendix B.2 and Appendix B.3.

Overall, our results indicate that pseudorandomization does not significantly impede the quality of our generated instances. However, in order to properly validate these results, more thorough tests, i.e. an analysis of the variance of our results, are required. We leave this analysis as a possible topic for future work.

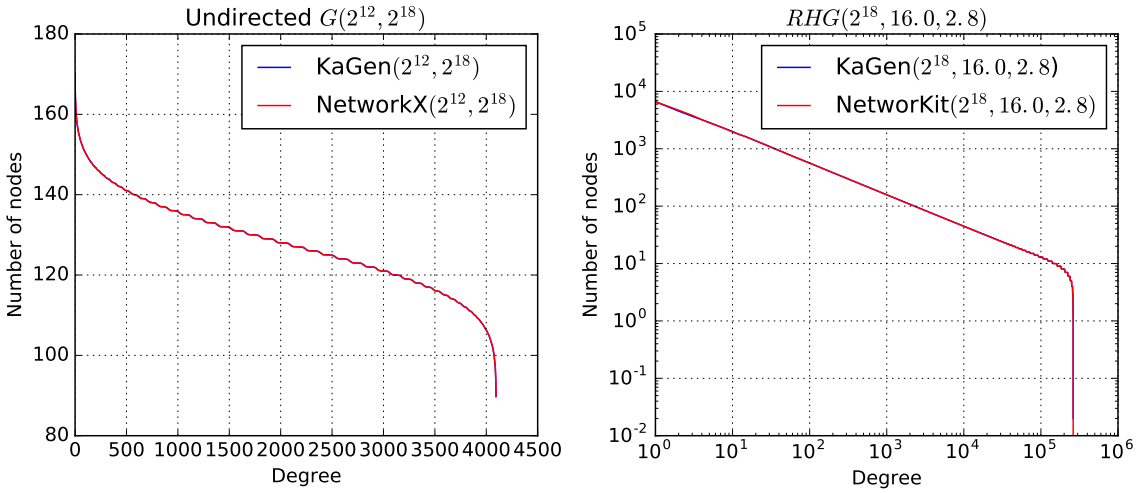


Figure 6.11: Degree distributions for the $G(n, m)$ and RHG graph generators. The distributions for the Erdős-Rényi model follow a binomial distribution. For the RHG model they follow a power-law distribution.

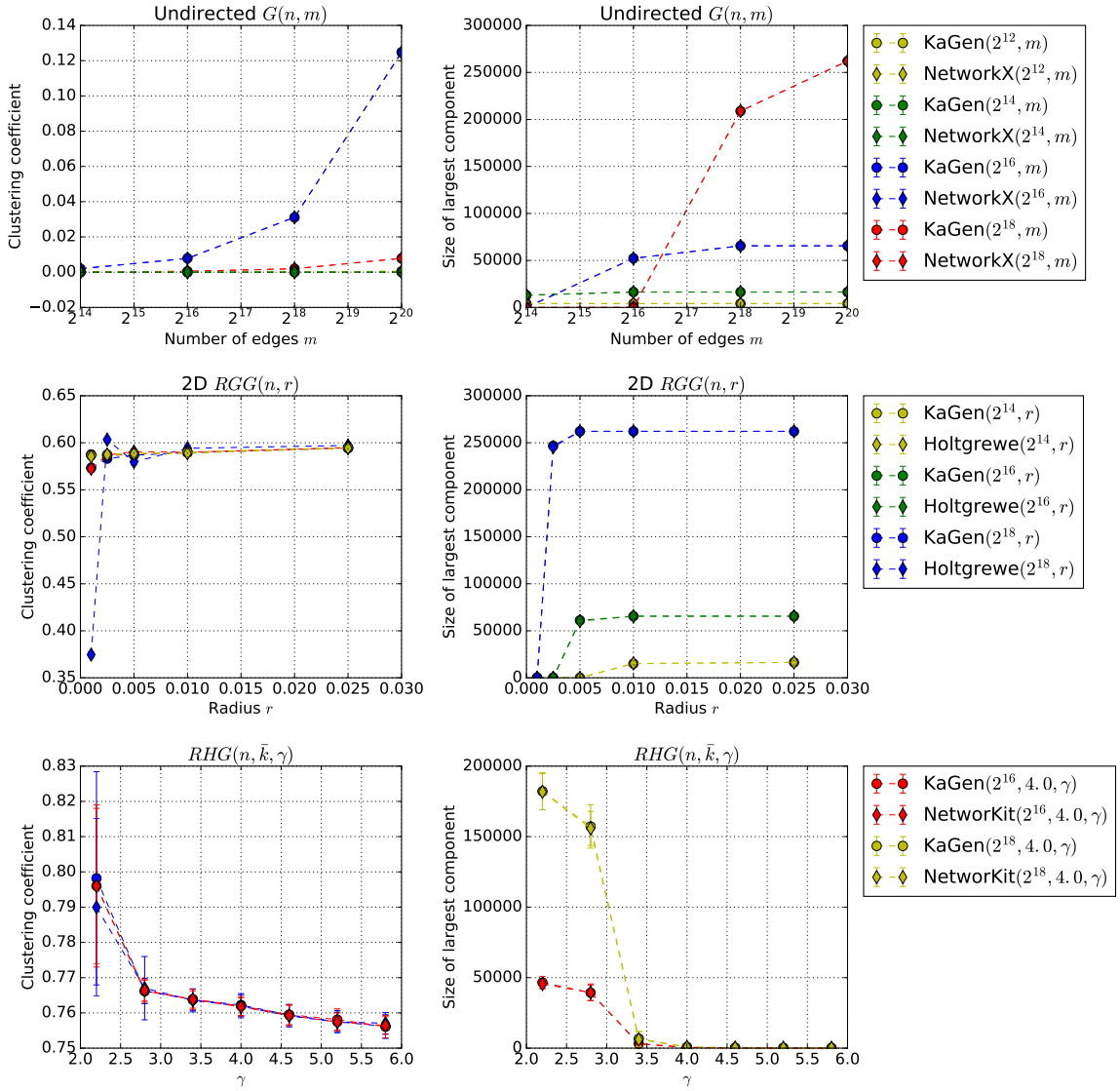


Figure 6.12: Clustering coefficients and sizes of the largest connected component including standard deviations for different graph generators and instances of varying sizes.

7 Discussion

We now give a short conclusion of the work presented in this thesis. Additionally, we highlight some interesting areas for future work.

7.1 Conclusion

In this thesis we developed scalable graph generators for a set of commonly used network models. Our work includes the classic Erdős-Rényi model, in both the $G(n, m)$ and $G(n, p)$ variants, as well as random geometric graphs and random hyperbolic graphs. As part of our thesis we also developed an efficient divide-and-conquer sampling algorithm.

All of our graph generators share a common goal of communication efficiency through redundant computations and pseudorandomization. Since all our generators require no communication at all, we call them *communication agnostic*.

Most of our algorithms make use of a combination of divide-and-conquer schemes and grid data structures to narrow down their local sampling space. We then redundantly compute parts of the network that are within the neighborhood of local vertices. These computations are made possible through the use of pseudorandomization via hash functions. The resulting algorithms are often embarrassingly parallel. Additionally, we provided theoretical running times for our algorithms in both the sequential and parallel case.

We then compared our graph generators to existing state-of-the-art implementations. This includes the graph generators found in the Boost, NetworkX and NetworKit libraries. Our experimental evaluation indicates that our generators are often able to outperform their competitors. To be more specific, our Erdős-Rényi generators are up to 20 times faster than state-of-the-art implementations found in Boost and NetworKit. Our random geometric generators are up to 4.5 faster than an algorithm of Holtgrewe et al. [35, 36]. Finally, our random hyperbolic generator is competitive with an existing implementation of von Looz et al. [74]. However, in contrast to its competitor, our algorithm has an expected linear running time.

We were also able to experimentally verify our theoretical bounds on the scalability of our generators. All of our generators exhibit very good weak and strong scaling, especially for larger instances. Only for a significantly large number of processor, we were able to see an at most linear increase in running time over all generators. Therefore, we are able to generate instances graph instances with up to 243 vertices and 2^{47} vertices in less than 22 minutes on 32.768 processors. This is competitive to the size of graph instances found in common supercomputer benchmarks such as the Graph 500 benchmark [4].

Finally, we examined the effects of pseudorandomization on the quality of our generated graphs. We did so using common graph properties such as the degree distribution, clustering coefficients and connected component sizes. Our evaluation of these properties shows that pseudorandomization has little to no effect on the quality of our graphs, compared to existing implementations.

7.2 Future Work

Even tough we provided theoretical bounds for the expected running time of our hyperbolic random graph generator, it would be interesting to perform a more thorough analysis of this generator. Especially for the the parallel setting, we are still missing some important theoretical results. It would also be beneficial to perform a more extensive evaluation of this generator for different parameter combinations. This would allow us to better validate the theoretical results presented in this thesis.

Another interesting possibility would be to expand our communication agnostic graph generation paradigm to other network models. This includes community driven models such as the Block Two-level Erdős-Rényi model [67], or structure-driven models such as the Chung-Lu model [8].

Finally, we would like to expand our algorithms to different parallel computation architectures. This includes efficient implementations of our graph generators for shared memory systems and GPUs. As a result, we would be able to provide efficient graph generators for a variety of different application.

A Pseudocode

Algorithm 1: Distributed algorithm for sampling n' elements on processors $\{j, \ldots, k\}$ $(j..k)$ where $i \in \{j, \ldots, k\}$ is the PE executing the function. The initial call on processor i is $sampleP(n, 1..p, i, h)$.

```
1  Function sampleP(n', j..k, i, h)
2      if k − j = 1 then
3          use h(i) to seed the local pseudorandom number generator
4          M := sampleLocally(n', N_i − N_{i−1} + 1)      // e.g. using algorithms H, D
5          return {N_{i−1} + x : x ∈ M}
6      m := ⌊(j+k)/2⌋                                    // middle processor number
7      x := hyperGeometricDeviate(n', N_m − N_j + 1, N_k − N_j + 1, j..k, h)
8      if i ≤ m then
9          return sampleP(x, j..m, i, h)
10     else
11         return sampleP(n' − x, m + 1..k, i, h)
```

Algorithm 2: Algorithm for (sequential) divide-and-conquer sampling of n elements without replacement from a population of N elements.

```
1  Function sampleR(n, N)
2      if n < n_0 then
3          return sampleBase(n, N)                       // e.g. using algorithms H, D
4      x := hyperGeometricDeviate(n, ⌊N/2⌋, N)
5      A := sampleR(x, ⌊N/2⌋)
6      B := sampleR(n − x, N − ⌊N/2⌋)
7      return A ∪ {x + ⌊N/2⌋ : x ∈ B}
```

Algorithm 3: Triangle case for the undirected $G(n, m)$ generator. m' is the number of edges that remain to be sampled. $j..k$ is shorthand for the set of processors $\{j, \ldots, k\}$ responsible for the current set of rows. Likewise, $j'..k'$ is the set of processors remaining for the set of columns. i is the processor ID and h is a given hash function.

Function $generateTriangle(m', j..k, j'..k', i, h)$
  if $(k - j) = 1 \wedge (k' - j') = 1$ **then**
    use $h(j, j')$ to seed the local pseudorandom number generator
    M:= $sampleLocally(m', \frac{(N_j - N_{j-1} + 1) * (N_{j'} - N_{j'-1})}{2}, h)$
    forall $x \in M$ **do**
      v_s:= $\frac{\sqrt{8(x-1)+1}-1}{2}$
      v_t:= $(x - 1) - \frac{v_s(v_s+1)}{2}$
      add $\{v_s + N_{j-1}, v_t + N_{j'-1}\}$
    return
  r:= $\lfloor \frac{k+j}{2} \rfloor$, c:= $\lfloor \frac{k'+j'}{2} \rfloor$
  m:= $\frac{(N_j - N_k + 1) * (N_j - N_k)}{2}$
  // number of edges in each quadrant
  m_2:= $\frac{(N_j - N_r + 1) * (N_{j'} - N_c)}{2}$
  m_3:= $(N_{r+1} - N_k + 1) * (N_{j'} - N_c + 1)$
  m_4:= $\frac{(N_{r+1} - N_k + 1) * (N_{c+1} - N_{k'})}{2}$
  // compute number of samples in each quadrant
  $x_2 = generateHypergeometric(m', m_2, m, j..k, j'..k', h)$
  $x_3 = generateHypergeometric(m' - x_2, m_3, m_3 + m_4, j..k, j'..k', h)$
  if $i < r$ **then**
    $generateTriangle(x_2, j..r, j'..c, i, h)$
    $generateRectangle(x_3, r + 1..k, j'..c, i, h)$
  else
    $generateRectangle(x_3, r + 1..k, j'..c, i, h)$
    $generateTriangle(m' - x_2 - x_3, r + 1..k, c + 1..k', i, h)$

Algorithm 4: Rectangle case for the undirected $G(n, m)$ generator. m' is the number of edge that remain to be sampled. $j..k$ is shorthand for the set of processors $\{j, \ldots, k\}$ responsible for the current set of rows. Likewise, $j'..k'$ is the set of processors remaining for the set of columns. i is the processor ID and h is a given hash function.

Function $generateRectangle(m', j..k, j'..k', i, h)$
  if $(k - j) = 1 \wedge (k' - j') = 1$ **then**
    use $h(j, j')$ to seed the local pseudorandom number generator
    M:= $sampleLocally(m', \frac{(N_j - N_{j-1} + 1) * (N_{j'} - N_{j'-1})}{2}, h)$
    forall $x \in M$ **do**
      v_s:= $\lfloor \frac{x-1}{N_j} \rfloor$
      v_t:= := $(x - 1) \mod N_{j'}$
      add $\{v_s + N_{j-1}, v_t + N_{j'-1}\}$
    return
  r:= $\lfloor \frac{k+j}{2} \rfloor$, c:= $\lfloor \frac{k'+j'}{2} \rfloor$
  m:= $\frac{(N_j - N_k + 1) * (N_j - N_k)}{2}$
  // number of edges in each quadrant
  m_1:= $(N_j - N_r + 1) * (N_{c+1} - N_{k'} + 1)$
  m_2:= $(N_j - N_r + 1) * (N_{j'} - N_c + 1)$
  m_3:= $(N_{r+1} - N_k + 1) * (N_{j'} - N_c + 1)$
  m_4:= $(N_{r+1} - N_k + 1) * (N_{c+1} - N_{k'} + 1)$
  // compute number of samples in each quadrant
  $x = generateHypergeometric(m', m_1 + m_2, m, j..k, j'..k', h)$
  $x_2 = generateHypergeometric(x, m_2, m_1 + m_2, j..k, j'..k', h)$
  $x_3 = generateHypergeometric(m' - x - x_2, m_3, m_3 + m_4, j..k, j'..k', h)$
  if $i > j$ **then** *// only row of chunks remaining*
    if $i < r$ **then**
      $generateRectangle(x_2, r + 1..k, j'..c, i, h)$
      $generateRectangle(x - x_2, j..r, j'..c, i, h)$
    else
      $generateRectangle(x_3, r + 1..k, j'..c, i, h)$
      $generateRectangle(m' - x - x_3, r + 1..k, c + 1..k', i, h)$
  else *// only column of chunks remaining*
    if $i < c$ **then**
      $generateRectangle(x_2, r + 1..k, j'..c, i, h)$
      $generateRectangle(x_3, r + 1..k, j'..c, i, h)$
    else
      $generateRectangle(x - x_2, j..r, j'..c, i, h)$
      $generateRectangle(m' - x - x_3, r + 1..k, c + 1..k', i, h)$

Algorithm 5: Directed $G(n,p)$ generator. We use p_m as the probability for any individual edge to avoid ambiguity. i is the processor ID and h is a given hash function.

Function $generateGnpDirected(p_m, i, h)$
  use $h(i)$ to seed the local pseudorandom number generator
  m':= $binomialDeviate(M_i - M_{i-1} + 1, p_m)$
  M:= $sampleLocally(m', M_i - M_{i-1} + 1)$ *// e.g. using algorithms H, D, or R*
  forall $x \in M$ **do**
    v_s:= $\lfloor \frac{x-1}{M_i} \rfloor$
    v_t:= := $(x-1) \mod (n-1)$ *// n is the total number of nodes*
    add $(v_s + N_{i-1}, v_t + N_{i-1})$

Algorithm 6: Undirected $G(n,p)$ generator. We use p_m as the probability for any individual edge to avoid ambiguity. Processor i generates the rectangle chunks $\{(i,0),\ldots,(i,i-1)\}$ and $\{(i+1,i),\ldots,(p,i)\}$, as well as the triangle chunk (i,i). i is the processor ID and h is a given hash function.

Function $generateTriangle(p_m, i, j, h)$
  use $h(i,j)$ to seed the local pseudorandom number generator
  $M_{i,j}$:= $\frac{(N_i - N_{i-1}+1)*(N_j - N_{j-1})}{2}$
  m':= $binomialDeviate(M_{i,j}, p_m)$
  M:= $sampleLocally(m', M_{i,j}, h)$
  forall $x \in M$ **do**
    v_s:= $\frac{\sqrt{8(x-1)+1}-1}{2}$
    v_t:= := $(x-1) - \frac{v_s(v_s+1)}{2}$
    add $\{v_s + N_{i-1}, v_t + N_{j-1}\}$

Function $generateRectangle(p_m, i, j, h)$
  use $h(i,j)$ to seed the local pseudorandom number generator
  $M_{i,j}$:= $(N_i - N_{i-1} + 1) * (N_j - N_{j-1} + 1)$
  m':= $binomialDeviate(M_{i,j}, p_m)$
  M:= $sampleLocally(m', M_{i,j}, h)$
  forall $x \in M$ **do**
    v_s:= $\lfloor \frac{x-1}{N_i} \rfloor$
    v_t:= := $(x-1) \mod N_j$
    add $\{v_s + N_{i-1}, v_t + N_{j-1}\}$

Algorithm 7: Two dimensional random geometric generator. n' is the number of vertices remaining for the current level. $j..k$ is shorthand for the set of processors $\{j, \ldots, k\}$ responsible for the y-dimension. Likewise, $j'..k'$ is the set of processors handling the x-dimension. i is the processor ID and h is a given hash function.

```
1  Function generate2dRgg(n', r, j..k, j'..k', i, h)
2      if (k − j) = 1 ∧ (k' − j') = 1 then
3          use h(i) to seed the local pseudorandom number generator
4          V := sampleVertices(n', ⌊i/p⌋, i mod p, h)
5          forall v ∈ V do
6              grid_x = ⌊v_x/r⌋, grid_y = ⌊v_y/r⌋
7              add v to grid[grid_x][grid_y]
8          if i is local chunk then
9              generateEdges(i, r)                    // generate intra-chunk edges
10             N := gatherNeighbors(i)
11             forall i' ∈ N do
12                 generate2dRgg(n, r, 1..p, 1..p, i', h)
13                 generateEdges(i, i', r)            // generate inter-chunk edges
14         return
15     h := ⌊(k+j)/2⌋, v := ⌊(k'+j')/2⌋
       // compute number of vertices in each quadrant
16     y = binomialDeviate(n', h/(k+j), j..k, j'..k', h)
17     x_1 = binomialDeviate(y, v/(k'+j'), j..k, j'..k', h)
18     x_2 = binomialDeviate(n' − y, v/(k'+j'), j..k, j'..k', h)
19     if ⌊i/p⌋ < h ∧ (i mod p) ≥ v then
20         generate2dRgg(y − x_1, j..h, v + 1..k', i, h)        // first quadrant
21     else if ⌊i/p⌋ < h ∧ (i mod p) < v then
22         generate2dRgg(x_1, j..h, j'..v, i, h)                // second quadrant
23     else if ⌊i/p⌋ ≥ h ∧ (i mod p) ≥ v then
24         generate2dRgg(x_2, h + 1..k, j'..v, i, h)            // third quadrant
25     else
26         generate2dRgg(n' − y − x_2, h + 1..k, v + 1..k', i, h)   // fourth quadrant
```

Algorithm 8: Sequential random hyperbolic generator. n is the number of vertices. γ is the power-law exponent of the resulting degree distribution, and $\bar{k}$ is the average degree of a node. Additionally, we are given a hash function h to seed the pseudorandom number generators.

```
1  Function generateHyp(n, γ, k̄, h)
2      α := (γ−1)/2
3      R := getTargetRadius(n, k̄, α)
4      A := generateAnnuli(n, R, h)            // generate log n ordered annuli
5      forall a ∈ A do
6          n_a := pointsInAnnulus(a)
7          C_a := generatePoints(n_a, h)       // generate cells for annulus
8      forall a ∈ A do
9          forall c ∈ C_a do
10             forall v ∈ c do                 // iterate over vertices in cell
11                 generateOutwardEdges(v, a, c, R)
12
13 Function generatePoints(n, h)
14     C_a := {c_0, ..., c_max}                // expected constant number of points per cell
15     φ_g = 2π/|C_a|
16     use h(a) to seed the local pseudorandom number generator
17     for v ∈ 0..n_a do                       // iterate over vertices in annulus
18         draw φ_v uniformly from [0, 2π)
19         draw r_v with density f(r) = α sinh(αR)/(cosh(αR) − 1)
20         insert (φ_v, r_v) in suitable cell c_i = ⌊φ_v/φ_g⌋;
21     return C_a
22
23 Function generateOutwardEdges(v, a, c, R)
24     φ_g = 2π/|C_a|
25     start := ⌊φ_v/φ_g⌋                      // compute starting cell
26     min_φ, max_φ := getMinMaxPhi(v, a, R)   // compute boundaries
27     generateEdges(v, C_a, R)                // generate edges from v to points in C_a
28     if start_φ < max_φ then
29         continue with next cell that has a larger angular boundary
30         stop when out of boundary
31     if start_φ > min_φ then
32         continue with next cell that has a smaller angular boundary
33         stop when out of boundary
34     if a < |A| then
35         generateOutwardEdges(v, a + 1, c, R)
```

Algorithm 9: Parallel random hyperbolic generator. n is the number of vertices. γ is the power-law exponent of the resulting degree distribution, and $\bar{k}$ is the average degree of a node. i is the processor ID. Additionally, we are given a hash function h to seed the pseudorandom number generators. Point generation and edge searches are similar to the sequential case.

```
1  Function generateHypParallel(n, γ, k̄, i, h)
2      α := (γ−1)/2
3      R := getTargetRadius(n, k̄, α)
4      A := generateAnnuli(n, R, h)            // generate log n ordered annuli
5      forall a ∈ A do
6          n_a := pointsInAnnulus(a)
7          K_a := generateChunks(n_a, a, 1..p, i)      // generate chunks for annulus
8          forall k ∈ K_a do
9              if k is local chunk of i then
10                 C_{a,k} := generatePoints(n, a, k, h)   // generate cells for local chunk
11     forall a ∈ A do
12         forall k ∈ K_a do
13             if k is local chunk of i then
14                 forall c ∈ C_{a,k} do
15                     forall v ∈ c do                  // iterate over vertices in cell
16                         generateInwardEdges(v, a, k, c, R)
17                         generateOutwardEdges(v, a, k, c, R)
18
19 Function generateChunks(n′, a, j..k, h)
20     if (k-j = 1) then
21         setPointsInChunk(C_{a,k}, n′)
22         return C_{a,k}
23     m := ⌊(j+k)/2⌋                          // middle processor number
24     x := binomialDeviate(n′, m/(k−j), j..k, h)
25     return generateChunks(x, j..m, i, h) ∪ generateChunks(n′ − x, m + 1..k, i, h)
```

B Graph Properties

B.1 Erdos-Renyi Generators

B.1.1 Directed $G(n, m)$

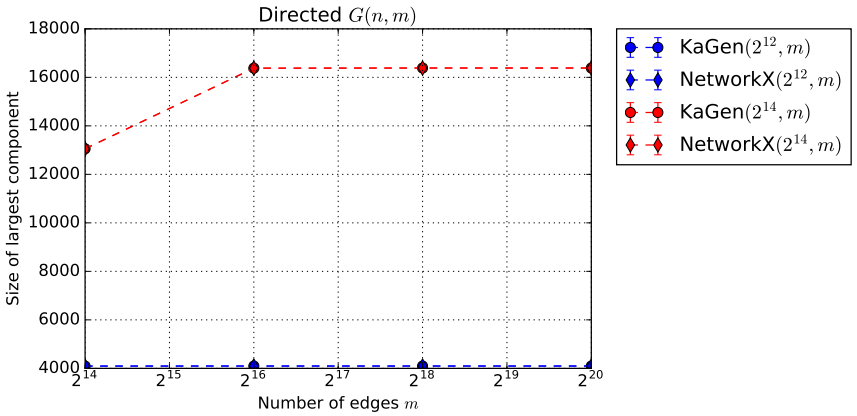


Figure B.1: Size of largest connected component and standard deviation for the directed $G(n, m)$ generators. All results are averaged over 100 iterations.

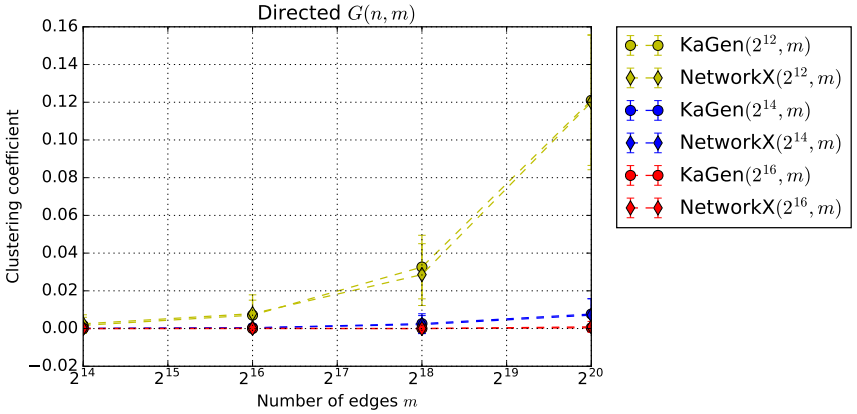


Figure B.2: Clustering coefficients and standard deviation for the directed $G(n, m)$ generators. All results are averaged over 100 iterations.

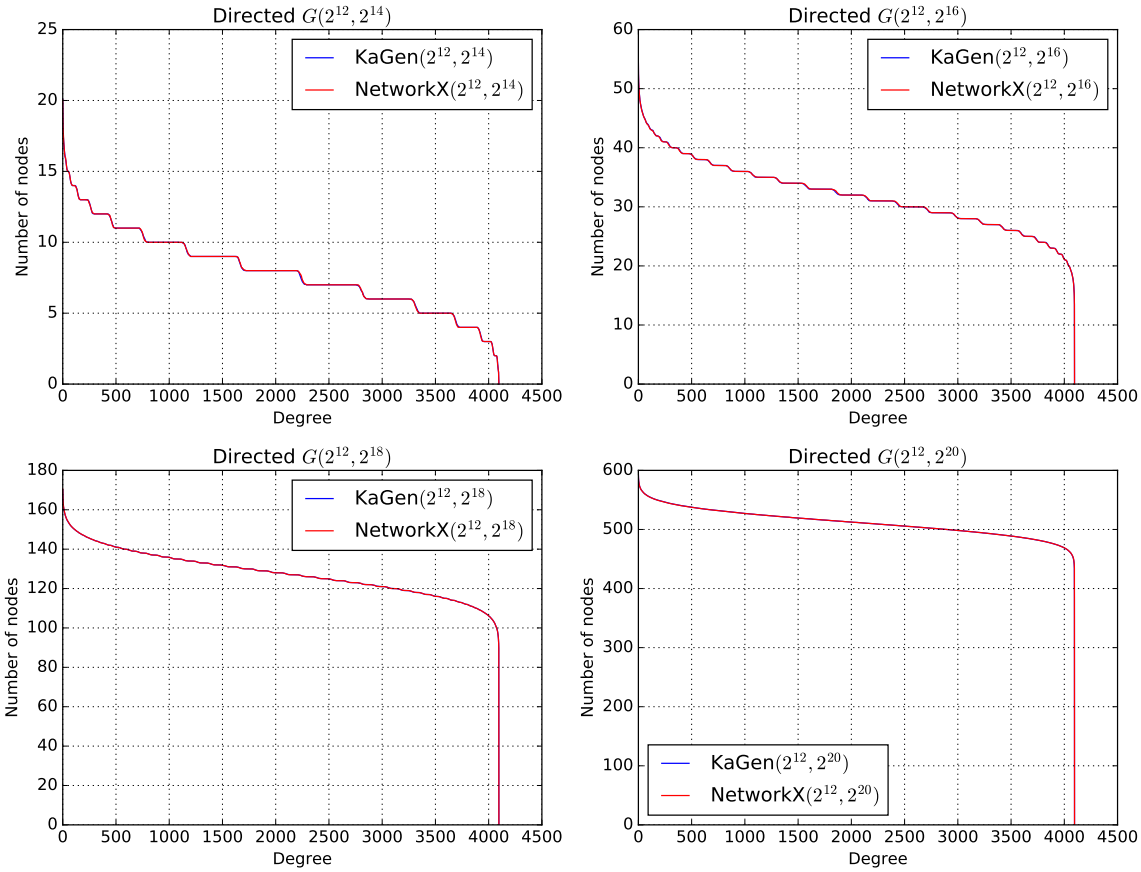


Figure B.3: Degree distribution for the directed $G(n, m)$ generators for $n = 2^{12}$. All results are averaged over 100 iterations.

B.1.2 Undirected $G(n, m)$

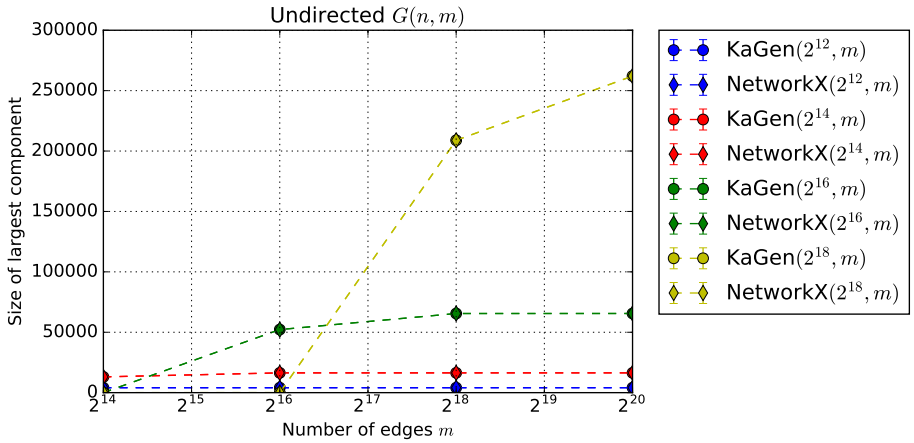


Figure B.4: Size of largest connected component and standard deviation for the undirected $G(n, m)$ generators. All results are averaged over 100 iterations.

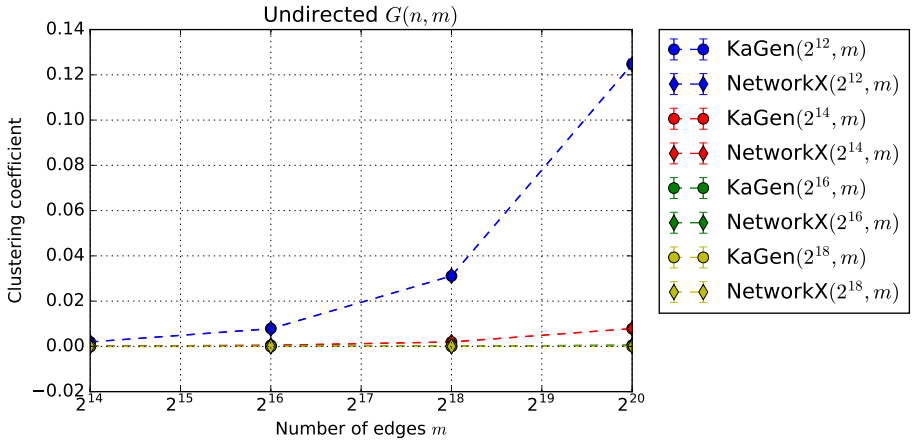


Figure B.5: Clustering coefficients and standard deviation for the undirected $G(n, m)$ generators. All results are averaged over 100 iterations.

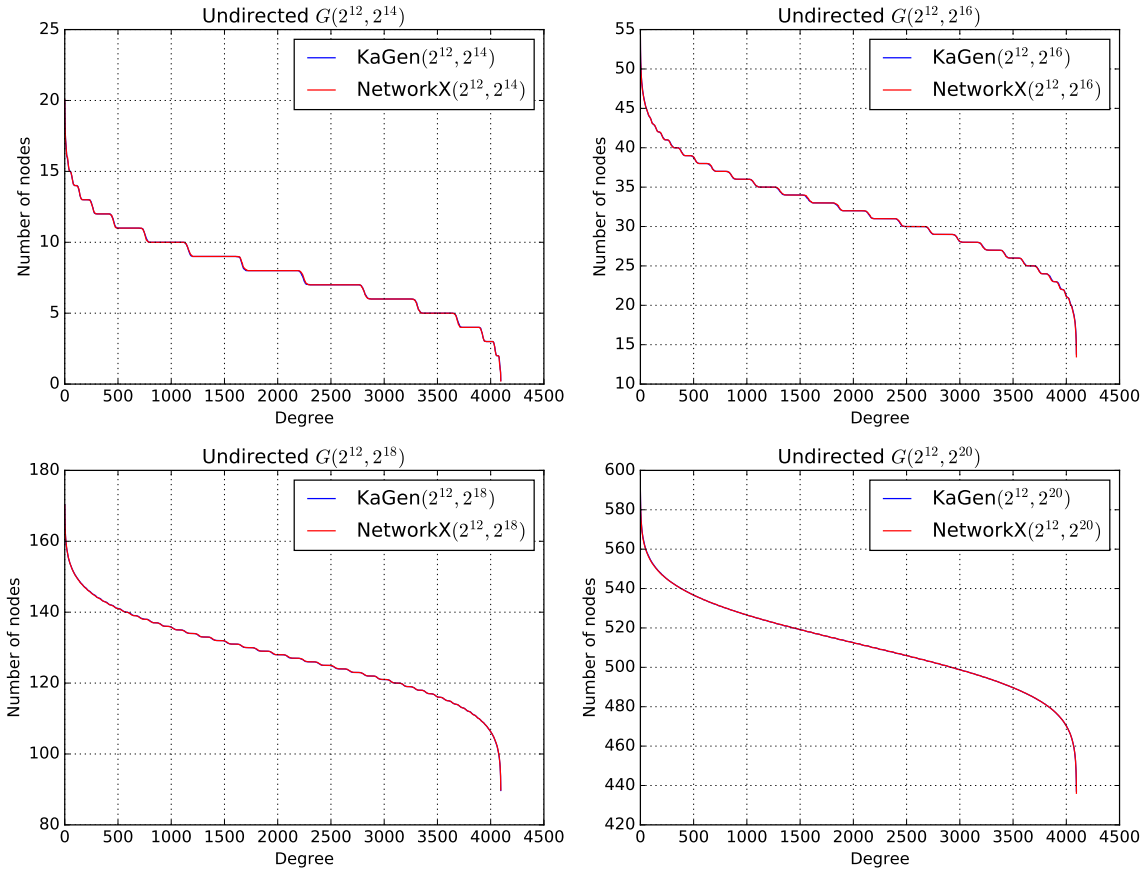


Figure B.6: Degree distribution for the undirected $G(n, m)$ generators for $n = 2^{12}$. All results are averaged over 100 iterations.

B.1.3 Directed $G(n, p)$

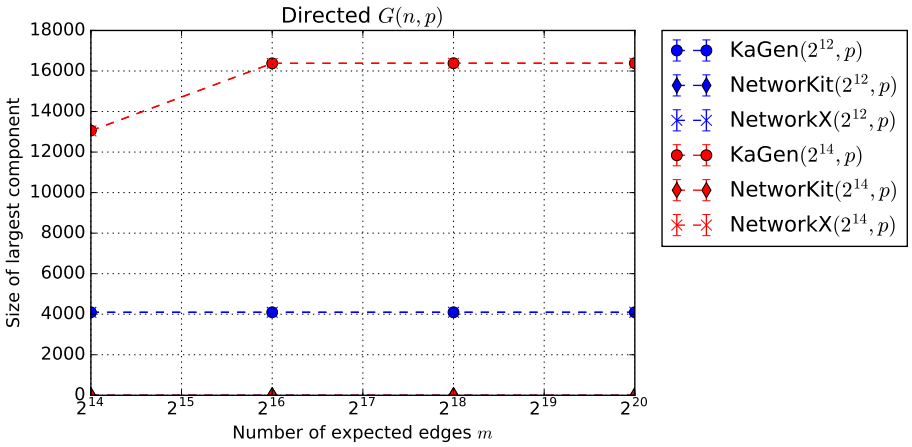


Figure B.7: Size of largest connected component and standard deviation for the directed $G(n, p)$ generators. The NetworKit generator always produces a largest component of size 0. All results are averaged over 100 iterations.

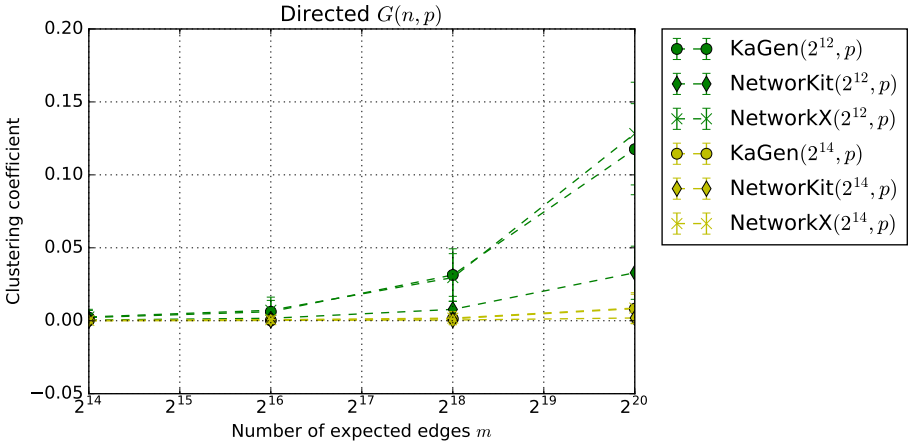


Figure B.8: Clustering coefficients and standard deviation for the directed $G(n, p)$ generators. All results are averaged over 100 iterations.

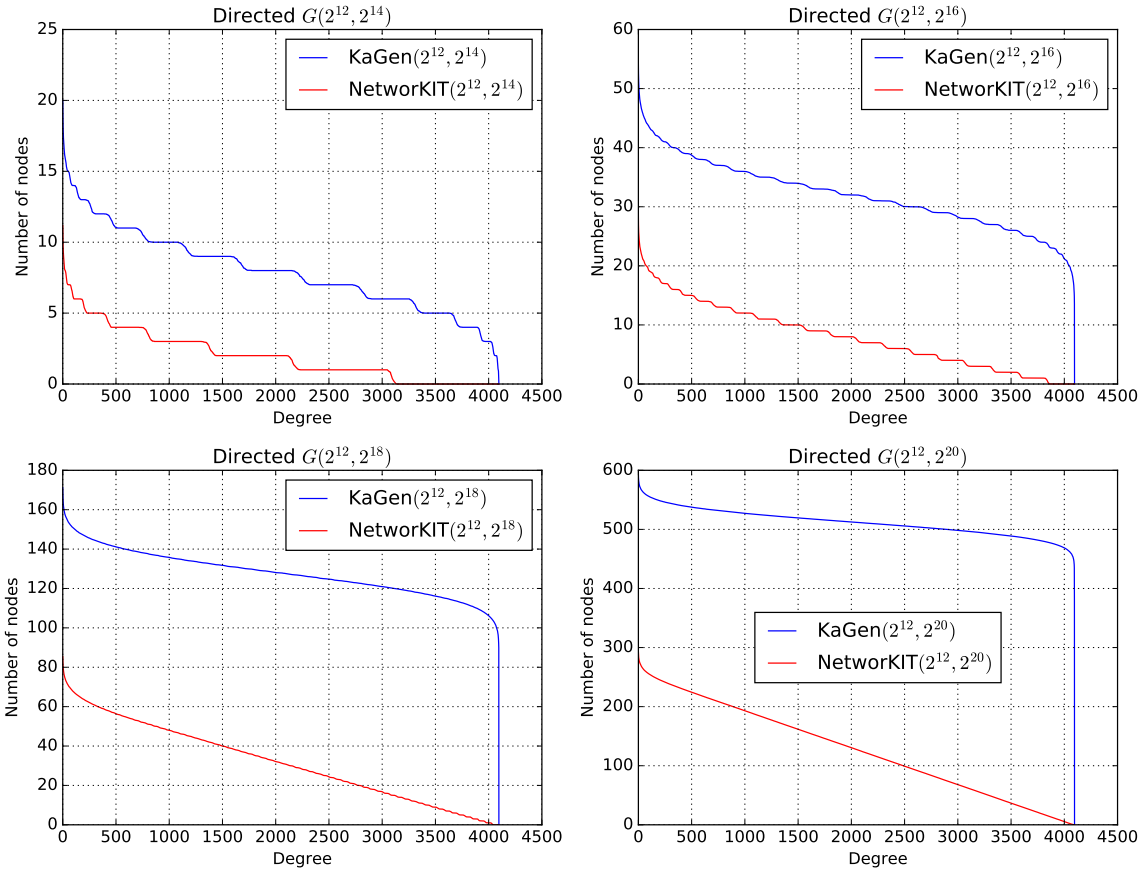


Figure B.9: Degree distribution for the directed $G(n, p)$ generators for $n = 2^{12}$. The NetworKit generator exhibits a different binomial degree distribution than expected in theory. All results are averaged over 100 iterations.

B.1.4 Undirected $G(n, p)$

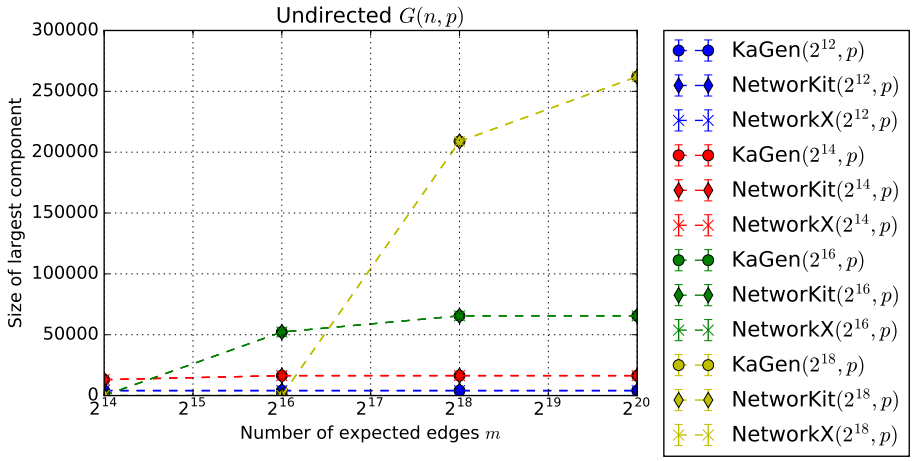


Figure B.10: Size of largest connected component and standard deviation for the undirected $G(n, p)$ generators. All results are averaged over 100 iterations.

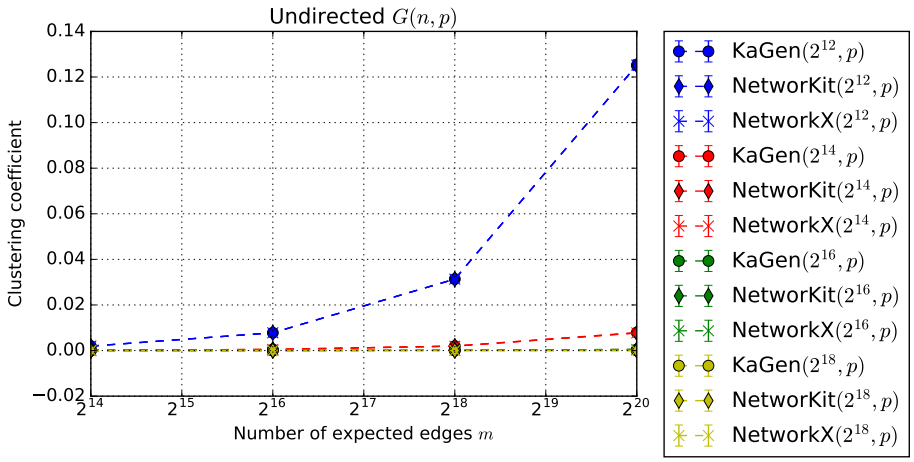


Figure B.11: Clustering coefficients and standard deviation for the undirected $G(n, p)$ generators. All results are averaged over 100 iterations.

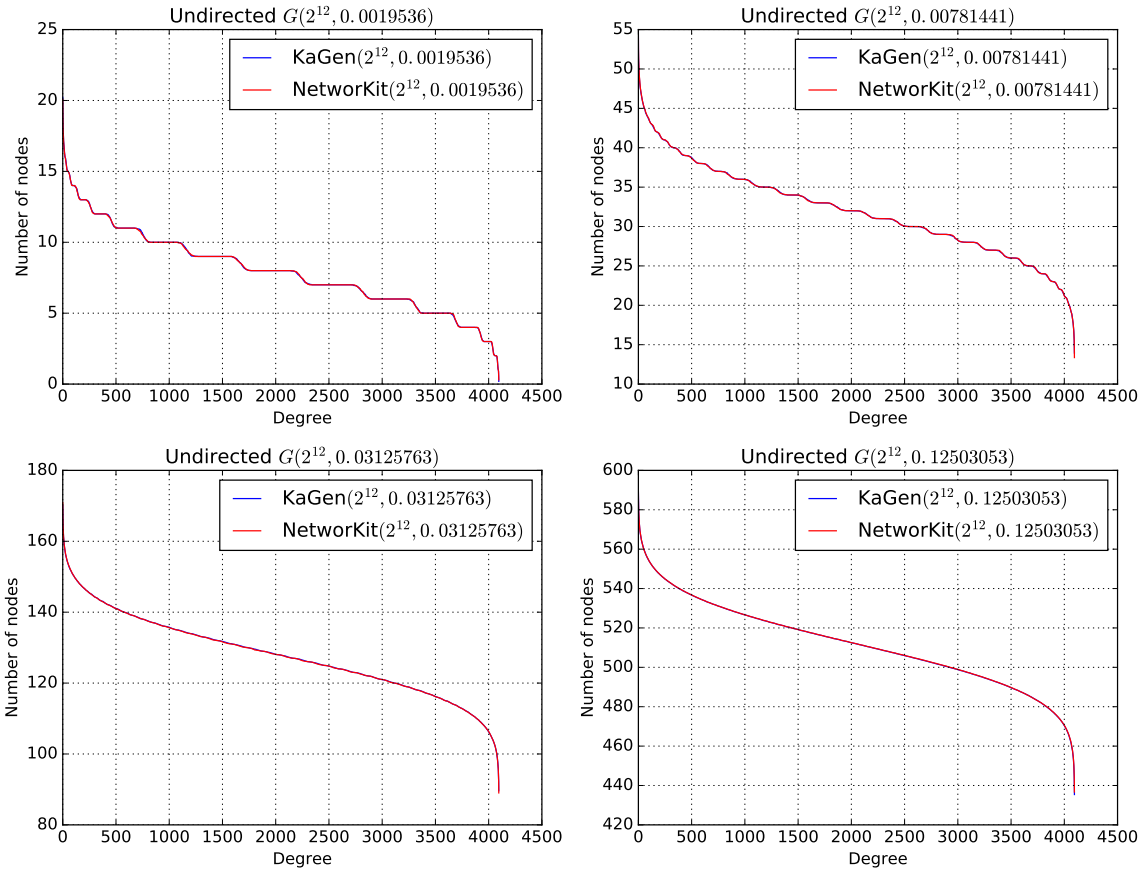


Figure B.12: Degree distribution for the undirected $G(n, p)$ generators for $n = 2^{12}$. All results are averaged over 100 iterations.

B.2 Random Geometric Generators

B.2.1 2D RGG

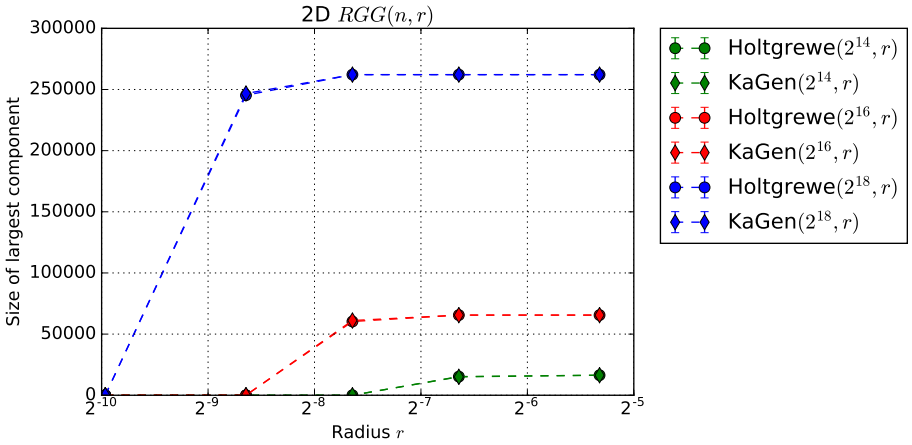


Figure B.13: Size of largest connected component and standard deviation for the 2D RGG generators. All results are averaged over 100 iterations.

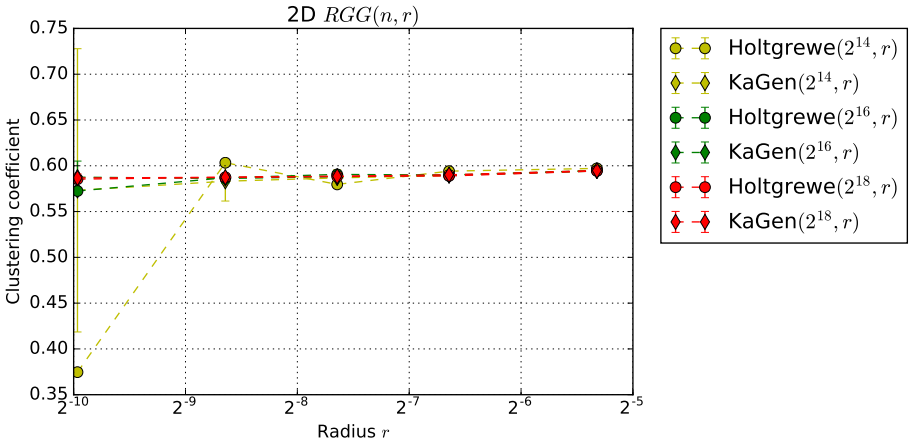


Figure B.14: Clustering coefficients and standard deviation for the 2D RGG generators. All results are averaged over 100 iterations.

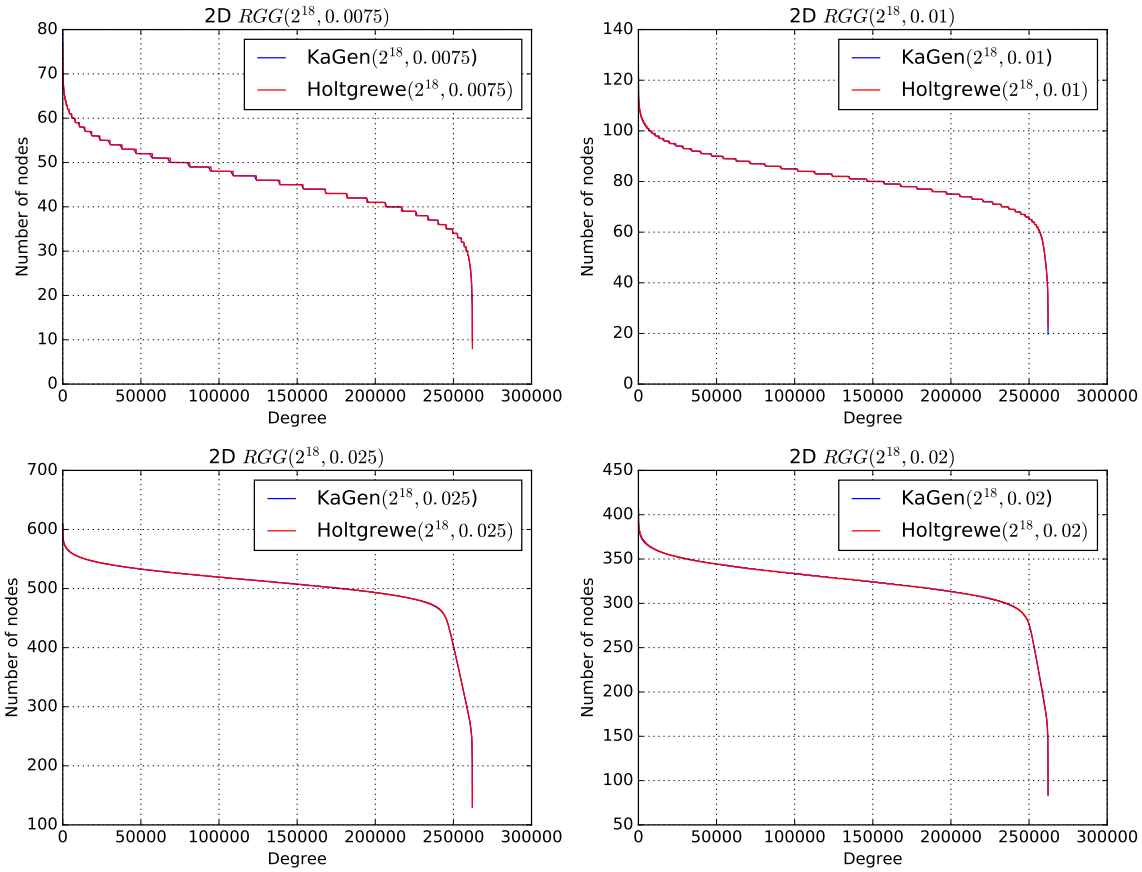


Figure B.15: Degree distribution for the 2D RGG generators for $n = 2^{18}$. All results are averaged over 100 iterations.

B.2.2 3D RGG

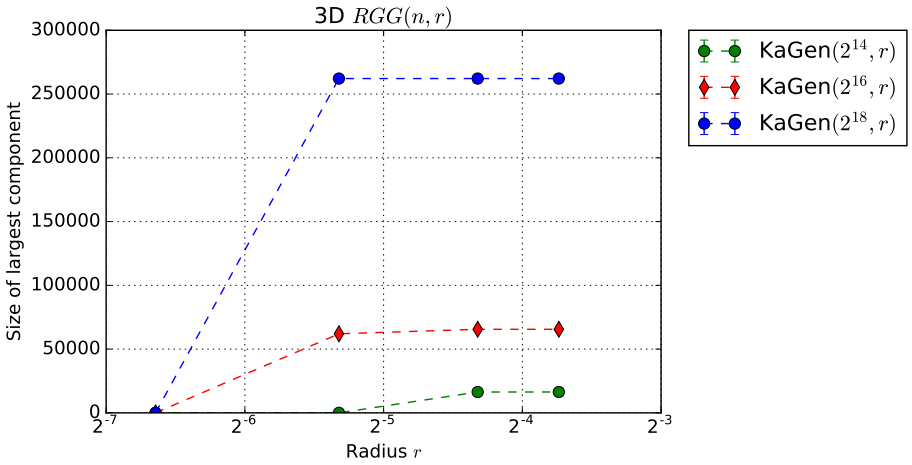


Figure B.16: Size of largest connected component and standard deviation for the 3D RGG generator. All results are averaged over 100 iterations.

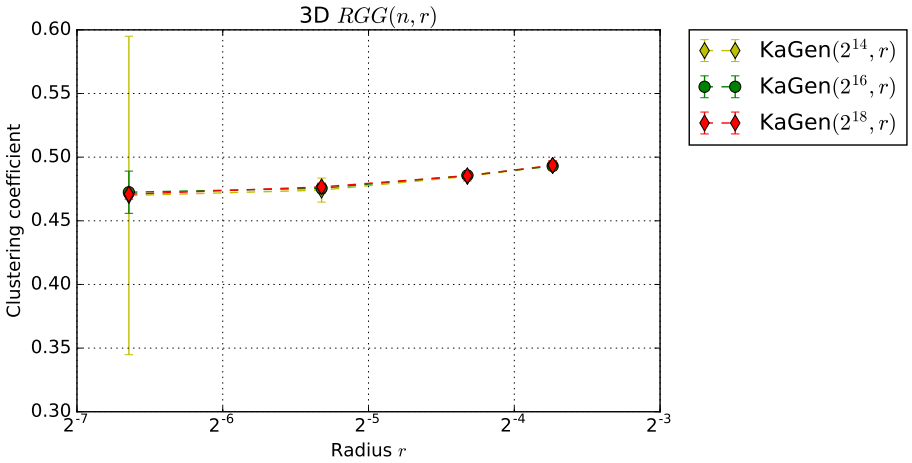


Figure B.17: Clustering coefficients and standard deviation for the 3D RGG generator. All results are averaged over 100 iterations.

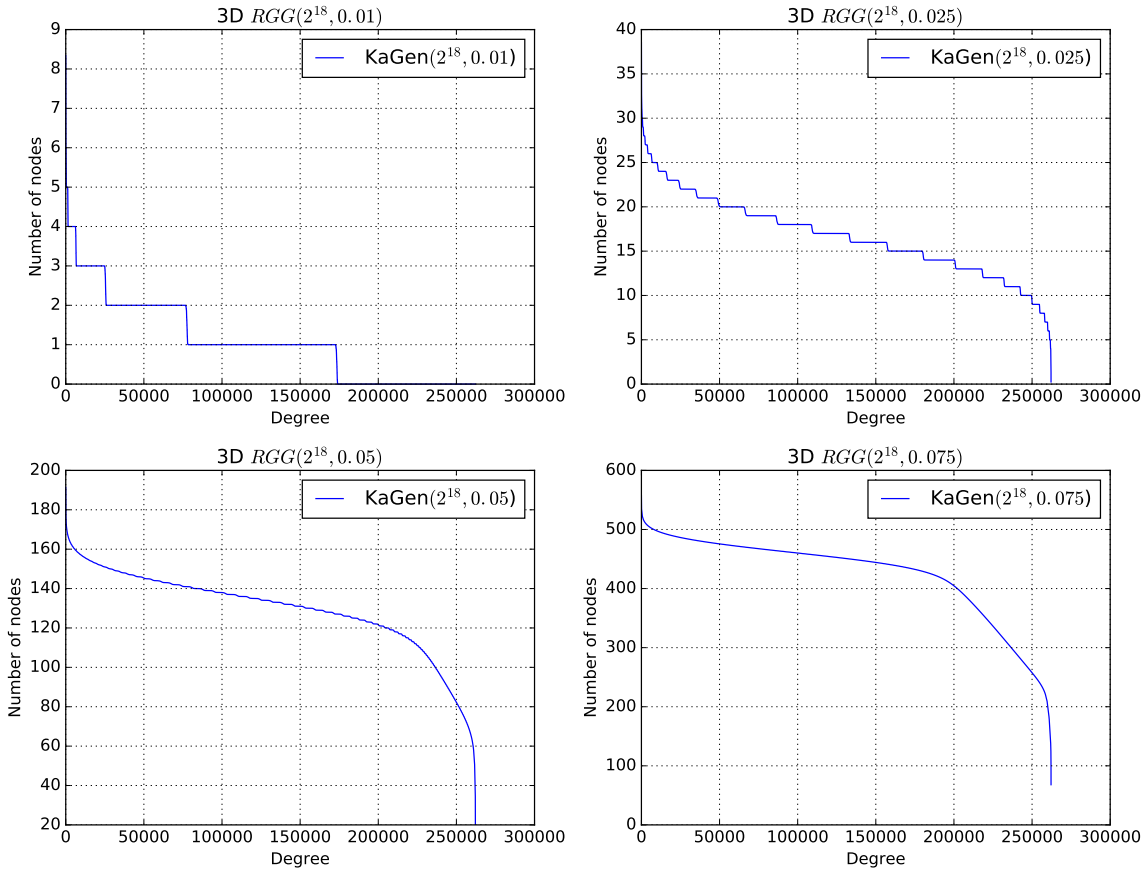


Figure B.18: Degree distribution for the 3D RGG generator for $n = 2^{18}$. All results are averaged over 100 iterations.

B.3 Random Hyperbolic Generators

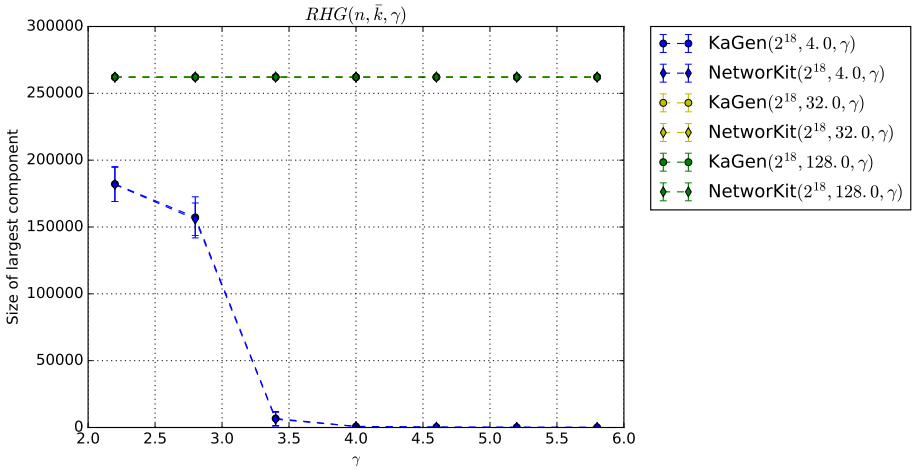


Figure B.19: Size of largest connected component and standard deviation for the RHG generators and $n = 2^{18}$. All results are averaged over 100 iterations.

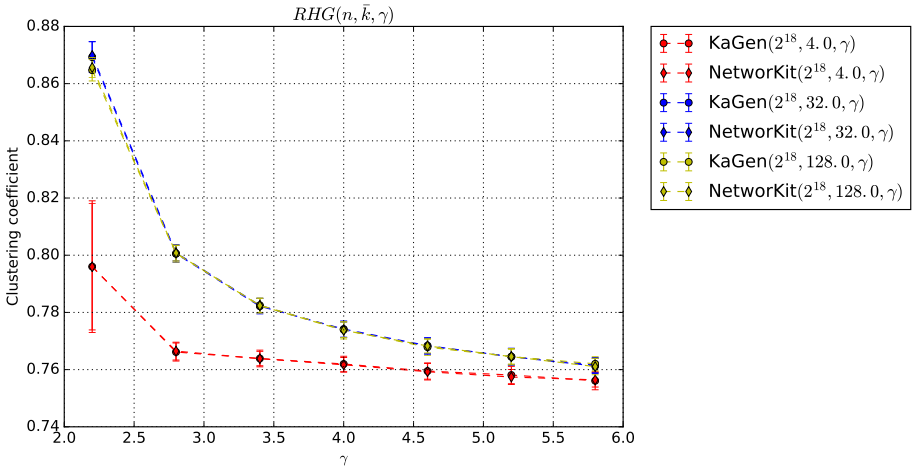


Figure B.20: Clustering coefficients and standard deviation for the RHG generators and $n = 2^{18}$. All results are averaged over 100 iterations.

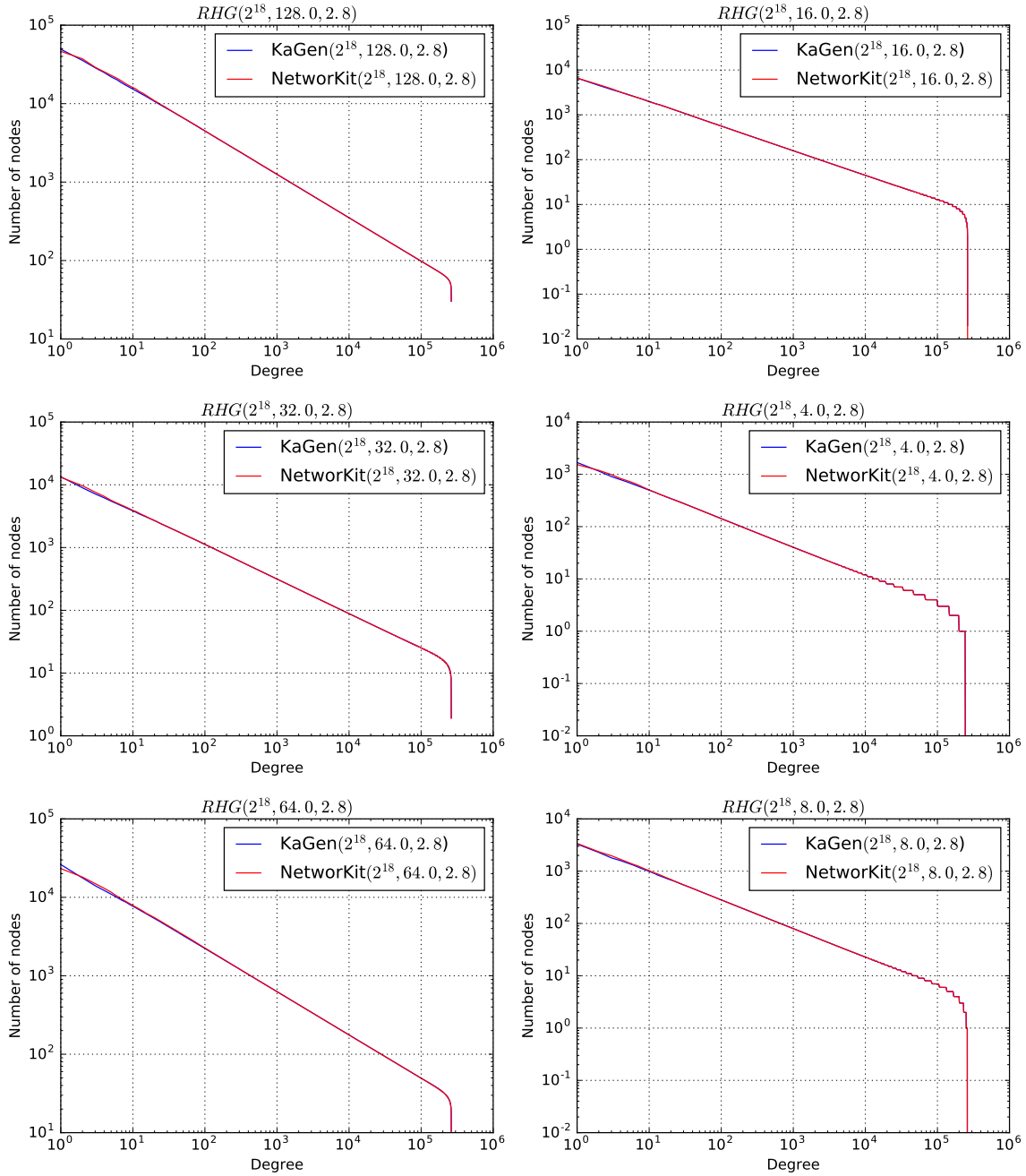


Figure B.21: Degree distribution for the RHG generators for $n = 2^{18}$. All results are averaged over 100 iterations.

Bibliography

[1] Choosing a good hash function, part 2. https://research.neustar.biz/2011/12/29/choosing-a-good-hash-function-part-2. Accessed: 2016-10-05.

[2] Choosing a good hash function, part 3. https://research.neustar.biz/2012/02/02/choosing-a-good-hash-function-part-3. Accessed: 2016-10-05.

[3] GNU scientific library. https://www.gnu.org/software/gsl/. Accessed: 2016-10-05.

[4] Graph 500 benchmark. http://www.graph500.org. Accessed: 2016-10-05.

[5] Number of monthly active Facebook users worldwide as of 2nd quarter 2016. https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/. Accessed: 2016-10-05.

[6] The size of the World Wide Web (The Internet). http://www.worldwidewebsize.com. Accessed: 2016-10-05.

[7] J. H. Ahrens and U. Dieter. Sequential random sampling. *ACM Trans. Math. Softw.*, 11(2):157–169, 1985.

[8] W. Aiello, F. R. K. Chung, and L. Lu. A random graph model for massive graphs. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 171–180. ACM, 2000.

[9] F. Amblard. Linked: The new science of networks *by Albert-László Barabási*. *J. Artificial Societies and Social Simulation*, 6(2), 2003.

[10] A. Arenas, A. Díaz-Guilera, J. Kurths, Y. Moreno, and C. Zhou. Synchronization in complex networks. *Physics Reports*, 469(3):93 – 153, 2008.

[11] L. Arge. The Buffer Tree: A new technique for optimal I/O-algorithms. In *Algorithms and Data Structures, 4th International Workshop, WADS '95, Kingston, Ontario, Canada, August 16-18, 1995, Proceedings*, volume 955 of *Lecture Notes in Computer Science*, pages 334–345. Springer, 1995.

[12] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[13] V. Batagelj and U. Brandes. Efficient generation of large random networks. *Phys. Rev. E*, 71:036113, Mar 2005.

[14] O. Batarfi, R. E. Shawi, A. G. Fayoumi, R. Nouri, S. Beheshti, A. Barnawi, and S. Sakr. Large scale graph processing systems: Survey and an experimental evaluation. *Cluster Computing*, 18(3):1189–1213, 2015.

[15] J. Bernoulli. *Ars conjectandi*. Impensis Thurnisiorum, fratrum, 1713.

[16] T. Bläsius, T. Friedrich, A. Krohmer, and S. Laue. Efficient embedding of scale-free graphs in the hyperbolic plane. In *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark*, volume 57 of *LIPIcs*, pages 16:1–16:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[17] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[18] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D.-U. Hwang. Complex networks: Structure and dynamics. *Physics Reports*, 424(4–5):175 – 308, 2006.

[19] C. E. Bonferroni. *Teoria statistica delle classi e calcolo delle probabilita*. Libreria internazionale Seeber, 1936.

[20] K. Bringmann, R. Keusch, and J. Lengler. Geometric inhomogeneous random graphs. *CoRR*, abs/1511.00576, 2015.

[21] D. Chakrabarti and C. Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Comput. Surv.*, 38(1), 2006.

[22] H. Chen, J. Schroeder, R. V. Hauck, L. Ridgeway, H. Atabakhsh, H. Gupta, C. Boarman, K. Rasmussen, and A. W. Clements. COPLINK connect: Information and knowledge management for law enforcement. *Decision Support Systems*, 34(3):271–285, 2003.

[23] K. Choromanski, M. Matuszak, and J. Miekisz. Scale-free graph with preferential attachment and evolving internal vertex structure. *Journal of Statistical Physics*, 151(6):1175–1183, 2013.

[24] P. Erdős and A. Rényi. On Random Graphs I. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959 1959.

[25] C. Fan, M. E. Muller, and I. Rezucha. Development of sampling plans by using sequential (item by item) selection techniques and digital computers. *Journal of the American Statistical Association*, 57(298):387–402, 1962.

[26] H. Feistel. Cryptography and computer privacy. *Scientific American*, 228:15–23, 1973.

[27] R. A. Finkel and U. Manber. DIB - A distributed implementation of backtracking. *ACM Trans. Program. Lang. Syst.*, 9(2):235–256, 1987.

[28] R. A. S. Fisher and F. Yates. *Statistical tables for biological, agricultural, and medical research*. Edinburgh Oliver and Boyd, 6th ed., rev. and enlarged edition, 1963. Bibliography: p. 41-43.

[29] A. S. Garge and S. A. Shirali. Triangular numbers. *Resonance*, 17(7):672–681, 2012.

[30] E. N. Gilbert. Random graphs. *Ann. Math. Statist.*, 30(4):1141–1144, 12 1959.

[31] J. Goldstein. Emergence as a construct: History and issues. *Emergence*, 1(1):49–72, 1999.

[32] L. Gugelmann, K. Panagiotou, and U. Peter. Random hyperbolic graphs: Degree sequence and clustering. In *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part II*, volume 7392 of *Lecture Notes in Computer Science*, pages 573–585. Springer, 2012.

[33] T. Hagerup and C. Rüb. A guided tour of Chernoff bounds. *Inf. Process. Lett.*, 33(6):305–308, 1990.

[34] T. L. Heath et al. *The thirteen books of Euclid's Elements*. Courier Corporation, 1956.

[35] M. Holtgrewe. *A scalable coarsening phase for a multi-level graph partitioning algorithm*. PhD thesis, University of Karlsruhe, 2009.

[36] M. Holtgrewe, P. Sanders, and C. Schulz. Engineering a scalable high quality graph partitioner. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*, pages 1–12. IEEE, 2010.

[37] C. Huygens. *De ratiociniis in ludo aleae*. Ex officinia J. Elsevirii, 1657.

[38] Intel. Intel digital random number generator (DRNG): Software implementation guide. `https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide`, 2012.

[39] C. Jacoboni and L. Reggiani. The Monte Carlo method for the solution of charge transport in semiconductors with applications to covalent materials. *Rev. Mod. Phys.*, 55:645–705, Jul 1983.

[40] X. Jia. Wireless networks and random geometric graphs. In *7th International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN 2004), 10-12 May 2004, Hong Kong, SAR, China*, pages 575–580. IEEE Computer Society, 2004.

[41] J. M. Kleinberg. The small-world phenomenon: An algorithmic perspective. In F. F. Yao and E. M. Luks, editors, *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 163–170. ACM, 2000.

[42] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[43] D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981.

[44] D. V. Krioukov, F. Papadopoulos, M. Kitsak, A. Vahdat, and M. Boguñá. Hyperbolic geometry of complex networks. *CoRR*, abs/1006.5169, 2010.

[45] P. L'Ecuyer and R. Simard. TestU01: A C library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, 33(4):22:1–22:40, Aug. 2007.

[46] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos. *Realistic, Mathematically Tractable Graph Generation and Evolution, Using Kronecker Multiplication*, pages 133–145. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[47] G. Marsaglia. DIEHARD: A battery of tests of randomness. 1996.

[48] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[49] A. Medina, A. Lakhina, I. Matta, and J. Byers. BRITE: An approach to universal topology generation. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2001. Proceedings. Ninth International Symposium on*, pages 346–353. IEEE, 2001.

[50] U. Meyer and M. Penschuck. Generating massive scale-free networks under resource constraints. pages 39–52, 2016.

[51] C. Z. Mooney. *Monte Carlo simulation*, volume 116. Sage Publications, 1997.

[52] M. Newman. Random graphs as models of networks. *eprint arXiv:cond-mat/0202208*, Feb. 2002.

[53] M. Newman, A.-L. Barabasi, and D. J. Watts. *The Structure and Dynamics of Networks: (Princeton Studies in Complexity)*. Princeton University Press, Princeton, NJ, USA, 2006.

[54] S. Nobari, X. Lu, P. Karras, and S. Bressan. Fast random graph generation. In *EDBT 2011, 14th International Conference on Extending Database Technology, Uppsala, Sweden, March 21-24, 2011, Proceedings*, pages 331–342. ACM, 2011.

[55] J.-P. Onnela, J. Saramäki, J. Hyvönen, G. Szabó, D. Lazer, K. Kaski, J. Kertész, and A.-L. Barabási. Structure and tie strengths in mobile communication networks. *Proceedings of the National Academy os Sciences (USA)*, 104(7332), 2007.

[56] J. O'Madadhain, D. Fisher, S. White, and Y. Boey. The Jung (Java universal network/graph) framework. *University of California, Irvine, California*, 2003.

[57] R. Pastor-Satorras and A. Vespignani. Epidemic spreading in scale-free networks. *Phys. Rev. Lett.*, 86:3200–3203, Apr 2001.

[58] M. Penrose. *Random geometric graphs*. Number 5. Oxford University Press, 2003.

[59] M. Raab and A. Steger. "balls into bins" - A simple and tight analysis. In *Randomization and Approximation Techniques in Computer Science, Second International Workshop, RANDOM'98, Barcelona, Spain, October 8-10, 1998, Proceedings*, volume 1518 of *Lecture Notes in Computer Science*, pages 159–170. Springer, 1998.

[60] C. Robert and G. Casella. *Monte Carlo statistical methods*. Springer Science & Business Media, 2013.

[61] D. W. O. Rogers, B. A. Faddegon, G. X. Ding, C.-M. Ma, J. We, and T. R. Mackie. BEAM: A Monte Carlo code to simulate radiotherapy treatment units. *Medical Physics*, 22(5):503–524, 1995.

[62] M. Saito and M. Matsumoto. *A PRNG Specialized in Double Precision Floating Point Numbers Using an Affine Transition*, pages 589–602. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[63] P. Sanders. *Lastverteilungsalgorithmen für parallele Tiefensuche*. PhD thesis, University of Karlsruhe, 1996.

[64] P. Sanders, S. Lamm, L. Hübschle-Schneider, E. Schrade, and C. Dachsbacher. Efficient random sampling - Parallel, vectorized, cache-efficient, and online. *CoRR*, abs/1610.05141, 2016.

[65] P. Sanders and C. Schulz. Scalable generation of scale-free graphs. *Inf. Process. Lett.*, 116(7):489–491, 2016.

[66] D. A. Schult. Exploring network structure, dynamics, and function using NetworkX. In *In Proceedings of the 7th Python in Science Conference (SciPy)*, pages 11–15, 2008.

[67] C. Seshadhri, T. G. Kolda, and A. Pinar. Community structure and scale-free collections of Erdös-Rényi graphs. *CoRR*, abs/1112.3644, 2011.

[68] E. Stadlober. Ratio of uniforms as a convenient method for sampling from classical discrete distributions. pages 484–489, 1989.

[69] E. Stadlober and H. Zechner. The Patchwork rejection technique for sampling from unimodal distributions. *ACM Trans. Model. Comput. Simul.*, 9(1):59–80, 1999.

[70] C. Staudt, A. Sazonovs, and H. Meyerhenke. NetworKit: An interactive tool suite for high-performance network analysis. *CoRR*, abs/1403.3005, 2014.

[71] S. Strogatz. *Sync: The emerging science of spontaneous order*. Hyperion, 2003.

[72] J. S. Vitter. An efficient algorithm for sequential random sampling. *ACM Trans. Math. Softw.*, 13(1):58–67, 1987.

[73] M. von Looz, H. Meyerhenke, and R. Prutkin. Generating random hyperbolic graphs in subquadratic time. In *Algorithms and Computation - 26th International Symposium, ISAAC 2015, Nagoya, Japan, December 9-11, 2015, Proceedings*, volume 9472 of *Lecture Notes in Computer Science*, pages 467–478. Springer, 2015.

[74] M. von Looz, M. S. Özdayi, S. Laue, and H. Meyerhenke. Generating massive complex networks with hyperbolic geometry faster in practice. pages 1–6, 2016.

[75] J. Von Neumann. 13. Various techniques used in connection with random digits. 1951.

[76] W.-X. Wang, B.-H. Wang, C.-Y. Yin, Y.-B. Xie, and T. Zhou. Traffic dynamics based on local routing protocol on a scale-free network. *Phys. Rev. E*, 73:026111, Feb 2006.

[77] E. Zegura. GT-ITM: Georgia Tech internetwork topology models (software). *Georgia Tech," http://www. cc. gatech. edu/fac/Ellen. Zegura/gt-itm/gt-itm/tar. gz*, 1996.