Geometric Inhomogeneous Random Graphs for Algorithm Engineering

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

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

Dissertation

von

Christopher Weyand

Tag der mündlichen Prüfung:	21. April 2023
1. Referent:	T.T. Prof. Dr. Thomas Bläsius
2. Referent:	Prof. Dr. Ulrich Meyer

Summary

The design and analysis of graph algorithms is heavily based on the worst case. In practice, however, many algorithms perform much better than the worst case would suggest. Furthermore, various problems can be tackled more efficiently if one assumes the input to be, in a sense, realistic. The field of network science, which studies the structure and emergence of real-world networks, identifies locality and heterogeneity as two frequently occurring properties.

A popular model that captures these properties are geometric inhomogeneous random graphs (GIRGs), which is a generalization of hyperbolic random graphs (HRGs). Aside from their importance to network science, GIRGs can be an immensely valuable tool in algorithm engineering. Since they convincingly mimic real-world networks, guarantees about quality and performance of an algorithm on instances of the model can be transferred to real-world applications. They have model parameters to control the amount of heterogeneity and locality, which allows to evaluate those properties in isolation while keeping the rest fixed. Moreover, they can be efficiently generated which allows for experimental analysis. While realistic instances are often rare, generated instances are readily available. Furthermore, the underlying geometry of GIRGs helps to visualize the network, e.g., for debugging or to improve understanding of its structure.

The aim of this work is to demonstrate the capabilities of geometric inhomogeneous random graphs in algorithm engineering and establish them as routine tools to replace previous models like the Erdős-Rényi model, where each edge exists with equal probability. We utilize geometric inhomogeneous random graphs to design, evaluate, and optimize efficient algorithms for realistic inputs. In detail, we provide the currently fastest sequential generator for GIRGs and HRGs and describe algorithms for maximum flow, directed spanning arborescence, cluster editing, and hitting set. For all four problems, our implementations beat the state-of-the-art on realistic inputs.

On top of providing crucial benchmark instances, GIRGs allow us to obtain valuable insights. Most notably, our efficient generator allows us to experimentally show sublinear running time of our flow algorithm, investigate the solution structure of cluster editing, complement our benchmark set of arborescence instances with a density for which there are no real-world networks available, and generate networks with adjustable locality and heterogeneity to reveal the effects of these properties on our algorithms.

Contents

1 Introduction

Networks are a very general concept and appear almost everywhere. From the friendships that we form, the roads that we take, to the news that we read — almost everything can be modelled as a network with entities and their relations; vertices connected by edges. One of the most prominent examples are social networks like Twitter and Facebook but also atoms connecting to form molecules or protein interactions can be seen as a network. Even this very text is a network of pages and cross references between them. It comes as no surprise that many real-world tasks can be boiled down to a concise problem statement on a network: finding the shortest trip between two cities, separating students into two groups, distributing assignments among coworkers, or tracking down the origin of a disease. Depending on the exact circumstances, these problems could map to the graph theoretic problems shortest path, minimum cut, bipartite matching, and directed spanning tree. For decades now, researchers have been working on algorithms to solve problems like these.

The performance of an algorithm is usually measured as a function of its input size. To focus on the essentials, big-O notation is a formalization of time complexity that ignores constant factors and classifies performance based on asymptotic scaling alone. The implications of this runtime complexity cannot be overstated. Since networks tend to be quite large — e.g., Wikipedia has millions of English articles — the asymptotic scaling of the complexity makes the differences between practical and impractical algorithms. For example, an algorithm that scales cubically in the input size would take decades to process a million-node network whereas an algorithm with quadratic scaling finishes in under an hour. Some problems are even harder to solve with all known algorithms running in exponential time. For example *vertex cover* asks for the smallest set of vertices such that each edge has at least one endpoint selected. Unless verification in polynomial time is equal to solving in polynomial time (i.e., $P = NP$), a polynomial time algorithm for such problems cannot exist. Those NP-hard problems are considered intractable, because the time to solve an instance with just a hundred nodes would, at least in theory, surpass the lifetime of the computer it runs on. These theoretical results stand in contrast to empirical results for classical NP-hard problems like *boolean satisfiability* (SAT) which asks if a given boolean formula in conjunctive normal form has a satisfying assignment. Many problems can be reduced to SAT and the resulting instances, even if they have millions of variables and clauses, are often solved by state-of-the-art SAT solvers in seconds.

Algorithm design and analysis focuses on the worst possible input (worst-case) to provide a running time guarantee that holds for all inputs. But an algorithm does not necessarily take the same amount of time on each network of the same size. For example, if the network has no cycles, then vertex cover is rather easy and solvable in linear time. Moreover, many algorithms perform much better in practice than the worst case would suggest. Realistic inputs are usually far easier than instances that are specifically crafted to be difficult. To bridge the gap between theory and practice and to better formalize algorithm performance and instance difficulty is an open problem and the focus of multiple fields of research.

Parametrized complexity is an approach to better describe the time complexity of algorithms. The idea is to find parameters of the input instance, most commonly solution size, and then express the running time of an algorithm in terms of that parameter in an attempt to extract the difficult core of a problem. A problem is called *fixed parameter tractable* (FPT) if the complexity can be expressed as an arbitrary function in the parameter but with a polynomial dependence on the input size. Instances with a small parameter thus allow for efficient algorithms. Vertex cover, for example, is FPT in the solution size, as well as in a parameter called *treewidth* which describes how similar the instance is to a tree. This nicely generalizes the statement above that vertex cover is fast on graphs without cycles. Unfortunately, most real-world networks have neither a small solution size for vertex cover nor a small treewidth. Despite this, solvers were able to solve vertex cover on instances with thousands of nodes in the *Parameterized Algorithms and Computational Experiments Challenge* (PACE) in 2019. It is an ongoing challenge of parametrized complexity to find parameters that are small in practice, allow FPT algorithms for relevant problems, and can be computed efficiently.

Another early attempt to tackle the gap between theory and practice is average-case analysis, that is, evaluating performance based on the average over all possible inputs instead of the worst case. Equivalently, one considers the expected performance on inputs drawn at random. Each edge has the same probability of existing and each graph with a fixed number of vertices and edges is equally likely. Such a uniform distribution over graphs of a fixed size is called the Erdős-Rényi random graph model, or ER-model for short [ER59]. Although the ER-model has been analyzed extensively and is used in theory and in practice to evaluate existing algorithms, average-case analysis is regarded as unrealistic because real-world networks are, in addition to being far from the worst case, also not entirely random. In fact, similar properties frequently appear in networks from vastly different domains, which are absent (with high probability) in the ER-model. These structures are exploitable and facilitate the design and creation of algorithms that are efficient on real-world data. To obtain meaningful results with average-case analysis, a realistic *network model*, i.e., a probability distribution over the input graphs, is key.

The scientific field that deals with the structure and emergence of real-world networks is called network science. Since the beginnings of average-case analysis, network science has made huge progress characterizing important properties and formulating network models reflecting these properties. There are three properties that occur very frequently. First, many networks have local structure or clustering, meaning that two vertices with a common neighbor are more likely to be connected than two arbitrary vertices. We call this *locality*. Locality is typically measured by the *clustering coefficient*, which counts the number of closed triangles divided by the total number of triangles in a graph. In other words, for two neighbors of a random vertex, the probability that they are joined by an edge. Figure 1.1 shows a graph with random connections (ER model) beside a

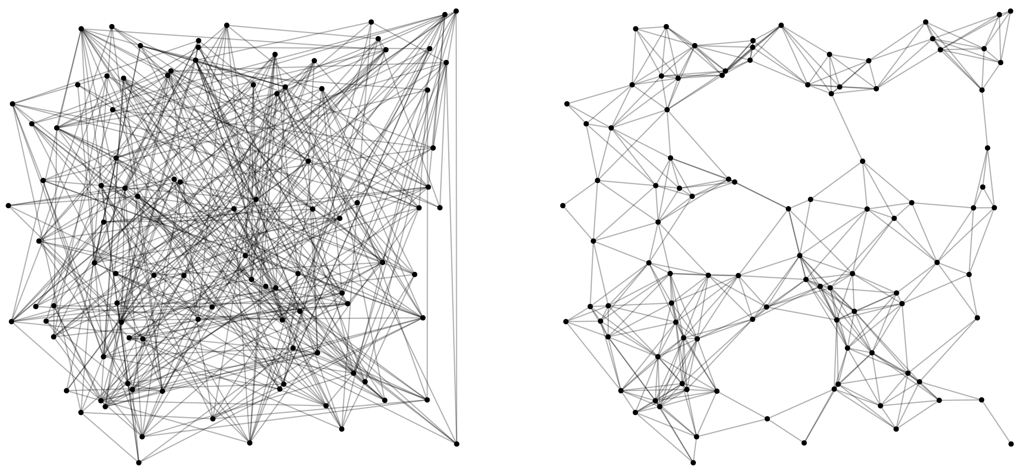

(a) random (b) locality

Figure 1.1: Networks with 100 vertices and average degree eight. The left graph has random connections, while the right graph connects vertices that are close.

graph with high locality (random geometric model). In real-world networks, locality can emerge naturally due to an underlying geometry. Two people living in the same neighborhood are certainly more probable to be friends on Facebook than two arbitrary people on earth. Thus, in social networks, geographic proximity facilitates a community structure. However, locality can appear for a variety of reasons. In actor-collaboration networks, local structure arises due to the genres of the movies. Actors like Jim Carrey are known for their roles in comedy movies and are more likely to appear in movies with other comedy actors.

The second property is the *small-world* property. It states that the shortest path between two vertices is very short on average. The urban legend *six degrees of separation* claims that all people know each other over at most six “friend of a friend” relations. In the late 60s, a popular study by social psychologist Stanley Milgram investigated the average path length of social networks between people in the United States [Mil67]. Although the concept of six degrees of separation is not entirely true, it still captures the astonishing connectedness of many real-world networks. A website called six degrees of Wikipedia allows the user to enter two Wikipedia articles and then finds the shortest path of hyperlinks between them. They published statistics of the first 500000 searches and revealed an average path length of approximately 3 and the longest path, running between Embleton and McCombie, with 11 degrees of separation[1].

Finally, the third property is called *heterogeneity* and refers to the importance of vertices in the network. There are many vertices with low degree (number of connections)

[1] `www.sixdegreesofwikipedia.com/blog/search-results-analysis`

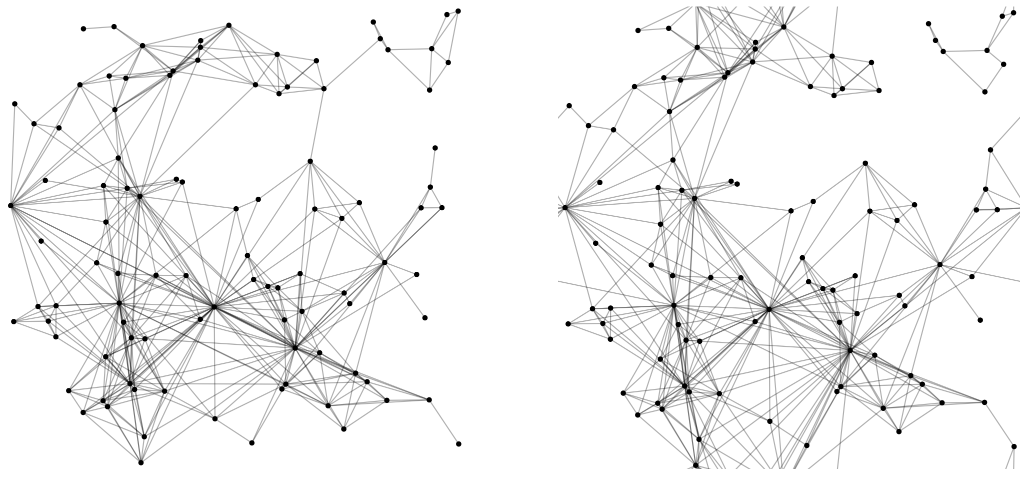

(a) heterogeneity (b) no border

Figure 1.2: Networks with 100 vertices and average degree eight. Both graphs have a heterogeneous degree distribution. The left has a square as ground space and the right graph uses a torus instead. Positions are the same as in Figure 1.1.

and few vertices with very high degree, often called hubs. This is typically described by a *scale-free* degree distribution, which means that the degree sequence follows a power law. Namely, the fraction of vertices with degree k is proportional to $k^{-\beta}$ for some constant β, which we call the *power-law exponent*. Heterogeneity can be observed in social networks, where celebrities have far more connections than the average person. In the Wikipedia network, the article about the United States is the most linked-to article with almost half a million direct references. Figure 1.2a adds a heterogeneous degree distribution to the graph in Figure 1.1b. It still has high locality and roughly the same average degree but the average path length goes down from 4.3 in Figure 1.1b to 3.1 in Figure 1.2a. Although this is a very small example, a similar statement can be made in general. In fact, it was shown that random scale-free networks already have the small-world property [CH03].

Locality, heterogeneity, and the small-world property were found in a great number of networks from different domains [Ama+00; EG17; WS98a]. Popular examples reach from the Internet's Autonomous System topology [AJB99; MMO14; BPK10], protein interaction networks [MS02a; Mil+07] and industrial SAT instances [ABL09; Ans+16; WGS03] over connections in the brain [BB06; Spo+04; Yu+08], metabolic networks [WF01] and social networks [Bar+02; Sco17; WF94] to the bitcoin transaction network [BFL14; LF16], ecological networks [MS02b] and many others [SBT04; Teo17; VSH04].

Knowing these properties, a large body of work was dedicated to formalize a generation mechanism that yields synthetic networks similar to real ones and to explain their emergence. A notable example is the *preferential attachment* model [BA99]. There, the network is grown by iteratively adding new nodes. As a new node joins, it gets connected to a fixed number of existing nodes that are chosen with probabilities proportional to their degree. This simple mechanism of preferential attachment [Pri76] already yields a scale-free degree distribution, thereby providing a convincing explanation for the emergence of such distributions in real-world networks. To mention another example, Watts and Strogatz [WS98b] proposed a network model based on a regular ring lattice to create very high locality. They observed that adding a few random long-range connections suffices to guarantee a small diameter. Since the model is not heterogeneous, it provides an alternative explanation for why many real-world networks exhibit the small-world property despite heavily favoring local over long-range connections.

While the preferential attachment model and the Watts-Strogatz model excel at the purpose they were designed for, i.e., explanatory power, they unfortunately are not very suitable for average-case analysis or algorithm engineering in general, because they produce highly artificial properties as a side product of the generation process. To investigate the implications of, for example, heterogeneity, a network model is required that assumes nothing beyond the heterogeneity itself. For this specific use case, the configuration model or the soft configuration model are best suited [GL04; PN04], because the former produces each graph with a fixed degree sequence with equal probability, while the latter relaxes the constraint to match the given degree sequence in expectation. Both these models have maximum entropy under the given constraints [HLK18].

Promising models that capture heterogeneity, locality as well as the small-world property are *geometric inhomogeneous random graphs* (GIRGs) and their specialization called *hyperbolic random graphs* (HRGs) [BKL19; Kri+10]. To incorporate heterogeneity, GIRGs are based on the Chung-Lu model [CL02b; CL02a], which is asymptotically equivalent to the soft configuration model [HLK18]. To produce locality, they assume an underlying geometry where vertices that are close have a higher probability of being connected. The small-world property arises naturally from the heterogeneous degree distribution despite the preference of local over global connections [BKL16]. Assuming geometry as a reasonable origin of locality, these models thus concisely capture all three properties mentioned above without introducing an unnecessary bias. Recent work supports the claim, that the validity of HRGs and GIRGs is sufficient to transfer insights — practical as well as theoretical — gained on synthetic instances to real-world applications [BF22].

Aside from their importance to network science, GIRGs can be an immensely valuable tool in algorithm engineering. Since they convincingly mimic real-world networks, guarantees about quality and performance of an algorithm on instances of the model can be transferred to real-world applications. They have model parameters to control the amount of heterogeneity and locality, which allows to evaluate those properties in isolation while keeping the rest fixed. Moreover, they can be efficiently generated, which allows for experimental analysis [Blä+19b]. While realistic instances are often rare, generated instances are readily available. Furthermore, the underlying geometry of GIRGs and HRGs helps to visualize the network, e.g., for debugging or to improve understanding of its structure. Finally, GIRGs use the torus as a ground space, that is, a hypercube where opposite sides are identified. Unlike a hypercube, the torus has no border and is therefore easy to handle mathematically. The difference between a two dimensional torus and a square as a ground space is visualized in Figure 1.2.

The aim of this work is to demonstrate the capabilities of geometric random graphs and hyperbolic random graphs in algorithm engineering and establish them as routine tools to replace previous models like the ER-model. To this end, we utilize them to design, evaluate, and optimize efficient algorithms for inputs with the above properties. Specifically, we provide the currently fastest sequential generators for GIRGs and HRGs and describe algorithms for maximum flow, directed spanning arborescence, cluster editing, and hitting set. For all four problems, our implementations beat the state-of-the-art on realistic inputs, that is, synthetic and real-world networks.

1.1 Contribution and Outline

The general contribution of this work is to establish GIRGs as a tool in algorithm engineering. Beyond our main goal, we develop and implement multiple algorithms for different problems that are, on their own, valuable contributions in their respective fields.

Chapter 2 deals with the efficient generation of GIRGs. To enable large scale benchmarks and analyses using GIRGs one has to draw instances of the model. Since real-world networks tend to be very large, a generator has to be efficient to match the size of these networks. The naive way to sample an instance takes quadratic time [AOK15], which quickly becomes infeasible for networks with millions of vertices. There exist multiple generators for the HRG model with subquadratic running time [LMP15; LM16; Loo+16; Blä+18b; Pen17; Fun+18; Fun+19; Loo19], but, although an efficient GIRG sampling algorithm is known in theory [BKL19], there is no implementation available. We provide the first efficient GIRG implementation by adapting the algorithm of Bringmann, Keusch, and Lengler [BKL19]. Our special-case implementation for HRGs beats all other HRG generators in a sequential setting while allowing a wider range of parameters than most existing implementations. Specifically, we support non-zero temperature T, which means our generator is able to vary the amount of locality and thereby provides more diverse and more challenging benchmark instances. Besides the generators themselves, we also provide an efficient algorithm to determine the non-trivial dependency between the average degree of the resulting graph and the input parameters of the GIRG model. This makes it possible to specify the desired expected average degree as input. Additionally, we investigate the relation between GIRGs and HRGs using our generators. Although HRGs represent, in a certain sense, a special case of the GIRG model, we find that a straight-forward inclusion does not hold in practice. However, the difference is negligible for most use cases.

In Chapter 3, we design and evaluate an efficient algorithm for solving the maximum flow problem on scale-free networks. The algorithm is based on the well-known algorithm by Dinitz [Din70]. Motivated by recent results, which show sublinear running time for bidirectional breadth-first search on Erdős-Rényi random graphs and hyperbolic random graphs [BN16; Blä+18a], we adapt the algorithm by Dinitz with a bidirectional breadth-first search. Our experiments on GIRGs indicate sublinear run time. On scale-free real-world networks, we outperform the commonly used highest-label Push-Relabel implementation [CG97] by up to two orders of magnitude. Compared to Dinitz's original algorithm, our modifications reduce the search space, e.g., by a factor of 275 on an autonomous systems graph. Since the network has to be read into memory and a residual network has to be constructed before computation, our sublinear running time on scale-free networks becomes relevant if multiple flow computations are performed on the same network. This is the case, for example, when computing a Gomory-Hu tree [GH61], which is a compact representation of the minimum s-t cuts for all vertex pairs. On a social network with 70 000 nodes, our algorithm computes the Gomory-Hu tree in 3 seconds compared to 12 minutes when using Push-Relabel. While our code is faster on scale-free networks, other solvers perform better on different kinds of networks.

Chapter 4 contains an experimental evaluation of the minimum spanning arborescence problem, which is the directed version of the well-studied minimum spanning tree problem. For a given root r, the goal is to find a directed spanning tree of minimum weight rooted at r. There is a general strategy for solving the problem [Edm67; Chu65; Boc71], which was refined into two algorithms. Tarjan's algorithm running in $O(\min(n^2, m \log n))$ [Tar77; CFM79] and the GGST algorithm running in $O(n \log n + m)$ [Gab+86]. Although different versions and generalizations of the problem were studied [Geo03; KKT09; Kam14], there exists no experimental evaluation of either Tarjan's algorithm or the GGST algorithm. In fact, the GGST algorithm has, to the best of our knowledge, never been implemented due to its intricate design and the presence of the much simpler version by Tarjan. However, the GGST algorithm theoretically beats Tarjan's algorithm by a log factor if the graph is neither too sparse nor too dense. We simplify and implement the GGST algorithm to perform a study on a wide range of real-world networks. We compare it against our own as well as publicly available Tarjan implementations. Unfortunately, the GGST algorithm shows a higher constant factor overhead and is outperformed by Tarjan implementations on real-world networks. Nevertheless, we use GIRGs to supply instances in the density regime where the GGST algorithm beats Tarjan's algorithm and indeed find a niche where it consistently outperforms all other solvers. Our solvers are faster than the publicly available solvers for most density ranges.

In Chapter 5 we discuss the hitting set problem. The hitting set problem asks for a collection of sets over a universe U to find a minimum subset of U that intersects each of the given sets. It is NP-hard and equivalent to the problem set cover. We give a branch-and-bound algorithm to solve hitting set. Though it requires exponential time in the worst case, it can solve many practical instances from different domains in reasonable time. Our algorithm outperforms a modern ILP solver, the state-of-the-art for hitting set, by at least an order of magnitude on most instances. We use GIRGs to evaluate the effect of locality and heterogeneity on our algorithm. To this end, we modify the GIRG generator to create bipartite graphs, which we interpret as hitting set instances. The results indicate that both properties drastically reduce the search space of our solver.

In Chapter 6, we describe and evaluate an exact branch-and-bound algorithm for the NP-hard cluster editing problem [KM86]. The cluster editing problem asks to transform a given graph into a disjoint union of cliques by inserting and deleting as few edges as possible. We introduce new reduction rules and adapt existing ones. Moreover, we generalize a known packing technique to obtain lower bounds and experimentally show that it contributes significantly to the performance of the solver. Our experiments further evaluate the effectiveness of the different reduction rules and examine the effects of locality and average degree of the input graph on solver performance using GIRGs. Our solver won the exact track of the 2021 PACE challenge [Kel+21].

We conclude our findings in Chapter 7.

1.2 Network Models

In this section, we introduce the most important network models that led up to the creation of geometric inhomogeneous random graphs. Erdős-Rényi random graphs are the oldest and simplest model. Random geometric graphs and Chung-Lu random graphs are the two models which combine into the GIRG model. The configuration model is strongly related to Chung-Lu graphs and hyperbolic random graphs can be seen as a special case of GIRGs.

1.2.1 Erdős-Rényi Random Graphs

The introduction of this random graph model forms the basis of modern network science. Instead of the $G(n, L)$ model by Erdős and Rényi [ER59], we describe the $G(n, p)$ model independently introduced by Gilbert [Gil59]. Both models are commonly referred to as the Erdős–Rényi Random Graph model. In the model, each vertex pair is connected with probability p. Therefore, all graphs on a fixed vertex set with a fixed number of edges are equally likely. The Erdős–Rényi model has an expected $pn(n-1)/2$ number of edges, an average degree of $p(n-1)$, and the degree sequence follows a binomial distribution. Unlike in real networks, in large random networks the degree of all nodes, the maximum degree, and the minimum degree are very close to the average degree. A major drawback of the ER model is that there is no locality or heterogeneity since all edges are equally likely. On the other hand, the diameter is logarithmic for sparse random graphs, which means the ER model has the small-world property.

1.2.2 (Soft) Configuration Model

The (soft) configuration model is a network model that is tailored towards providing maximum entropy under the constrains of a given degree sequence [GL04; Bia07; PN04; SG11; GL08]. The configuration model produces each graph with the given degree sequence with the same probability. In the soft version of the model, the given degree sequence is met in expectation. Furthermore, all networks with the same degree sequence have the same probability. The soft configuration model has the additional advantage that each network has a non-zero probability of appearing. An edge between vertices u and v exists with probability

$$p_{uv} = \frac{w_u w_v}{1 + w_u w_v},$$

where w_u and w_v are the weights of the two vertices which are proportional to their expected degree in the network. The model was used, for example, to approximate the topology of the World Trade Web, the network formed by the trade relationships between all world countries [GL04].

1.2.3 Chung-Lu Random Graphs

The Chung-Lu random graph model receives as input a weight sequence with vertex v having weight w_v and $W = \sum w_i$ being the sum of all weights [CL02a; CL02b]. Two vertices are connected by an edge independently at random with probability proportional to the product of their weights. In detail,

$$p_{uv} = \min(1, w_u w_v / W).$$

The expected degree of a vertex is proportional to its weight, thus the degree sequence follows the given weight sequence in expectation. The Erdős-Rényi model with connection probability p is a special case of this model if the weight of all vertices is set to pn.

1.2.4 Random Geometric Graphs

The random geometric graph model [Gil61; Pen03] distributes the vertices uniformly at random in the unit hypercube. Two vertices are then connected by an edge if their distance is below a certain threshold. The degree distribution in random geometric graphs is homogeneous. The clustering coefficient is non-vanishing with increasing graph size and only depends on the dimension of the ground space. However, the model does not have the small world property as the diameter is, due to the construction, at least R^{-1} where R is the connectivity threshold. Figure 1.1b shows an example of a random geometric graph in a two dimensional space, that is, vertices have positions in the unit square $[0,1]^2$.

1.2.5 Hyperbolic Random Graphs

Hyperbolic random graphs are generated by sampling random positions in the hyperbolic plane and connecting vertices that are close [Kri+10]. More formally, let $V = \{1, \dots, n\}$ be a set of vertices. Let $\alpha > 1/2$ and $C \in \mathbb{R}$ be two constants, where α controls the power-law degree distribution with exponent $\beta = 2\alpha{+}1 > 2$, and C determines the average degree. For each vertex $v \in V$, we sample a random point $p_v = (r_v, \theta_v)$ in the hyperbolic plane, using polar coordinates. Its angular coordinate θ_v is chosen uniformly from $[0, 2\pi)$ while its radius $0 \le r_v < R$ with $R = 2\log(n) + C$ is drawn according to the density function

$$f(r) = \frac{\alpha \sinh(\alpha r)}{\cosh(\alpha R) - 1}. \tag{1.1}$$

In the threshold case of HRGs two vertices $u \neq v$ are connected if and only if their distance is below R. The hyperbolic distance $d(p_u, p_v)$ is defined as

$$\cosh(d(p_u, p_v)) = \cosh(r_u)\cosh(r_v) - \sinh(r_u)\sinh(r_v)\cos(\theta_u - \theta_v), \tag{1.2}$$

where the angle difference $\theta_u - \theta_v$ is modulo π.

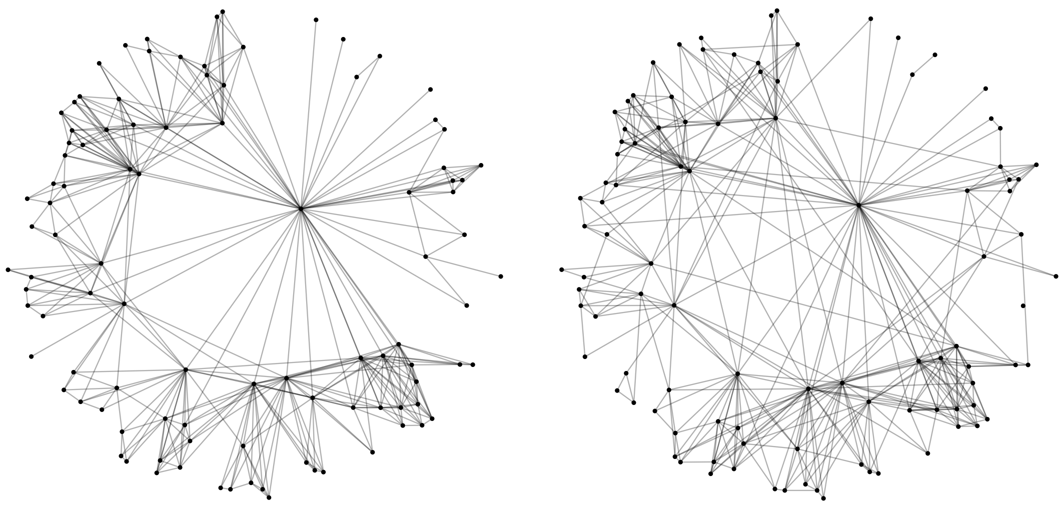

(a) $T = 0$ (b) $T = 0.3$

Figure 1.3: HRGs with 100 vertices, average degree eight, and different temperature.

The binomial variant adds a temperature $T \in [0, 1]$ to control the clustering, with lower temperatures leading to higher clustering as shown in Figure 1.3. Two nodes $u, v \in V$ are then connected with probability $p_T(d(p_u, p_v))$ where

$$p_T(x) = \frac{1}{e^{(x-R)/(2T)} + 1}. \tag{1.3}$$

For $T \to 0$, the two definitions (threshold and binomial) coincide.

1.2.6 Geometric Inhomogeneous Random Graphs

Geometric inhomogeneous random graphs [BKL19] combine elements from random geometric graphs [Gil61] and Chung-Lu graphs [CL02b; CL02a]. Each vertex v has a weight w_v and a position x_v. Vertices are connected independently based on their weight and distance. The model is defined as follows.

Let $V = \{1, \dots, n\}$ be a set of vertices with positive weights $w_1, \dots, w_n$ following a power law with exponent $\beta > 2$ and let W be their sum. Let $\mathbb{T}^d$ be the d-dimensional torus for a fixed dimension $d \geq 1$ represented by the d-dimensional cube $[0, 1]^d$ where opposite boundaries are identified. For each vertex $v \in V$, let $x_v \in \mathbb{T}^d$ be a point drawn uniformly and independently at random. For $x, y \in \mathbb{T}^d$ let $||x - y||$ denote the L_∞-norm on the torus, i.e. $||x - y|| = \max_{1 \leq i \leq d} \min\{|x_i - y_i|, 1 - |x_i - y_i|\}$. Two vertices $u \neq v$ are independently connected with probability p_{uv}. For a positive temperature $0 < T < 1$,

$$p_{uv} = \min\left\{1, c\left(\frac{w_u w_v / W}{||x_u - x_v||^d}\right)^{1/T}\right\} \tag{1.4}$$

while for $T = 0$ a threshold variant of the model is obtained with

$$p_{uv} = \begin{cases} 1 & \text{if } ||x_u - x_v|| \leq c(w_u w_v / W)^{1/d}, \\ 0 & \text{else.} \end{cases}$$

The constant $c > 0$ controls the expected average degree.

Bringmann, Keusch, and Lengler [BKL19] show that the HRG model can be seen as a special case of the GIRG model in the following sense. Let d_{HRG} be the average degree of a HRG. Then there exist GIRGs with average degree d_{GIRG} and D_{GIRG} with $d_{\mathrm{GIRG}} \leq d_{\mathrm{HRG}} \leq D_{\mathrm{GIRG}}$ such that they are sub- and supergraphs of the HRG, respectively. Moreover, d_{GIRG} and D_{GIRG} differ only by a constant factor. Formally, this is achieved by using the big-O notation instead of a single constant c for the connection probability. In that sense, the above formulation of GIRGs deviates from the original definition, which we call the *generic GIRG framework*. It basically captures any specific model whose connection probabilities differ from Equation (1.4) by only a constant factor. From a theoretical point of view this is useful as proving something for the generic GIRG framework also proves it for any manifestation, including HRGs.

To see how HRGs fit into the generic GIRG framework, consider the following mapping [BKL19]. Radii are mapped to weights $w_v = e^{(R - r_v)/2}$, and angles are scaled to fit on a 1-dimensional torus $x_v = \theta_v / (2\pi)$. One can then see that the hyperbolic connection probability $p_T(d)$ under the provided mapping deviates from Equation (1.4) by only a constant. Thus, c in Equation (1.4) can be chosen such that all GIRG probabilities are larger or smaller than the corresponding HRG probabilities, leading to the two average degrees d_{GIRG} and D_{GIRG} mentioned above. Bringmann, Keusch, and Lengler [BKL19] note that the two constants, which they hide in the big-O notation, do not have to match. They leave it open if they match, converge asymptotically, or how large the interval between them is in practice. We investigate this empirically in Section 2.4.3.

2 Generating Geometric Inhomogeneous Random Graphs

This chapter is based on joint work with Thomas Bläsius, Tobias Friedrich, Maximilian Katzmann, Ulrich Meyer and Manuel Penschuck [Blä+19b; Blä+22b]. The paper won the best paper award at ESA 2019. It was preceeded by a master thesis by Marianne Thieffry.

2.1 Introduction

Network models play an important role in different scientific fields [CF06]. From the perspective of network science, models can be used to explain observed behavior in the real world. From the perspective of computer science, and specifically algorithmics, realistic random networks can provide input instances for graph algorithms. This facilitates theoretical approaches (e.g., average-case analysis), as well as extensive empirical evaluations by providing an abundance of benchmark instances, solving the pervasive scarcity of real-world instances.

There are some crucial features that make a network model useful. The generated instances have to resemble real-world networks. The model should be as simple and natural as possible to facilitate theoretical analysis, and to prevent untypical artifacts. And it should be possible to efficiently draw networks from the model. This is particularly important for the empirical analysis of model properties and for generating benchmark instances.

A model that has proven itself useful in recent years is the *hyperbolic random graph (HRG)* model [Kri+10] we described in Section 1.2.5. HRGs are generated by drawing vertex positions uniformly at random from a disk in the hyperbolic plane. Two vertices are joined by an edge if and only if their distance lies below a certain threshold. HRGs resemble real-world networks with respect to crucial properties. Most notable are the *power-law degree distribution* [GPP12] (i.e., the number of vertices of degree k is roughly proportional to $k^{-\beta}$ with $\beta \in (2, 3)$), the high *clustering coefficient* [GPP12] (i.e., two vertices are more likely to be connected if they have a common neighbor), and the small diameter [FK18; MS17]. Moreover, HRGs are accessible for theoretical analysis (see, e.g., [GPP12; FK18; MS17; Blä+18a]). Finally there is a multitude of efficient generators with different emphases [AOK15; LMP15; LM16; Loo+16; Pen17; Fun+18; Fun+19]; see Section 2.1.2 for a discussion.

As mentioned in Section 1.2, HRGs are generalized by the *geometric inhomogeneous random graph (GIRG)* model [BKL19]. Here every vertex has a position on the d-dimensional torus and a weight following a power law. Two vertices are then connected if and only if their distance on the torus is smaller than a threshold based on the product of their weights. When using positions on the circle ($d = 1$), GIRGs approximate HRGs in the following sense: the processes of generating a HRG and a GIRG can be coupled such that it suffices to decrease and increase the average degree of the GIRG by only a constant factor to obtain a subgraph and a supergraph of the corresponding HRG, respectively. Compared to HRGs, GIRGs are potentially easier to analyze, generalize nicely to higher dimensions, and the weights allow to directly adjust the degree distribution.

Above, we described the idealized *threshold variants* of the models, where two vertices are connected if an only if their distance is small enough. Arguably more realistic are the *binomial variants*, which allow longer edges and shorter non-edges with a small probability. This is achieved with an additional parameter T, called *temperature*. For $T \to 0$, the binomial and threshold variants coincide. Many publications focus on the threshold case, as it is typically simpler. This is particularly true for generation algorithms: in the threshold variants, one can ignore all vertex pairs with sufficient distance, which can be done using geometric data structures. In the binomial case, any pair of vertices could be adjacent, and the search space cannot be reduced as easily. For practical purposes, however, a non-zero temperature is crucial as real-world networks are generally assumed to have positive temperature allowing so called *weak ties* [Gra73]. That is, edges between nodes that have no strong reason to be connected and where the endpoints don't have many common neighbours. Moreover, from an algorithmic perspective, the threshold variants typically produce particularly well-behaved instances, while a higher temperature leads to more difficult problem inputs. Thus, to obtain benchmark instances of varying difficulty, generators for the binomial variants are key.

2.1.1 Contribution & Outline

Based on the algorithm by Bringmann, Keusch, and Lengler [BKL19], we provide an efficient and flexible GIRG generator. It includes the binomial case and allows higher dimensions. Its expected running time is linear in the graph size. To the best of our knowledge, this is the first efficient generator for the GIRG model. Moreover, we adapt the algorithm to the HRG model, including the binomial variant. Compared to existing HRG generators (most of which only support the threshold variant), our implementation is the fastest sequential HRG generator.

A refactoring of the original GIRG algorithm [BKL19] allows us to parallelize our generators. They do not use multiple processors as effectively as the threshold-HRG generator by Penschuck [Pen17], which was specifically tailored towards parallelism. However, in a setting realistic for commodity hardware (8 cores, 16 threads), we still achieve comparable run times.

Our generators come as an open-source C++ library[1] with documentation, command-line interface, unit tests, micro benchmarks, and OpenMP [Boa18] parallelization using shared memory.

Besides the efficient generators, we have three secondary contributions. (I) We provide

[1] `https://github.com/chistopher/girgs`

Table 2.1: Existing hyperbolic random graph generators. The columns show the names used throughout the paper; the conference appearance; a reference (journal if available); whether the generator supports the binomial model; and the asymptotic running time. The time bounds hold in the worst case (wc), with high probability (whp), in expectation (exp), or empirically (emp).

Name	First Published	Ref.	Bin.	Running Time
Pairwise	CPC'15	[AOK15]	✓	$\Theta(n^2)$ (wc)
QuadTree	ISAAC'15	[LMP15]		$O((n^{3/2}+m)\log n)$ (wc)
NkQuad	IWOCA'16	[LM16]	✓	$O((n^{3/2}+m)\log n)$ (wc)
NkGen, NkOpt	HPEC'16	[Loo+16]		$O(n\log n+m)$ (emp)
Embedder	ESA'16	[Blä+18b]	✓	$\Theta(n+m)$ (exp)
HyperGen	SEA'17	[Pen17]		$O(n\log\log n+m)$ (whp)
RHG	IPDPS'18	[Fun+18]		$\Theta(n+m)$ (exp)
sRHG	JPDC'19	[Fun+19]		$\Theta(n+m)$ (exp)
NkGenBin	KIT'19	[Loo19]	✓	$O(n\log^2 n+m)$ (exp)
HyperGIRGs (ours)	ESA'19	[Blä+22b]	✓	$\Theta(n+m)$ (exp)

a comprehensible description of the sampling algorithm that should make it easy to understand how the algorithm works, why it works, and how it can be implemented. Although the core idea of the algorithm is not new [BKL19], the previous description is somewhat technical. (II) The expected average degree can be controlled via an input parameter. However, the dependence of the average degree on the actual parameter is non-trivial. In fact, given the average degree, there is no closed formula to determine the parameter. We provide a linear-time algorithm to estimate it. (III) We investigate how GIRGs and HRGs actually relate to each other by measuring how much the average degree of the GIRG has to be decreased and increased to obtain a subgraph and supergraph of the HRG, respectively. We find that a GIRG with only slightly lower average degree already yields a subgraph. In fact, our experiments indicate that the gap between average degrees vanishes for growing n, i.e., the GIRG subgraph is lacking only a sublinear fraction of edges. On the other hand, one has to increase the average degree significantly to obtain a GIRG supergraph.

In the following we first discuss our main contribution in the context of existing HRG generators. Afterwards we describe the sampling algorithm in Section 2.2. In Section 2.3 we discuss implementation details, including the parameter estimation for the average degree (Section 2.3.3) as well as multiple performance improvements. Section 2.4 contains our experiments: we investigate the scaling behavior of our generator in Section 2.4.1, compare our HRG generator to existing ones in Section 2.4.2, and compare GIRGs to HRGs in Section 2.4.3.

2.1.2 Comparison with Existing Generators

Concerning HRGs, most previous algorithms only support the threshold case; see Table 2.1. A quad-tree data structure was used to achieve the first subquadratic threshold generator (QuadTree) [LMP15]. It was later improved leading to the algorithm currently implemented in NetworKit (NkGen) [Loo+16]. A later re-implementation by Penschuck [Pen17] improves it by about a factor of 2 (NkOpt). However, the main contribution of Penschuck [Pen17] was a new generator that features sublinear memory and near optimal parallelization (HyperGen). Up to date, HyperGen was the fastest threshold-HRG generator on a single processor. Our generator, HyperGIRGs, improves by a factor of 1.3 – 2 (depending on the parameters) but scales worse for more processors. Finally, Funke et al. [Fun+18] provide a generator designed for a distributed setting (RHG) and later combine it with the streaming technique of HyperGen to generate enormous instances (sRHG) [Fun+19].

The published generators for the binomial model are the trivial quadratic algorithm [AOK15], and an $O((n^{3/2} + m) \log n)$ algorithm [LM16] based on the above mentioned quad-tree data structure [LMP15]. The latter is part of NetworkKit; we call it NkQuad. In his thesis, v. Looz adapted NkGen for the binomial model resulting in the NkGenBin algorithm [Loo19]. Moreover, the code for a hyperbolic embedding algorithm [Blä+18b] includes an HRG generator implemented by Bringmann based on the GIRG algorithm [BKL19]; we call it Embedder in the following. Embedder has been widely ignored as a high performance generator. This is because it was somewhat hidden, and it is heavily outperformed by other threshold generators. Experiments show that our generator HyperGIRGs is much faster than NkQuad, which is to be expected considering the asymptotic running time. Moreover, on a single processor, we outperform Embedder by an order of magnitude for $T = 0$ and by a factor of 4 for higher temperatures. As Embedder does not support parallelization, this speed-up increases for multiple processors. Finally, we are two to three times faster than NkGenBin, which was shown to perform slightly better than Embedder for $T > 0$ [Loo19].

We are not aware of a previous GIRG generator.

2.2 Sampling Algorithm

As mentioned in the introduction, the core of our sampling algorithm is based on the algorithm by Bringmann et al. [BKL19]. In the following, we first give a description of the core ideas and then work out the details that lead to an efficient implementation.

To explain the idea, we make two temporary assumptions and relax them in Section 2.2.1 and Section 2.2.2, respectively. For now, assume that all weights are equal and consider only the threshold variant $T = 0$. The task is to find all vertex pairs that form an edge, i.e., their distance is below the threshold $c(w_u w_v/W)^{1/d}$. Since all weights are equal, the threshold in this restricted scenario is the same for all vertex pairs. One approach to quickly identify adjacent vertices is to partition the ground space into a grid of cells. The size of the cells should be chosen, such that (I) the cells are as small as possible and

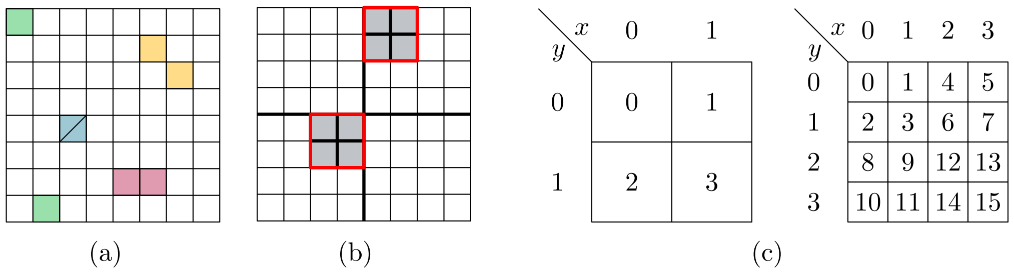


Figure 2.1: **(a),(b)** The grid used by weight bucket pairs with a connection probability threshold between 2^{-3} and 2^{-4} in two dimensions. **(a)** Each pair of colored cells represent neighbors. Note that the ground space is a torus and a cell is also a neighbor to itself. **(b)** The eight gray cells represent multiple distant cell pairs, which are replaced by one pair consisting of the red outlined parent cell pair. **(c)** Linearization of the cells on level 1 (left) and 2 (right) for $d = 2$.

(II) the diameter of cells is larger than the threshold $c(w_u w_v/W)^{1/d}$. The latter implies that only vertices in neighboring cells can be connected thus narrowing down the search space. The former ensures that neighboring cells contain as few vertex pairs as possible reducing the number of comparisons. Figure 2.1a shows an example of such a grid for a 2-dimensional ground space.

2.2.1 Inhomogeneous Weights

Assume that we have vertices with two different weights w_1, w_2, rather than one. As before, the cells should still be as small as possible while having a diameter larger than the connection threshold. However, there are three different thresholds now, one for each combination of weights. To resolve this, we can group the vertices by weight and use three differently sized grids to find the edges between them.

As GIRGs require not only two but many weights, considering one grid for every weight pair is infeasible. The solution is to discretize the weights by grouping ranges of weights into *weight buckets*. When searching for edges between vertices in two weight buckets, the pair of largest weights in these buckets provides the threshold for the cell diameter. This choice of the cell diameter satisfies property (II). Property (I) is violated only slightly, if the weight range within the bucket is not too large. Thus, each combination of two weight buckets uses a grid of cells, whose granularity is based on the maximum weight in the respective buckets.

As a tradeoff, we choose $\lceil \log_2 n \rceil$ many buckets which yields a sublinear number of grids. Moreover, the largest and smallest weight in a bucket are at most a factor two apart. Thus, the diameter of a cell is too large by at most a factor of four.

With this approach, a single vertex has to appear in grids of different granularity. To do this in an efficient manner, we recursively divide the space into ever smaller grid cells, leading to a hierarchical subdivision of the space. This hierarchy is naturally described by a tree. For a 2-dimensional ground space, each node has four children, which is why we call it *quadtree*. Note that each level of the quadtree represents a grid of different granularity. Moreover, the side length of a grid cell on level ℓ is $2^{-\ell}$. For a pair (i, j) of weight buckets, we then choose the level that fits best for the corresponding weights, i.e., the deepest level such that the diameter of each grid cell is above the connection threshold for the largest weights in bucket i and j, respectively. We call this level the *comparison level*, denoted by $CL(i, j)$. It suffices to insert vertices of a bucket into the deepest level among all its comparison levels. This level is called the *insertion level* and we denote it by $I(i)$. In Section 2.2.4, we discuss in detail how to efficiently access all vertices in a given grid cell belonging to a given weight bucket.

2.2.2 Binomial Variant of the Model

For $T > 0$, neighboring cell pairs are still easy to handle: a constant fraction of vertex pairs will have an edge and one can sample them by explicitly checking every pair. For distant cell pairs and a fixed pair of weight buckets, the distance between the cells yields an upper bound on the connection probability of included vertices; see Equation (1.4). The probability bound depends on both, the weight buckets and the cell pair distance, using the maximum weight within the buckets and the minimum distance between points in the cells. We note that the individual connection probabilities are only a constant factor smaller than the upper bound.

Knowing this, we can use geometric jumps to skip most vertex pairs [AD85]. The approach works as follows. Assume that we want to create an edge with probability $\overline{p}$ for each vertex pair. For this process, we define the random variable X to be the number of vertex pairs we see until we add the next edge. Then X follows a geometric distribution. Thus, instead of throwing a coin for each vertex pair, we can do a single experiment that samples X from the geometric distribution and then skip X vertex pairs ahead. Since not all vertex pairs reach the upper bound $\overline{p}$, we accept encountered pairs with probability $p_{uv}/\overline{p}$ to get correct results.

Although distant cell pairs are handled efficiently, their number is still quadratic, most of which yield no edges. To circumvent this problem, the sampling algorithm, yet again, uses a quadtree. In the quadratic set of cell pairs to compare for one weight bucket pair, non-neighboring cells are grouped together along the quadtree hierarchy. They are replaced by their parents as shown in Figure 2.1b until their parents become neighbors.

In conclusion, for each pair of weight buckets (i, j) the following two types of cell pairs have to be processed: any two neighboring cell pairs on the comparison level $CL(i, j)$; and any distant cell pair with level larger or equal $CL(i, j)$ that has neighboring parents. The resulting set of distant and neighboring cell pairs for a fixed bucket pair partitions $\mathbb{T}^d \times \mathbb{T}^d$.

2.2.3 Efficiently Iterating Over Cell Pairs

The previous description sketches the algorithm as originally published. Here, we propose a refactoring that greatly simplifies the implementation and enables parallelization. We attribute a significant amount of HyperGIRGs' speed up over Embedder to this change.

Instead of first iterating over all bucket pairs and then over all corresponding cell pairs, we reverse this order. This removes the need to repeatedly determine the cell pairs to process for a given bucket pair. Instead it suffices to find the bucket pairs that process a given cell pair. This only depends on the level of the two cells and their type (neighboring or distant). Inverting the mapping from bucket pairs to cell pairs in the previous section yields the following. A neighboring cell pair on level ℓ is processed for bucket pairs with a comparison level of exactly ℓ. A distant cell pair on level ℓ (with neighboring parents) is processed for bucket pairs with a comparison level larger than or equal to ℓ. Thus, for each level of the quadtree we must enumerate all neighboring cell pairs, as well as distant cell pairs with neighboring parents. Algorithm 2.1 recursively enumerates exactly these cell pairs.

Algorithm 2.1: Sample GIRG by Recursive Iteration of Cell Pairs

```
   Input: cell pair (A,B); initially called with A,B set to the root of the quadtree
1  forall bucket pairs $(i, j)$ that process the cell pair $(A, B)$ do
2      if $A$ and $B$ are neighbors then
3          emit each edge $(u, v) \in V_i^A \times V_j^B$ with probability $p_{uv}$
4      else
5          choose candidates $S \subseteq V_i^A \times V_j^B$ using geometric jumps and $\overline{p}$
6          emit each edge $(u, v) \in S$ with probability $p_{uv}/\overline{p}$
7  if $A$ and $B$ are neighbors and not maximum depth reached then
8      forall children $X$ of $A$ do
9          forall children $Y$ of $B$ do
10             recur($X, Y$)
```

2.2.4 Efficient Access to Vertices by Bucket and Cell

A crucial part of the algorithm is to quickly access the set of vertices restricted to a weight bucket i and a cell A, which we denote by V_i^A. To this end, we linearize the cells of each level as illustrated in Figure 2.1c. This linearization is called Morton code [Mor66] or z-order curve [OM84]. It has the nice properties that (I) for each cell in level ℓ, its descendants in level $\ell' > \ell$ in the quadtree appear consecutively; and (II) it is easy to convert between a cells position in the linear order and its d-dimensional coordinates (see Section 2.3.2).

We sort the vertices of a fixed weight bucket i by the Morton code of their containing cell on the insertion level $I(i)$, using arbitrary tie-breaking for vertices in the same cell. This has the effect that for any cell A with level$(A) \leq I(i)$, the vertices of V_i^A appear consecutive. Thus, to efficiently enumerate them, it suffices to know for each cell A the index of the first vertex in V_i^A. This can be precomputed using prefix sums leading to the following lemma.

Lemma 1. *After linear preprocessing, for all cells A and weight buckets i with* $\text{level}(A) \leq I(i)$*, vertices in the set V_i^A can be enumerated in $\mathcal{O}(|V_i^A|)$.*

Proof. As mentioned above, we have to sort the vertices V_i of each weight bucket i according to the index (Morton code) of the containing cell. Clearly, the d-dimensional coordinates of the cell containing a given vertex is obtained in constant time by rounding. From this one can obtain the index in constant time (also see Section 2.3.2). This can be done using, e.g., bucket sort with respect to this index to sort the vertices. In the following, we refer to this sorted array with V_i.

Besides these sorted arrays V_i of vertices, one for each weight bucket i, we store for each cell C at level $I(i)$ the number of vertices preceding the vertices in cell C. Note that this is simply the prefix sum of the number of vertices in all cells that come before cell C. Denote this prefix sum of cell C with P_C.

Now let i be a weight bucket and let A be a cell identifying the requested set of vertices V_i^A (with $\text{level}(A) \leq I(i)$). Let $C_1, \dots, C_j$ be the descendants of cell A at level $I(i)$, appearing in this order according to the Morton code. Recall that the vertices in $C_1, \dots, C_j$ appear consecutive in the sorted array V_i. Thus, V_i^A is given by the range $[P_{C_1}, \dots, P_{C_{j+1}})$ in V_i.

In terms of running time, each weight bucket requires $\mathcal{O}(|V_i| + 2^{d \cdot I(i)})$ time for bucket sort and $\mathcal{O}(2^{d \cdot I(i)})$ time for the prefix sums, where $2^{d \cdot I(i)}$ is the number of cells in the insertion level $I(i)$. Over all weight buckets, the term $|V_i|$ sums up to $|V|$ and Bringmann et al. [BKL19] show that the same holds for $2^{d \cdot I(i)}$. $\square$

2.2.5 Adapting the Algorithm to HRGs

One possibility to generate HRGs with this algorithm would be to convert hyperbolic points to GIRG coordinates according to Section 1.2.6 and use the algorithm as is. However, the generic GIRG framework captures HRGs only up to constant factor deviations in connection probabilities. In fact, we find that using only one scaling constant as in Equation (1.4) is insufficient to exactly represent the corresponding HRG probabilities (see Section 2.4.3).

To generate exact HRGs, we adapt the algorithm to work with hyperbolic data in the first place, as was done in [Blä+18b]. Concretely, we sample and store only hyperbolic coordinates instead of mapping them to GIRG data, use the hyperbolic distance function to determine distance between vertices and cells, control the expected average degree with an estimate for the radius R of the hyperbolic disc, use the exact hyperbolic connection probability p_T, and trivially find the comparison- and insertion levels instead of using the closed form for canonical GIRGs. The resulting implementation is called HyperGIRGs.

2.3 Implementation Details

The description in the previous section is an idealized version of the algorithm. For an actual implementation, there are some gaps to fill in. Omitting many minor tweaks, we want to mention implementation details and optimizations that are crucial to achieve a good practical run time in the following.

2.3.1 Avoiding Double Counting Buckets, Cells, and Vertices

The algorithm as described in Section 2.2 iterates over pairs of buckets, cells, and vertices. All three entities need to be handled correctly to avoid visiting vertex or cell pairs multiple times. Consider the cell pairs (A, B) and (B, A) as well as two bucket pairs (i, j) and (j, i) that process them. When the bucket pair (i, j) processes (A, B) it samples edges between V_i^A and V_j^B while the bucket pair (j, i) processes (B, A) to sample edges between V_j^B and V_i^A. Meaning these edges are sampled twice; once in each direction. Since we want undirected edges this introduces double counting. To solve this, one can restrict the algorithm to cell pairs $A \leq B$ (or to bucket pairs $i \leq j$). In any case, bucket pairs (i, i) require special treatment for cell pairs of the form (A, A). Then, only edges between vertices $u < v$ should be checked, because this call samples edges within a set of vertices instead of between two disjoined vertex sets. If self loops are desired, the constraint can be relaxed to $u \leq v$.

2.3.2 Efficiently Encoding and Decoding Morton Codes

Recall from Section 2.2.4 that we linearize the d-dimensional grid of cells using Morton code. As vertex positions are given as d-dimensional coordinates, we have to convert the coordinates to Morton codes (i.e., the index in the linearization) and vice versa. This is done by bitwise interleaving of the coordinates. For example, the 2-dimensional Morton code of the four-bit coordinates $a = a_3a_2a_1a_0$ and $b = b_3b_2b_1b_0$ is $a_3b_3a_2b_2a_1b_1a_0b_0$.

Implementation-wise, there are the following encoding approaches.

FOR, FOR OPT Set each bit of the result with shifts and bitwise operations (FOR). Since we know the level of a cell, we know the number of relevant bits in each coordinate. Considering only relevant bits improves performance significantly (FOR OPT).

MASKS For details on this method, we refer to the open-source library *libmorton* [Bae18] and the authors related blog posts[2].

LUT A lookup table computed at compile time[3] can be used. The input is divided into chunks; a precomputed result for each chunk is obtained and shifted into place.

BMI2 The *Parallel Bits Deposit/Extract* assembler instructions from Intels Bit Manipulation Instruction Set 2 [Int19] provide a solution with one assembler instruction per input coordinate. BMI2 is available on Intel CPUs since 2013 and supported by recent AMD CPUs (Zen).

[2]`https://www.forceflow.be`
[3]`https://github.com/kevinhartman/morton-nd`

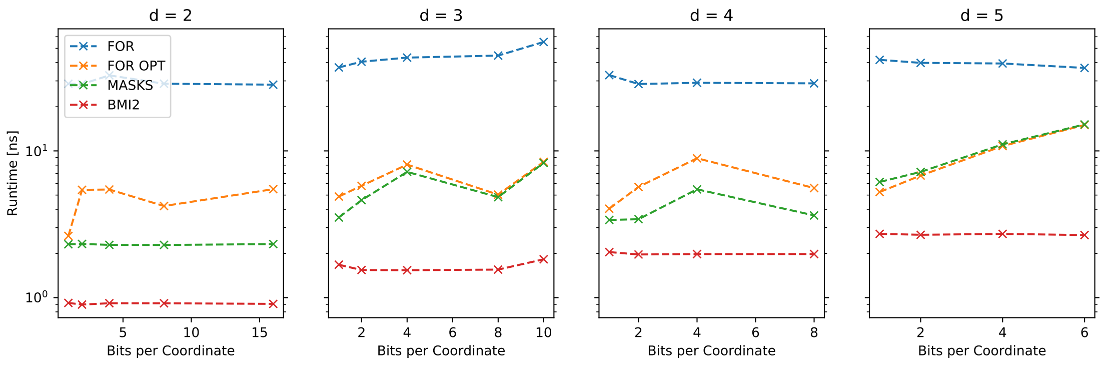


Figure 2.2: Performance of Morton code generation in dimensions 2 to 5 on an Intel processor. Input coordinates are limited to $\lfloor 32/d \rfloor$ bits each, because the result is saved as a 32 bit integer.

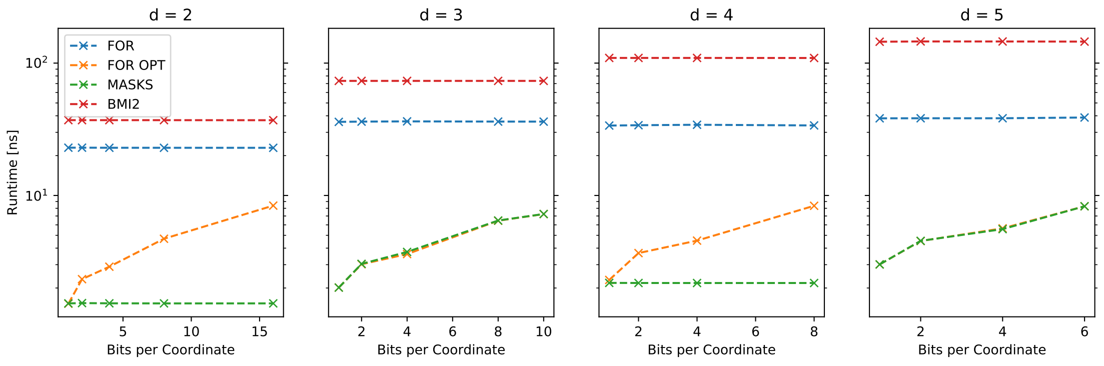


Figure 2.3: Performance of Morton code generation in dimensions 2 to 5 on an AMD processor. Input coordinates are limited to $\lfloor 32/d \rfloor$ bits each, because the result is saved as a 32 bit integer.

All approaches except LUT support a complementary decoding operation. We measured the approaches, excluding LUT, on an Intel i7-8550U processor (see Figure 2.2) and an AMD Ryzen7-2700X (see Figure 2.3). On Intel, BMI2 is consistently the fastest and at least an order of magnitude faster than FOR. Surprisingly, FOR OPT is not monotone in the number of bits per coordinate for dimensions below 5. Inspection of the generated assembly[4] reveals that the compiler employed SIMD instructions. On AMD, BMI2 is the slowest. Our GIRG generator uses BMI2 if enabled and the loop with early termination (FOR OPT) otherwise.

2.3.3 Estimating the Average Degree Parameter

Here, we describe how to estimate the parameter c in Eq. (1.4) to achieve a given expected average degree[5]. This section covers the estimation for the binomial version of the model $T > 0$. The calculations for the threshold case $T = 0$ are analogous (and simpler). We estimate the constant based on the actual weights, not on their probability distribution. This leads to lower variance and allows user-defined weights.

We start with an arbitrary constant c, calculate the resulting expected average degree $\mathbb{E}[\bar{d}]$ and adjust c accordingly, using a modified binary search. This is possible, as $\mathbb{E}[\bar{d}]$ is monotone in c. We derive an exact formula for $\mathbb{E}[\bar{d}]$, depending on c and the weights. It cannot simply be solved for c, which is why we use binary search instead of a closed expression.

For the binary search, we need to efficiently evaluate $\mathbb{E}[\bar{d}]$ for different values of c. Let X_{uv} be a random indicator variable for the existence of the edge uv with fixed weights but unknown positions. The expected average degree can be expressed as

$$\mathbb{E}[\bar{d}] = \frac{1}{n} \cdot \mathbb{E}\left[\sum_{u \in V} \sum_{v \neq u} X_{uv}\right] = \frac{1}{n} \sum_{u \in V} \sum_{v \neq u} \mathbb{E}[X_{uv}] \tag{2.1}$$

with the expectation of a single edge being

$$\mathbb{E}[X_{uv}] = \mathbb{E}\left[\min\left\{1, c \cdot \left(\frac{w_u w_v / W}{||x_u - x_v||^d}\right)^{1/T}\right\}\right] = \mathbb{E}\left[\min\left\{1, \left(\frac{c^{\frac{T}{d}} \left(\frac{w_u w_v}{W}\right)^{\frac{1}{d}}}{||x_u - x_v||}\right)^{d/T}\right\}\right].$$

This is potentially problematic, as the formula for $\mathbb{E}[\bar{d}]$ sums over all vertex pairs. The issue preventing us from simplifying this formula is the minimum in the connection probability. We split it into *short edges* and *long edges* based on whether the minimum takes effect or not, that is, whether the numerator of the connection probability, call it $k = c^{\frac{T}{d}} \left(\frac{w_u w_v}{W}\right)^{1/d}$, is bigger than the distance in the denominator. If $||x_u - x_v|| \leq k$ we have a short edge and the vertices are so close together that they will definitely be

[4] g++8 -std=c++14 -O3 -march=skylake

[5] Actually, we implement GIRGs without explicitly modeling the parameter c because scaling all weights by c^T emulates the same behaviour.

connected. Else, we have a long edge with $k < ||x_u - x_v||$, thus we can drop the minimum. We get

$$\mathbb{E}[X_{uv}] = \Pr(||x_u - x_v|| \leq k) + \Pr(k < ||x_u - x_v||) \cdot \mathbb{E}\left[c \cdot \left(\frac{w_u w_v / W}{||x_u - x_v||^d} \right)^{1/T} \mid k < ||x_u - x_v|| \right].$$

For any constant $t \leq 0.5$, $\Pr(||x_u - x_v|| \leq t) = (2t)^d$, which is the fraction of the ground space which is covered by a hypercube with radius t. The probability for a short edge becomes

$$\Pr(||x_u - x_v|| \leq k) = \begin{cases} (2k)^d = 2^d c^T \left(\frac{w_u w_v}{W}\right) & \text{if } k \leq 0.5 \\ 1 & \text{else} \end{cases} \tag{2.2}$$

For long edges, one must also distinguish between $k > 0.5$ and $k \leq 0.5$ because the distance on a unit torus with the L_∞-norm is at most 0.5. Thus for $k > 0.5$, the probability for a long edge $\Pr(k < ||x_u - x_v||)$ becomes zero independent of the distance. For $k \leq 0.5$, we can simplify the formula for long edges by integrating over all possible values of $||x_u - x_v||$. The probability density function of $||x_u - x_v||$ between 0 and 0.5 is the derivative of $(2x)^d$, namely $d2^d x^{d-1}$. Using this we get

$$\begin{aligned}
&\Pr(k < ||x_u - x_v||) \cdot \mathbb{E}\left[c \cdot \left(\frac{w_u w_v / W}{||x_u - x_v||^d} \right)^{1/T} \mid k < ||x_u - x_v|| \right] \\
&= \Pr(k < ||x_u - x_v||) \cdot \frac{\int_k^{0.5} c \cdot \left(\frac{w_u w_v / W}{x^d}\right)^{1/T} \cdot d2^d x^{d-1} \,\mathrm{d}x}{\Pr(k < ||x_u - x_v||)} \\
&= c \left(\frac{w_u w_v}{W}\right)^{1/T} d2^d \int_k^{0.5} x^{d-1-d/T} \,\mathrm{d}x \\
&= c \left(\frac{w_u w_v}{W}\right)^{1/T} d2^d \left[\frac{1}{d(1-1/T)} \cdot x^{d-d/T} \right]_k^{0.5} \\
&= c \left(\frac{w_u w_v}{W}\right)^{1/T} \frac{d2^d}{d(1-1/T)} \left(\left(\frac{1}{2}\right)^{d-d/T} - k^{d(1-1/T)} \right) \\
&= c \left(\frac{w_u w_v}{W}\right)^{1/T} \frac{2^d}{1-1/T} \left(\frac{2^{d/T}}{2^d} - \left(c^{\frac{T}{d}} \left(\frac{w_u w_v}{W}\right)^{1/d} \right)^{d(1-1/T)} \right) \\
&= c \left(\frac{w_u w_v}{W}\right)^{1/T} \frac{2^{d/T}}{1-1/T} - c \left(\frac{w_u w_v}{W}\right)^{1/T} \frac{2^d}{1-1/T} c^{T-1} \left(\frac{w_u w_v}{W}\right)^{1-1/T} \\
&= c \left(\frac{w_u w_v}{W}\right)^{1/T} \frac{2^{d/T}}{1-1/T} - c^T \left(\frac{w_u w_v}{W}\right) \frac{2^d}{1-1/T}.
\end{aligned} \tag{2.3}$$

We add the $k \leq 0.5$ cases of short and long edges (Eq. 2.2 and Eq. 2.3), that is

$$\begin{aligned}
&2^d c^T \left(\frac{w_u w_v}{W}\right) + c\left(\frac{w_u w_v}{W}\right)^{1/T} \frac{2^{d/T}}{1-1/T} - c^T \left(\frac{w_u w_v}{W}\right) \frac{2^d}{1-1/T} \\
&= 2^d c^T \left(\frac{w_u w_v}{W}\right)\left(1 + \frac{1}{1/T-1}\right) - c\left(\frac{w_u w_v}{W}\right)^{1/T} \frac{2^{d/T}}{1/T-1} \\
&= c^T \frac{2^d}{1-T}\left(\frac{w_u w_v}{W}\right) - c\frac{2^{d/T}}{1/T-1}\left(\frac{w_u w_v}{W}\right)^{1/T},
\end{aligned} \tag{2.4}$$

to concisely express the expectation for X_{uv} as

$$\mathbb{E}[X_{uv}] = \begin{cases} c^T \frac{2^d}{1-T}\left(\frac{w_u w_v}{W}\right) - c\frac{2^{d/T}}{1/T-1}\left(\frac{w_u w_v}{W}\right)^{1/T} & \text{if } k \leq 0.5 \\ 1 & \text{if } k > 0.5 \end{cases} \tag{2.5}$$

Unfortunately, we still cannot simplify the expected average degree into a form that can be computed in subquadratic time because of the case distinction in Eq. 2.5. To circumvent this, we compute $\mathbb{E}[\bar{d}]$ for all vertex pairs as if $k \leq 0.5$ (call it Q_{over}), then add an error term Q_{error} to cancel out the pairs we treated wrongly, and add the correct value for those pairs. This results in $\mathbb{E}[\bar{d}] = Q_{\text{over}} + Q_{\text{error}}$.

Plugging the $k \leq 0.5$ case of Eq. 2.5 into Eq. 2.1 and pulling constants out of the sum yields

$$Q_{\text{over}} \cdot n = c^T \frac{2^d}{1-T} \sum_{u \in V} \sum_{v \neq u} \left(\frac{w_u w_v}{W}\right) - c\frac{2^{d/T}}{1/T-1} \sum_{u \in V} \sum_{v \neq u} \left(\frac{w_u w_v}{W}\right)^{1/T} \tag{2.6}$$

There are still quadratic sums in Eq. 2.6, but those can be simplified to

$$\sum_{u \in V} \sum_{v \neq u} \left(\frac{w_u w_v}{W}\right) = \sum_{u \in V} \sum_{v \in V} \frac{w_u w_v}{W} - \sum_{v \in V} \frac{w_v^2}{W} = W - \sum_{v \in V} \frac{w_v^2}{W}$$

and

$$\begin{aligned}
\sum_{u \in V} \sum_{v \neq u} \left(\frac{w_u w_v}{W}\right)^{1/T} &= \sum_{u \in V} \sum_{v \in V} \left(\frac{w_u w_v}{W}\right)^{1/T} - \sum_{v \in V} \left(\frac{w_v^2}{W}\right)^{1/T} \\
&= \frac{1}{W^{1/T}} \left(\sum_{u \in V} \sum_{v \in V} (w_u w_v)^{1/T} - \sum_{v \in V} w_v^{2/T}\right) \\
&= \frac{1}{W^{1/T}} \left(\left(\sum_{v \in V} w_v^{1/T}\right)^2 - \sum_{v \in V} w_v^{2/T}\right).
\end{aligned}$$

To obtain the error term Q_{error}, let E_S be the set of vertex pairs (u, v) with $0.5 < k$. So Q_{error} subtracts the $k \leq 0.5$ case and adds the $k > 0.5$ case given in Eq. 2.5 for all vertex pairs in E_S, thus

$$Q_{\text{error}} \cdot n = |E_S| - \sum_{(u,v) \in E_S} \left(c^T \frac{2^d}{1-T}\left(\frac{w_u w_v}{W}\right) - c\frac{2^{d/T}}{1/T-1}\left(\frac{w_u w_v}{W}\right)^{1/T}\right). \tag{2.7}$$

Now we are ready to find the constant c for a desired average degree using binary search over the monotone function $f(c) = \mathbb{E}[\bar{d}] = Q_{\text{over}} + Q_{\text{error}}$. The function f is given by Eq. 2.6 (using simplified sums) and adding the error Q_{error} from Eq. 2.7 for vertex pairs with $k > 0.5$.

$$\begin{aligned} f(c) = c^T \cdot \frac{2^d}{n(1-T)} &\left(W - \sum_{v \in V} \frac{w_v^2}{W} \right) \\ - c \cdot \frac{2^{d/T}}{n(1/T-1)} \cdot \frac{1}{W^{1/T}} &\left(\left(\sum_{v \in V} w_v^{1/T} \right)^2 - \sum_{v \in V} w_v^{2/T} \right) \\ - \frac{1}{n} \sum_{(u,v) \in E_S} &\left(c^T \frac{2^d}{1-T} \left(\frac{w_u w_v}{W} \right) - c \frac{2^{d/T}}{1/T-1} \left(\frac{w_u w_v}{W} \right)^{1/T} - 1 \right) \end{aligned}$$

The binary search would now take $O(n)$ time to compute the sums that are independent of c and $O(1 + |E_S|)$ per evaluation of $f(c)$. This assumes that E_S can be found efficiently, which may add additionaly overhead to the precomputation, e.g. by sorting. In the following, we further reduce the time to evaluate $f(c)$ from $O(|E_S|)$ to $O(|S|)$ with S being the set of vertices with at least one occurrence in E_S.

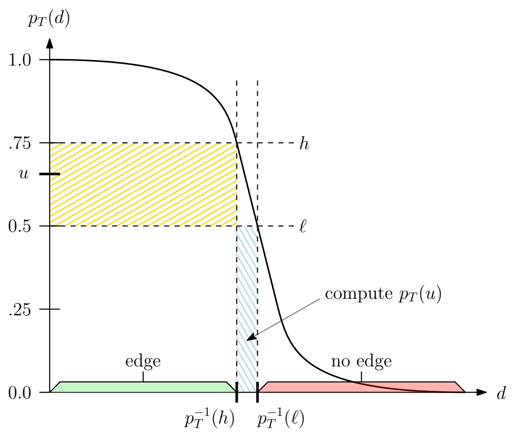
For a vertex $v \in V$, let $E_S(v) = \{u \in V \mid uv \in E_S\}$ be the set of partners in E_S. We rewrite the sum in the Q_{error} part of $f(c)$ as follows

$$\begin{aligned} &\sum_{(u,v) \in E_S} \left(c^T \frac{2^d}{1-T} \left(\frac{w_u w_v}{W} \right) - c \frac{2^{d/T}}{1/T-1} \left(\frac{w_u w_v}{W} \right)^{1/T} - 1 \right) \\ =& \sum_{v \in S} \sum_{u \in E_S(v)} \left(c^T \frac{2^d}{1-T} \left(\frac{w_u w_v}{W} \right) - c \frac{2^{d/T}}{1/T-1} \left(\frac{w_u w_v}{W} \right)^{1/T} - 1 \right) \\ =& \sum_{v \in S} \left(c^T \frac{2^d}{1-T} \left(\frac{w_v}{W} \right) \sum_{u \in E_S(v)} (w_u) - c \frac{2^{d/T}}{1/T-1} \left(\frac{w_v}{W} \right)^{1/T} \sum_{u \in E_S(v)} (w_u^{1/T}) - \sum_{u \in E_S(v)} 1 \right) \end{aligned}$$

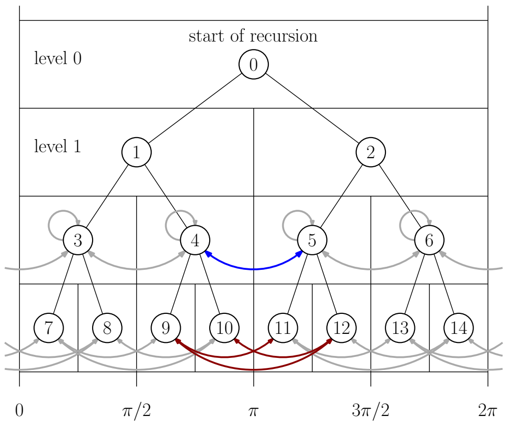
We reduce the running time by exploiting that for any two vertices $u, v \in S$, $w_u \leq w_v$ implies $E_S(u) \subseteq E_S(v)$. Thus, if we iterate the vertices in S by increasing weight, we can reuse the computations for the last vertex by maintaining $E_S(v)$ and the associated sums incrementally. Therefore, we partially sort the weights for all vertices in S. Since the upper and lower bound for the binary search are found with an exponential search, the size of S might grow until the upper bound is found. We lazily extend a sorted prefix of the weight array while raising the upper bound. We assume S to be very small compared to n and thus consider this overhead dominated by the other precomputations.

2.3.4 Avoiding Expensive Mathematical Operations for HRGs

HRGs introduce many computationally expensive mathematical operations like the hyperbolic cosine. We significantly improve the performance of the generator by avoiding or reusing the results of those operations.



(a) Sketch of the distance filter optimization to avoid computationally expensive mathematical operations, providing a speedup of 2.



(b) Visited cell pairs up to level 3. The arrows represent the 8 neighboring cell pairs in level 2 and 12 distant cell pairs in level 3.

Figure 2.4: Distance Filter (left) and tasks for parallelization in the 1-dimensional case (right).

For the threshold model, an edge exists if the distance d is smaller than R. Considering how the hyperbolic distance is defined (Section 1.2.5), reformulating it to $\cosh(d) < \cosh(R)$ avoids the expensive arccosh, while $\cosh(R)$ remains constant during execution and can thus be precomputed. Similar to recent threshold HRG generators, we compute intermediate values per vertex such that $\cosh(d)$ can be computed using only multiplication and addition [Fun+19; Pen17].

For the binomial model, evaluating the connection probability $p_T(d)$ from the optimized $\cosh(d)$ is a performance bottleneck and made up half of the total run time. Evaluating $p_T(d)$ includes an expensive exponential function and cannot avoid the arccosh like in the threshold model. We use a technique that we call a *distance filter* to reduce the frequency of the operation resulting in a speedup of approximately factor two.

To explain how the distance filter works (also see Figure 2.4a), consider the straightforward way to sample an edge. That is, one samples a uniform random value $u \in [0, 1]$ and creates the edge if and only if $u < p_T(d)$. Since p_T is monotonically decreasing, an alternative check would be $p_T^{-1}(u) > d$. The distance filter improves this by precomputing the inverse $p_T^{-1}(u)$ for equidistant values in $[0, 1]$. This lets us, for small ranges in $[0, 1]$, quickly access the corresponding range of distances. Then, the process of sampling an edge becomes the following. We first sample $u \in [0, 1]$, which falls in a range between two precomputed values, which in turn yields a range of distances. If the actual distance lies below that range, there has to be an edge and if it lies above, there is no edge. Only if it lies in the range, we actually have to compute the probability $p_T(d)$ and do the check the straightforward way. Since u is uniformly distributed, the probability to hit the interval where $p_T(d)$ has to be evaluated is $1/k$, where k is the number of precomputed entries.

Our generator uses $k = 100$ which is large enough to amortize the few times we have to compute $p_T(d)$ but still small enough to avoid cache misses. Additionally, we avoid the arccosh by directly storing $\cosh[p_T^{-1}(x)]$ in the filter.

2.3.5 Parallelization

This section describes how the sampling algorithm can be parallelized. The presented approach applies to the GIRG and HRG implementations. The algorithm has five steps: generate weights, generate positions, estimate the average degree constant, precompute the geometric data structure, and sample edges. The first two are trivial to parallelize. For estimating the constants, we parallelize the dominant computations with linear running time.

For the preprocessing we have to do three subtasks: compute for each vertex its containing cells on its insertion level, sort the vertices according to their Morton code index, and compute the prefix sum for all cells. We parallelize all three tasks and optimize them by handling all weight buckets together, sorting by weight bucket first and Morton code second. This is done by encoding this criterion into integers that are sorted with parallel radix sort. Then, the vertices of a weight bucket form a contiguous subsequence in the sorted array. Moreover, they are sorted by cell, allowing parallel computation of the prefix sums for all cells in the insertion level of the weight bucket.

To sample the edges, we make use of the fact that we iterate over cell pairs in a recursive manner. This can be parallelized by cutting the recursion tree at a certain level and distributing the loose ends among multiple processors. This works well, as the recursion tree is symmetric, leading to multiple tasks of similar load. As the number of run time intensive tasks is a power of 2, it works particularly well if the number of processors is also a power of 2 and experiments suggest a near optimal scaling in this case.

In detail, we parallelize the edge sampling as follows. Each thread has a local random generator. We use static scheduling to produce deterministic results even for the binomial model. However, the ordering of edges in the edge list varies, because each thread locally buffers generated edges before writing them while locking a mutex. We distinguish two stages of execution. The first stage is to “saw off” the recursion tree at a certain level and collect the omitted recursive calls as *tasks* to execute in stage two. A task is represented by a cell pair from which to pick up the execution later. One thread collects the tasks by traversing the recursion tree without sampling any edges (omitting lines 1-6 in Algorithm 2.1). Meanwhile, the other threads process the pairs that the main thread passed through, i.e., the work in the top of the recursion tree before the saw-off point. When all tasks are collected stage two begins. In stage two, the threads pick up the “loose ends” of the cut recursion tree. There are three different types of tasks with varying load. For 1-dimensional geometry, level $\ell > 2$, and assuming a number of threads that is constant in n, the types of tasks are the following. There are 2^ℓ *heavy tasks* given by a neighboring cell pair of the form (A, A). Their number of recursive calls grows exponentially with each subsequent level implying a load of $O(n)$. There are 2^ℓ *light tasks* given by a neighboring cell pair of the form $(A, A + 1)$. They produce four recursive calls per subsequent level implying a load of $O(\log n)$. Finally, there are $3 \cdot 2^{\ell-1}$ *constant tasks* given by a distant cell pair. They invoke no recursive calls at all. The number of distant cell pairs in a level is explained by Figure 2.4b. For each cell B in level $\ell - 1$ with children A and $A+1$, the distant cell pairs in level ℓ are $(A, A+2), (A, A+3), (A+1, A+3)$.

Since heavy tasks dominate the run time during stage two, we distribute heavy tasks evenly among all threads. This is why the approach scales best when the number of threads is a power of two. The level where we saw off the recursion tree is a tuning parameter of the generator. We choose it, such that there are two heavy tasks per thread to reduce load imbalance if one thread stalls. To apply the same scheduling approach to higher dimensions it suffices to know that the load of tasks remains similar and the number of heavy tasks is $2^{\ell d}$.

2.4 Experimental Evaluation

We perform three types of experiments. In Section 2.4.1 we investigate the scaling behavior of our GIRG generator, broken down into the different tasks performed by the algorithm. In Section 2.4.2 we compare our HRG generator with existing generators. In Section 2.4.3 we experimentally investigate the difference between HRGs and their GIRG counterpart. Whenever a data point represents the mean over multiple iterations, our plots include error-bars that indicate the standard deviation. Besides the implementation itself, all benchmarks and analysis scripts are also accessible in our source repository.

2.4.1 Scaling of the GIRG Generator

We investigate the scaling of the generator, broken down into five steps. 1. **(Weights)** Generate power-law weights. 2. **(Positions)** Generate points on $\mathbb{T}^d$. 3. **(Binary)** Estimate the constant controlling the average degree. 4. **(Pre)** Preprocess the geometric data structure (Section 2.2.4). 5. **(Edges)** Sample edges between all vertex pairs as described in Algorithm 2.1.

Figure 2.5 shows the sequential run time over the number of nodes n (top left), number of edges m (top right), temperature T (bottom right), and dimension d (bottom right). The performance is measured in nanoseconds per edge. Each data point represents the mean over 10 iterations. To make the measurements independent of the graph representation, we do not save the edges into RAM, but accumulate a checksum instead. Note that the top right plot increases the average degree, resulting in a decreased time per edge.

The empirical run times match the theoretical bounds: it is linear in n and m, grows exponentially in the dimension d, and is unaffected by the temperature T. The overall time is dominated by the edge sampling. Generating the weights includes expensive exponential functions, making it the slowest step after edge sampling. Generating the positions is significantly faster even for higher dimensions. For the parameter estimation using binary search, one can see that the run time never exceeds the time to generate the weights. For non-zero temperature T the performance of the binary search is similar to the generation of the weights, as it also requires exponential functions. The lower run

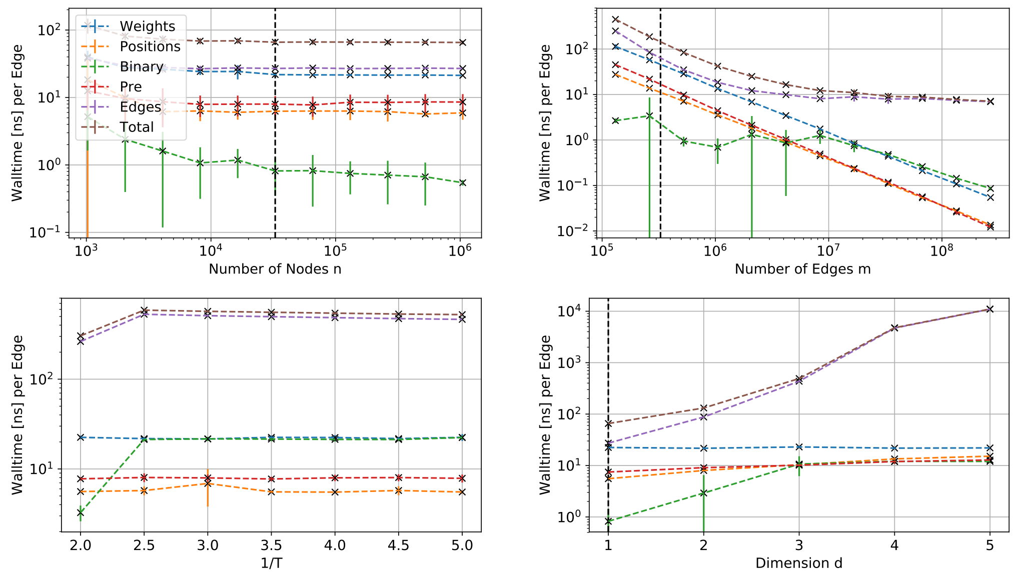


Figure 2.5: Sequential run time for the steps of the GIRG sampling algorithm averaged over 10 iterations. Each plot varies a different model parameter deviating from a base configuration $d = 1$, $n = 2^{15}$, $T = 0$, $\beta = 2.5$, and $\bar{d} = 10$. The base configuration is indicated by a dashed vertical line.

times per edge for the increasing number of edges (top right) show that the run time is dominated by the number of nodes n. Only for very high average degrees, the cost per edge outgrows the cost per vertex.

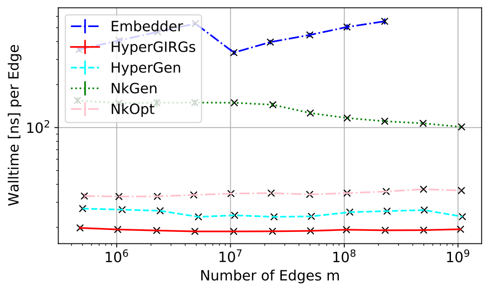
2.4.2 HRG Run Time Comparison

We evaluate the run time performance of HyperGIRGs compared to the generators in Table 2.1, excluding the generators with high asymptotic run time as well as RHG and sRHG, which are designed for distributed machines. Executed on a single compute node, the performance of the faster sRHG is comparable to HyperGen [Fun+19]. To avoid systematic biases between different graph representations, the implementations are modified[6] not to store the resulting graph. Instead, only the number of edges produced is counted and we ensure that the computation of incident nodes is not optimized away by the compiler.

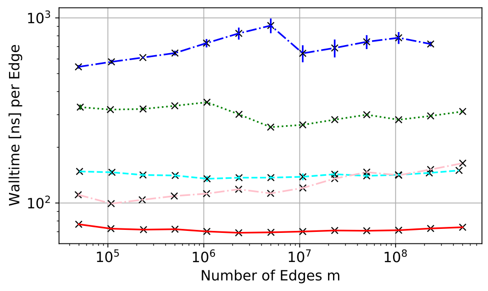
We used different machines for our sequential and parallel experiments. The former are done on an *Intel Xeon Skylake CP Gold 6144* with 192 GB RAM, the latter on an *Intel Xeon E5-2630 v3* with 8 cores (16 threads) and 64 GB RAM.

For threshold graphs, our generator HyperGIRGs is consistently faster than the competitors, independent of the parameter choices; see Figure 2.6a and 2.6b. Only for unrealistic average degrees (1 k), HyperGen slightly outperforms HyperGIRGs.

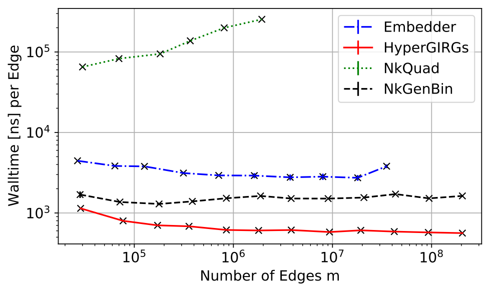
[6]The modifications are publicly available and referenced in our GitHub repository.



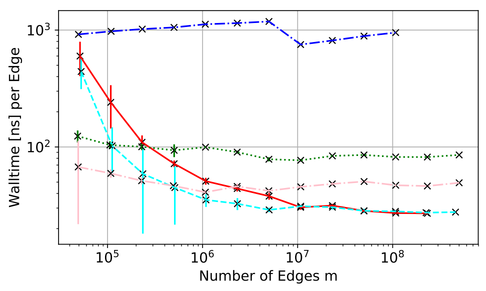
(a) $\bar{d} = 100$, $\beta = 2.2$, $T = 0$, sequential



(b) $\bar{d} = 10$, $\beta = 3$, $T = 0$, sequential



(c) $\bar{d} = 10$, $\beta = 2.2$, $T = 0.5$, sequential

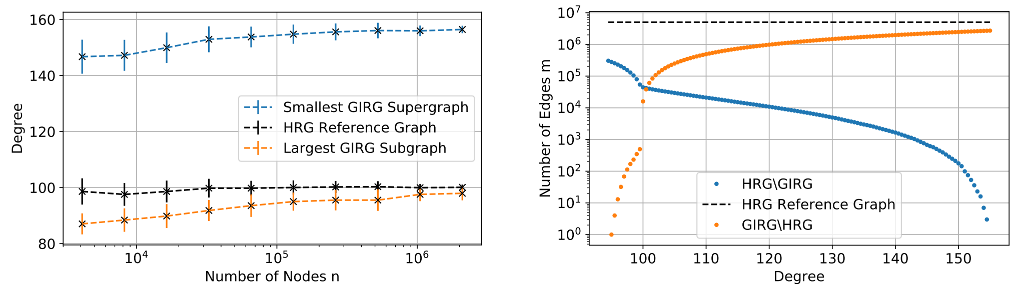


(d) $\bar{d} = 10$, $\beta = 3$, $T = 0$, parallel (16 threads)

Figure 2.6: Comparison of HRG generators averaged over 5 iterations. **(a)**, **(b)** Threshold variant for different average degrees $\bar{d}$ and power-law exponents β. **(c)** Binomial variant with temperature $T = 0.5$. **(d)** The same configuration as (b) but utilizing multiple cores.

For higher temperatures, we compare our algorithm with the three other non-quadratic generators NkQuad (included in NetworKit), NkGenBin, and Embedder; see Figure 2.6c. One can clearly see the worse asymptotic running time of NkQuad. HyperGIRGs is consistently 4 times faster than Embedder and 2-3 times faster than NkGenBin for graphs that are not too small. We note that Embedder uses a different estimation for R, which leads to an insignificant left-shift of the corresponding curve.

Figure 2.6d shows measurements for parallel experiments using 16 threads. The parameters coincide with Figure 2.6b. Embedder does not support parallelization and is outperformed even more by the other generators. For sufficiently large graphs, the fastest generator in this multi-core setting is HyperGen, which is specifically tailored towards parallel execution. Nonetheless, HyperGIRGs shows comparable performance and overtakes the other two generators NkGen and NkOpt. We note that even on parallel machines, the sequential performance is of high importance: one often needs a large collection of graphs rather than a single huge instance. In this case, it is more efficient to run multiple instances of a sequential generator in parallel.



(a) $n \in [2^{12}, 2^{21}]$, $\bar{d} = 100$, $\beta = 2.5$, $T = 0$ (b) $n = 10^5$, $\bar{d} = 100$, $\beta = 2.5$, $T = 0$

Figure 2.7: Relation between the HRG and the GIRG model. **(a)** The values for d_{HRG}, d_{GIRG}, D_{GIRG} averaged over 50 iterations. **(b)** The number of missing (HRG $\setminus$ GIRG) and additional (GIRG $\setminus$ HRG) edges depending on the expected degree of the corresponding GIRG. It can be interpreted as a cross-section of one iteration in (a).

2.4.3 Difference Between HRGs and GIRGs

Recall from Section 1.2.6 that a HRG with average degree d_{HRG} has a corresponding GIRG sub- and supergraphs with average degrees d_{GIRG} and D_{GIRG}, respectively.

We experimentally determine, for given HRGs, the values for d_{GIRG} by decreasing the average degree of the corresponding GIRGs until it is a subgraph of the HRG. Analogously, we determine the value for D_{GIRG}. We focus on the threshold variant of the models, as this makes the coupling between HRGs and GIRGs much simpler (the graph is uniquely determined by the coordinates). Figure 2.7a shows d_{GIRG} and D_{GIRG}, compared to d_{HRG} for growing n. One can see that d_{GIRG} and D_{GIRG} are actually quite far apart. They in particular do not converge to the same value for growing n. However, at least d_{GIRG} seems to approach d_{HRG}. This indicates that every HRG corresponds to a GIRG subgraph that is missing only a sublinear fraction of edges. On the other hand, the average degree of the GIRG has to be increased by a lot to actually contain all edges also contained in the HRG.

Figure 2.7b gives a more detailed view for a single HRG. Depending on the average degree of the GIRG, it shows how many edges the GIRG lacks and how many edges the GIRG has in addition to the HRG. For degree 100, the GIRG contains about 38 k additional and lacks about 42 k edges. These are rather small numbers compared to the 5 M edges of the graphs.

2.5 Conclusion

We provide the first efficient implementation of a geometric inhomogeneous graph generator and a special case adaption for hyperbolic random graphs that constitutes the fastest sequential HRG generator to date. Our code is publicly available. We describe the sampling algorithms along with crucial implementation details such as optimizations, parallelization strategies, and the non-trivial estimation of input parameters to control the average degree of the resulting graphs. Moreover, we relate the GIRG and HRG model and find that, although a straightforward inclusion does not hold, they are sufficiently similar in practice. For example a HRG with about 5 million edges and its corresponding GIRG equivalent have 99.24% of their edges in common.

3 Computing Maximum Flows in Scale-Free Networks

This chapter is based on joint work with Thomas Bläsius and Tobias Friedrich [BFW21].

3.1 Introduction

The maximum flow problem is arguably one of the most fundamental graph problems that regularly appears as a subtask in various applications [AMO93; Sch07; VB12]. The go-to general-purpose algorithm for computing flows in practice is the highest-label Push-Relabel algorithm by Cherkassky and Goldberg [CG97], which is also part of the boost graph library [Sch11]. Beyond that, the BK-algorithm by Boykov and Kolmogorov [BK04] or its later iteration [Gol+11] should be used for instances appearing in computer vision. Our main goal in this paper is to provide a flow algorithm tailored towards *scale-free* networks. Such networks are characterized by their heavy-tailed degree distribution resembling a power law, i.e., they are sparse with few vertices of comparatively high degree and many vertices of low degree.

At its core, our algorithm is a variant of Dinitz's algorithm [Din70], which is an augmenting path algorithm that iteratively increases the flow along collections of shortest paths in the residual network. In each iteration, at least one edge on every shortest path gets saturated, thereby increasing the distance between source and sink in the residual network. To exploit the structure of scale-free networks, we make use of the facts that, firstly, shortest paths tend to span only a small fraction of such networks, and secondly, a balanced bidirectional breadth-first search is able to find the shortest paths very efficiently [BN16; Blä+18a]. Using a bidirectional search to compute shortest paths in Dinitz's algorithm directly translates this efficiency to the first iteration, as the residual network initially coincides with the flow network. Though the structure of the residual network changes in later iterations, our experiments show that the run time improvements achieved by using a bidirectional search remain high. Scaling experiments with geometric inhomogeneous random graphs [BKL19] indicate that the flow computation of our algorithm runs in sublinear time. In comparison, previous algorithms (Push-Relabel, BK, and unidirectional Dinitz) require slightly super-linear time. This is also reflected in the high speedups we achieve on real-world scale-free networks.

With the flow computation itself being so efficient, the total run time for computing the maximum flow for a single source-sink pair in a scale-free network is heavily dominated by loading the graph and building data structures. Thus, our algorithm is particularly relevant when we have to compute multiple flows in the same network. This is, e.g., the case when computing the Gomory-Hu tree [GH61] of a network. The Gomory-Hu tree is a compact representation of the minimum s-t cuts for all source-sink pairs (s, t). It can be computed with Gusfield's algorithm [Gus90] using $n - 1$ flow computations in a network with n vertices. Using our bidirectional flow algorithm as the subroutine for flow computations in Gusfield's algorithm lets us compute the Gomory-Hu tree of, e.g., the `soc-slashdot` instance with 70 k nodes and 360 k edges in only 2.6 s. In this context, we observe that the Push-Relabel algorithm is also very efficient in computing the flow values by computing a preflow. However, converting this to a flow or extracting a cut from it takes significantly more time.

Our algorithm is designed to work particularly well on scale-free networks. Nonetheless, we also conducted experiments on networks that are not scale-free. We observe that our algorithm outperforms the Push-Relabel algorithm significantly on Erdős-Rényi random graphs and slightly on the Pennsylvania road network. Unsurprisingly, our algorithm is outperformed by the BK-algorithm on a segmentation instance from computer vision. Moreover, Push-Relabel performs best on a layered network that was specifically constructed to evaluate flow algorithms. However, we would argue that this type of instance is rather artificial.

Contribution. Our findings can be summarized in the following main contributions.

- We provide a simple and efficient flow-algorithm that significantly outperforms previous algorithms on scale-free networks [1].
- It's efficiency on non-scale-free instances makes it a potential replacement for the Push-Relabel algorithm for general-purpose flow computations.
- Our algorithm is well suited to compute the Gomory-Hu tree of large instances.
- In contrast to previous observations [CG97; DM89], situations exist where computing a flow with the Push-Relabel algorithm is significantly more expensive than computing a preflow.

Related Work. We briefly discuss only the work most related to our result. For a more extensive overview on the topic of flows, we refer to the survey by Goldberg and Tarjan [GT14].

Our algorithm is based on Dinitz's Algorithm [Din70], which belongs to the family of *augmenting path algorithms* originating from the Ford-Fulkerson algorithm [FF56]. Augmenting path algorithms use the *residual network* to represent the remaining capacities and iteratively increase the flow by augmenting it with paths from source to sink in the residual network, until no such path exists. At every point in time, a valid flow is known and at the end of execution, non-reachability in the residual network certifies maximality.

From this perspective, the *Push-Relabel algorithm* [GT88] does the reverse. At every point in time, the sink is not reachable from the source in the residual network, thereby

[1] `https://github.com/chistopher/scale-free-flow`

guaranteeing maximality, while the object maintained throughout the algorithm is a so-called *preflow* and the algorithm stops once the preflow is actually a flow. This is achieved using two operations *push* and *relabel*; hence the name. Different variants of the Push-Relabel algorithm mainly differ with regard to the order in which operations are applied. A strategy performing well in practice is the highest-label strategy [CG97]. The extensive empirical study by Ahuja et al. [Ahu+97] on ten different algorithms shows that the highest-label Push-Relabel algorithm indeed performs the best out of the ten. The only small caveat with these experiments is the fact that they are based on artificial networks that are specifically generated to pose difficult instances. Our experiments show that the structure of the instance matters in the sense that it impacts different algorithms differently; potentially yielding different rankings on different types of instances. The so-called pseudoflow algorithm by Hochbaum [Hoc08] was later shown to slightly outperform (low single-digit speedups on most instances) the highest-label Push-Relabel algorithm; again based on artificial instances [CH09].

Boykov and Kolmogorov [BK04] gave an algorithm tailored specifically towards instances that appear in computer vision; outperforming Push-Relabel on these instances. It was later refined by Goldberg et al. [Gol+11]. Most related to our studies is the work by Halim et al. [HYW11] who developed a distributed flow algorithm for MapReduce on huge social networks.

3.2 Network Flows and Dinitz's Algorithm

This section introduces the concept of network flow and describes Dinitz's algorithm [Din70].

3.2.1 Network Flows

A flow network is a directed graph $G = (V, E)$ with source and sink vertices $s, t \in V$, and a capacity function $c : V \times V \to \mathbb{N}$ with $c(u, v) = 0$ if $(u, v) \notin E$. A *flow* f on G is a function $f : V \times V \to \mathbb{Z}$ satisfying three constrains: (I) capacity $f(u, v) \leq c(u, v)$ (II) asymmetry $f(u, v) = -f(v, u)$ and (III) conservation $\sum_{v \in V} f(u, v) = 0$ for $u \in V \setminus \{s, t\}$. We call an edge $(u, v) \in E$ *saturated* if $f(u, v) = c(u, v)$. Denote the *value* of a flow f as $\sum_{v \in V} f(s, v)$. The maximum flow problem, *max-flow* for short, is the problem of finding a flow of maximum value.

Given a flow f in G, we define a network G_f called the *residual network*. G_f has the same set of nodes and contains the directed edge (u, v) if $f(u, v) < c(u, v)$. The capacity c' of edges in G_f is given by the residual capacity in the original network, i.e., $c'(u, v) = c(u, v) - f(u, v)$. An s-t path in G_f is called an *augmenting path*.

3.2.2 Dinitz's Algorithm

Let $d_s(v)$ be the distance from s to vertex v in G_f. We define a subgraph of G_f called the *layered network* by restricting the edge set to edges (u, v) of G_f for which $d_s(u)+1 = d_s(v)$, i.e., edges that increase the distance to the source. We call a flow *blocking* if every s-t path contains at least one edge saturated by this flow, i.e., there is no augmenting path.

Dinitz's algorithm (see Algorithm 3.1) groups augmentations into rounds. It augments a set of edges that constitutes a blocking flow of the layered network in each round. One can find such a set of edges by iteratively augmenting *s*-*t* paths in the layered network until source and sink become disconnected. After augmenting a blocking flow, the distance between the terminals in the residual network strictly increases.

Algorithm 3.1: Dinitz's Algorithm.

```
1 while s-t path in residual network do
2     build layered network
3     while s-t path in layered network do
4         augment flow with s-t path
```

3.2.3 Running Time Considerations

To better understand how our modifications impact the run time, we briefly sketch how Dinitz running time of $O(n^2m)$ is obtained. Since $d_s(t)$ increases each round, the number of rounds is bounded by $n-1$. Each round consists of two stages: building the layered network and augmenting a blocking flow. The layered network can be constructed in $O(m)$ using a breadth-first search (BFS). Finding the blocking flow is done with a repeated graph traversal, usually using a depth-first search (DFS). The number of found paths is bounded by m, because each found path saturates at least one edge, removing it from the layered network. A single DFS can be done in amortized $O(n)$ time as follows. Edges that are not part of an *s*-*t* path in the layered network do not need to be looked at more than once during one round. This is achieved by remembering for each node which edges of the layered network were already found to have no remaining path to the sink. Each subsequent DFS will start where the last one left off. Thus, per round, the depth-first searches have a combined search space of $O(m)$, while each individual search additionally visits the nodes on one *s*-*t* path which is $O(n)$.

In our experiments $d_s(t)$ remains mostly below 10, implying that the number of rounds is significantly lower than $n-1$. Also, the number of found augmenting paths during one rounds is far below m. In unweighted networks, for example, a DFS saturates all edges of the found path resulting in a bound of $O(m)$ to find a blocking flow. Dinitz's algorithm has a tight upper bound of $O(n^{2/3}m)$ in unweighted networks [ET75; Kar73].

3.3 Improving Dinitz on Scale-Free Networks

We adapt Dinitz's algorithm to exploit the specific structure of scale-free networks. We achieve a significant speedup by using the fact that a flow and cut respectively often depend only on a small fraction of the network. The following three modifications each tackle a performance bottleneck.

3.3.1 Bidirectional Search

Recently, sublinear running time was shown for balanced bidirectional search on hyperbolic random graphs [Blä+18a; BN16]. We use a bidirectional breadth-first-search to compute the distances that define the layered network during each round of Dinitz's algorithm. A forward search is performed from the source and a backward search from the sink, each time advancing the search that incurs the lower cost to advance one layer. A shortest s-t path is found when a vertex is discovered that was already seen from the other direction. Note that, for our purpose, the bidirectional search has to finish the current layer when such a vertex is discovered, because all shortest paths must be found. Figure 3.1 visualizes the difference in explored vertices between a normal and a bidirectional BFS. The augmentations with DFS are restricted to the visited part of the layered network, meaning the search space of the BFS plus the next layer.

The distance labeling obtained by the bidirectional BFS requires a change to the DFS. The purpose of the layered network is to contain all edges on shortest s-t paths. The DFS identifies edges (u, v) of the layered network by checking if they increase the distance from the source, i.e., $d_s(u) + 1 = d_s(v)$. However, we no longer obtain the distances from the source for all relevant vertices. For vertices processed by the backward search, distances to the sink $d_t(v)$ are known instead. To resolve the problem, we allow edges that either increase distance from the source or decrease distance to the sink, i.e., $d_s(u) + 1 = d_s(v)$ or $d_t(u) - 1 = d_t(v)$. This deviates from the definition of the layered network. But since edges on shortest s-t paths must both, increase the distance from the source and decrease the distance to the sink, we do not miss any relevant edges. This definition has the additional advantage that the DFS cannot deviate from shortest s-t paths in the search space of the backward search. First, the search space of the backward search can only be entered by the DFS via vertices on a shortest path. Second, when already on a shortest path, each edge that decreases the distance to the sink will also be on a shortest path.

3.3.2 Time Stamps

The bidirectional search reduces the search space of the breadth-first search and depth-first search substantially, potentially to sublinear. The initialization, however, still requires linear time. It includes distances from the source and to the sink and one progress counter per node for the augmentations. To avoid the linear initializations, we introduce time stamps to indicate if a vertex was seen during the current round. The initialization of distances and counters is done lazily as vertices are discovered during the BFS.

3.3.3 Skip Next Forward Layer

The DFS proceeds along edges outgoing from the last forward search layer independent from the target vertex being seen only by the forward search (gray in Figure 3.1) or also by the backward search (orange in Figure 3.1). However, the former type of vertex cannot be part of a shortest s-t path. By saving the number of explored layers of the forward search we can avoid the exploration of such vertices, thus limiting the DFS to

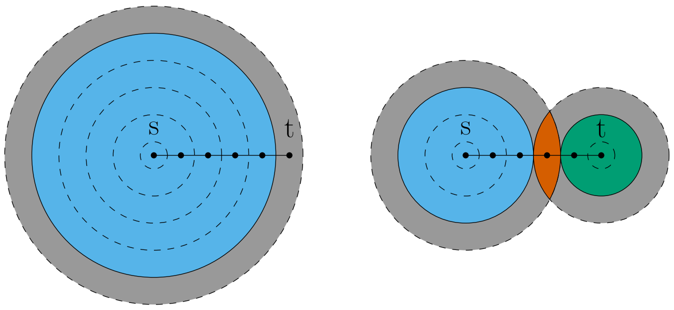


Figure 3.1: Search space of a breadth-first search from a source s to a sink t unidirectional (left) and bidirectional (right). The blue area represents the vertices that are explored, i.e., whose outgoing edges were scanned, by the forward search and the green area the backward search. In the gray area are vertices that are seen during exploration of the last layer, but not yet explored. Vertices in the intersection of the upcoming layers of the backward and forward search are marked orange.

vertices colored blue, green, or orange in Figure 3.1. With this optimization, the combined search space during augmentation (lines 3,4 in Algorithm 3.1) is almost limited to the search space of the BFS. The only additional edges that are visited originate from the intersection of the forward and backward search.

3.4 Implementation

Our implementation is based on a version of Dinitz's Algorithm that is commonly used in programming competitions[2]. In the following, we describe the details of the general implementation. Then, we elaborate our data layout, initialization, and optimizations.

3.4.1 Common Dinitz Implementations

Typical Dinitz implementations prepare the residual network by adding a reversed twin for each edge. To support undirected networks, one can represent each undirected edge as two directed edges. However, each directed edge already implies two edges in the residual network: one with the given capacity, and a reversed twin edge with no capacity. To avoid storing four times the amount of edges, the twin edge can be used to implement undirected flow. By giving the twin edge the same capacity as its counterpart, the implementation used for undirected as well as directed networks is the same.

Neither the residual network nor the layered network is constructed explicitly. The residual network is implicitly defined by the capacities and flow values on edges and the layered network by a distance labeling. This conveniently eliminates the need to modify the network structure during the algorithm. When, e.g., saturating an edge during

[2]`https://cp-algorithms.com/graph/dinic.html`

augmentation, this implicitly removes the edge from the residual network and layered network. However, with this representation, the BFS and DFS are performed on all edges and must check if edges are part of the residual or layered network when they are encountered. Note that the complexity considerations from Section 3.2.3 still apply for BFS and for DFS. The amortization argument for the DFS extends to edges that are not part of the layered- or residual network. That is, a counter into the adjacency list of each vertex indicates which outgoing edges were already processed this round.

3.4.2 Data Layout and Initialization

We represent the graph by a linearized adjacency list of outgoing edges. Edges are sorted by originating vertex in linear time. Each node stores a range of edges into this list. The Push-Relabel implementation we compare against uses the same structure. Performance-wise, the data structure significantly reduces the time to initialize large networks without negative impact on the flow computation. Another detail of our implementation is that we use begin and end indices into an array instead of a dynamically growing queue for the BFS. An array of length n is sufficient because during BFS each vertex is pushed at most once. We allocate this memory in advance and override the data each round.

We allocate memory for distance labels, counter, and the queue in advance when the network is built instead of per flow computation. The performance of initialization heavily depends on the data layout. We decided to store node data interleaved instead of in separate buffers. This data layout reduces memory loads and facilitates cache locality because all data for one node is fetched at once. On the other hand, the choice hinders efficient initialization with SIMD instructions.

3.4.3 Low-Level Optimizations

As we will see in Section 3.5.2, the BFS is the slowest part of the final algorithm. Line-by-line load analysis shows that more time is spent during the backward search than the forward search. The backward search from the sink has to consider incoming instead of outgoing edges but our implementation only maintains an adjacency list of outgoing edges. However, for each incoming edge, there is an outgoing twin edge with a reference to the incoming edge. This reference is used to determine the residual capacity of the incoming edge to check if the incoming edge is part of the residual network. We can save a memory lookup in the hot code of the algorithm, by determining the residual capacity of the incoming edge without loading it into memory. The residual capacity of an edge is obtained by subtracting the flow from the capacity. In undirected networks, the capacity of an edge is the same as that of its twin. Additionally, consistency of flow links the flow of both edges. Thus we can compute the residual capacity of incoming edges by looking only at the outgoing edges. The change improves performance by 20 to 40% in undirected networks. A similar optimization is possible for directed networks by caching the capacity of the back edge in each twin. This concept is known and was applied in previous flow implementations[3], however, we only use the optimization for undirected networks.

[3]`https://github.com/Zagrosss/maxflow`

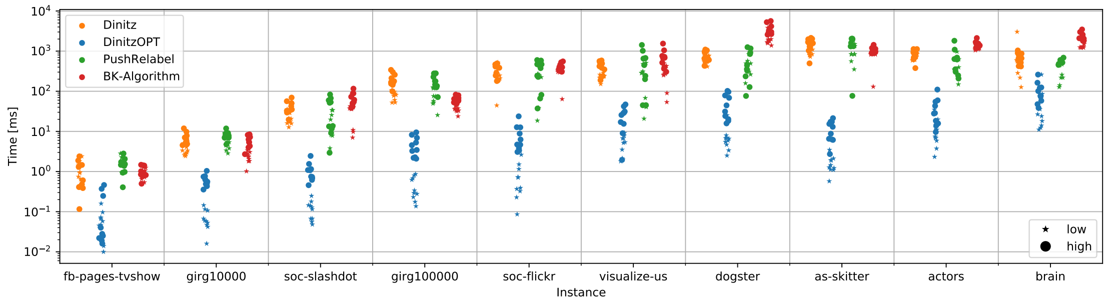


Figure 3.2: Runtime comparison of flow computations. The 20 computed flows per instance are divided into *low* and *high* terminal pairs. For *low*, the terminal degree is between 0.75 and 1.25 times the average degree. For *high*, it is between 10 and 100 times the average degree. Pairs are chosen uniformly at random from all vertices with the respective degree.

3.5 Experimental Evaluation

In this section, we investigate the performance of our algorithm *DinitzOPT*. First, we compare it to established approaches on real-world networks in Section 3.5.1. We additionally examine the scaling behavior and how the comparison is affected by problem size, i.e., if there is an asymptotic improvement over other algorithms. Then, Section 3.5.2 evaluates to which extent the different optimizations contribute to better run times and search space. In Section 3.5.3 we analyze the algorithms in a specific application (Gomory-Hu trees) and compare their usability beyond the speed of the actual flow computation. To this end, we test three different approaches to obtain a cut with the Push-Relabel algorithm. Lastly, we extend our considerations to other types of networks in Section 3.5.4, and discuss why the results on scale-free networks differ from previous studies. Recall that bidirectional search was found to perform particularly well on heterogeneous networks. For further reference, Section 3.5.5 describes the used datasets as well as integration and changes to the implementations we compare against. Experiments were done on a Dell XPS 15 9570 Laptop with an Intel Core i7-8750H CPU.

3.5.1 Runtime Comparison

In this section we compare our new approach to three existing algorithms: Dinitz [Din70], Push-Relabel [GT88], and the Boykov-Kolmogorov (BK) algorithm [BK04]. The experiments include two synthetic and eight real-world networks. All networks are undirected and all but `visualize-us` and `actors` are unweighted. We restrict our experiments in this section to the flow computation only excluding, e.g., loading times and resetting flow values between runs. For Push-Relabel we only measure the computation of the *preflow*, which is sufficient to determine the value of the flow/cut. Figure 3.2 shows the resulting run times. For this plot, the terminals were chosen uniformly at random from the set of vertices with degree close to the average (*low*) or considerably higher degree (*high*).

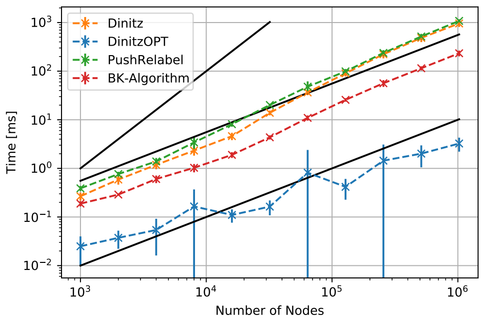
One can see that Dinitz and Push-Relabel display comparable times while BK is slightly slower on most large instances. DinitzOPT consistently outperforms the other algorithms by one to three orders of magnitude. The variance is also higher for DinitzOPT with *low* pairs approximately one order of magnitude faster on average than *high* pairs. This is best seen in the `girg100000` instance and suggests that DinitzOPT is able to better exploit easy problem instances. For all other algorithms the effect of the terminal degree on the run time is barely noticeable. Another observation is that all algorithms display drastically lower run times than their respective worst-case bounds would suggest.

The times in our experiments are close to what one might expect from linear algorithms. For example, Dinitz computes a flow on the `as-skitter` instance in one second. Considering the tight $O(mn^{2/3})$ bound on unweighted networks and assuming the throughput per second to be around 10^8 — which is a generous guess for graph algorithms — would result in an estimate of 30 minutes per flow. In contrast to our results, earlier studies found Dinitz to be slower than Push-Relabel and both algorithms clearly super-linear on a series of synthetic instances [Ahu+97]. However, these synthetic instances exhibit specifically crafted hard structures that are placed between designated source and sink vertices. These instances thus present substantially more challenging flow problems.

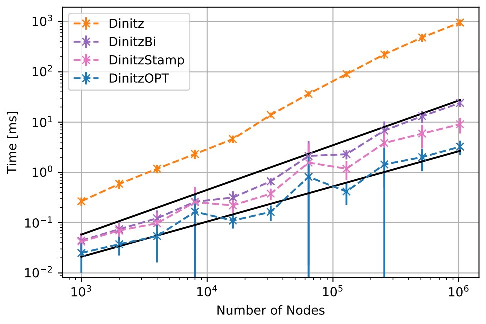
Effect of the Terminal Degree. In the following, we discuss the effect of terminal degree and structure of the cut on the run time of Dinitz and DinitzOPT. Note that the terminal degree is an upper bound on the size of the cut in unweighted networks. Moreover, the terminal degree in our experiments is based on the average degree, which is assumed to be constant in many real-world networks [Bar16]. Thus, the $O(mC)$ bound for augmenting path based algorithms, with C being the size of the cut, implies not only a linear bound for the eight unweighted networks in our experiments, but would also explain faster *low* pairs. Surprisingly, DinitzOPT exploits low terminal degrees much more than Dinitz. Another explanation for faster *low* pairs is that many cuts are close around one terminal, which is consistent with previous observations about cuts in scale-free networks [Les+09; SJN06]. Moreover, Dinitz tends to perform well when the source side of the cut is small [OZ14]. Although this does not fully explain why DinitzOPT is more sensitive to the terminal degree, we observe in Section 3.5.3 that Dinitz slows down massively when the source degree is high, even with low sink degree. Since DinitzOPT always advances the side with a smaller volume during the bidirectional search it does not matter which terminal has the higher degree.

Scaling. We perform additional experiments to analyze the scaling behavior of the algorithms. Since real networks are scarce and fixed in size, we generate geometric inhomogeneous random graphs to gradually increase the size while keeping the relevant structural properties fixed. The efficient generator discussed in Chapter 2 allows us to benchmark our algorithms on differently-sized networks with similar structure. Figure 3.3a and Figure 3.3b show the results.

We measure the run time over a series of GIRGs with the number of nodes growing exponentially from 1000 to 1 024 000 with 10 iterations each. In each iteration, we sample



(a) Runtime scaling of flow algorithms.



(b) Scaling of Dinitz variants.

Figure 3.3: (a) The average time per flow over multiple GIRGs and terminal pairs. (b) This plot differs from Figure 3.3a only in the set of displayed algorithms.

a new random graph with average degree 10, power-law exponent 2.8, dimension 1, and temperature 0. The run time for each algorithm is then averaged over 10 uniform random pairs of vertices with degrees between 10 and 20. Standard deviation is shown as error bars. The lower half of the symmetric error bars seems longer due to the logarithmic axis. We add a quadratic and two linear functions in Figure 3.3a. Figure 3.3b shows the functions $n^{0.88}$ and $n^{0.7}$ representing the theoretical upper bound and previously observed typical run times, respectively, for the bidirectional search on hyperbolic random graphs with the chosen power-law exponent [Blä+18a].

Dinitz, Push-Relabel, and BK show a near-linear running time. Compared to the linear functions in Figure 3.3a, Dinitz and Push-Relabel seem to scale slightly worse than linear, while DinitzOPT scales better than linear. In a construction with super-sink and super-source, a similar scaling was observed for Push-Relabel on the Yahoo Instant Messenger graph [Lan04].

3.5.2 Optimizations in Detail

In this section, we evaluate the performance impact of the changes discussed in Section 3.3. We present a search space analysis and in-depth profiler results[4]. All optimizations can be applied in any order and combination. Instead of considering all combinations of optimizations, we individually add them in a specific order, such that the next change always tackles a performance bottleneck. In fact, additional benchmarks reveal that the next optimization in the order speeds up the computation more than enabling all other remaining changes together. The four incrementally more optimized versions of the algorithm are: DinitzBi, DinitzReset, DinitzStamp, and DinitzOPT. Each algorithm corresponds to adding one optimization to the previous ones.

The experiments and benchmarks in this section consider 1000 uniform random terminal pairs close to the average degree on the `as-skitter` instance. The average distance between source and sink in the initial network is 4.2. The average number of rounds

[4]We used the Intel VTune profiler.

Table 3.1: Total run times and search space of visited edges for the five intermediate versions of our Dinitz implementation during the computation of 1000 flows in `as-skitter`. Terminals are chosen like *low* pairs in Figure 3.2. The first seven columns show times in seconds accumulated over all flow computations. BUILD is the construction of the residual network that is reused for all flow computations, RESET means clearing flow on edges between computations, INIT includes initialization of distances and counters per round, BFS and DFS refer to the respective subroutines, FLOW is the summed time during flow computations (sum of BFS, DFS, INIT), and TOTAL is the run time of the whole application including reading the graph from file. The last three columns contain the search space relative to the number of edges in the graph in percent. Search space columns for BFS and DFS are per round, while the FLOW column lists the search space per flow, e.g., Dinitz visits on average 65.66% of all edges per BFS and every edge is visited about 5.58 times on average in one flow computation.

			MaxFlow					Search Space [%]		
	BUILD	RESET	INIT	BFS	DFS	FLOW	TOTAL	BFS	DFS	FLOW
Dinitz	0.50	56.79	14.87	405.46	426.80	847.13	904.85	65.66	63.64	558.04
DinitzBi	0.55	58.15	21.02	2.78	8.94	32.73	91.82	0.26	1.87	8.38
DinitzReset	0.50	|	20.73	2.47	8.01	31.20	32.06	0.26	1.87	8.38
DinitzStamp	0.55	|	|	2.51	10.30	12.81	13.72	0.26	1.87	8.38
DinitzOPT	0.55	|	|	2.40	1.06	3.46	4.22	0.26	0.20	2.03

until a maximum flow is found is 4.8, whereas the last round runs only the BFS to verify that no augmenting path exists. Only counting rounds before the last round, 2.9 units of flow are found on average per round. Out of the 1000 cuts, 882 have a value equal to the degree of the smaller terminal. Table 3.1 shows profiler results and search space for Dinitz and the optimized versions of the algorithm. Figure 3.4 compares the search space with and without the bidirectional search.

Bidirectional Search. Dinitz takes 15 minutes to compute the 1000 flows and the search space per flow is more than five times the number of edges on average. Almost all of that time is spent in BFS or DFS. The bidirectional Dinitz reduces the flow-time from 14 minutes to 30 seconds, an improvement by a factor of 25.

The search space is reduced by factors of 252 for BFS, 34 for DFS, and 67 per flow. It is interesting to note, that the search space of BFS during the last round of each flow changes even more. In this round, the BFS will find no s-t path. The bidirectional search visits 39 edges on average, while the normal breadth-fist-search visits 44% of the graph. This not only emphasizes that the cuts are close around one terminal but also shows that the bidirectional search heavily exploits this structure.

The run time does not fully reflect this drastic reduction in search space, because DFS and BFS no longer dominate the flow computation. The initialization time per round increased by 50%, which can be explained by the additional distance label per node to store the distance to the sink (now 3 ints instead of 2). Although the initialization is a simple linear operation in the number of nodes, it takes twice as long as BFS and DFS combined. The real bottleneck, however, is to reset the flow values between computations. RESET takes almost a full minute which is twice as long as computing the flows.

Reset flow between computations. Between flow computations, the residual capacity of all edges has to be reset before another flow can be found. After changing the BFS to a bidirectional search, resetting the flow on all edges between computations dominates the run time. To reduce the time of our benchmarks, and to make the code more efficient in situations where multiple flows are computed in the same network, we address this bottleneck. Instead of explicitly resetting flow values for all edges, we remember the edges that contain flow and reset only those. This change is not mentioned in Section 3.3 because it does not speed up a single flow computation.

This change reduces the time for RESET to the point that it is no longer detected by the profiler, while other operations are not affected. The total time to compute all 1000 flows is thus three times lower with the flow computation making up for almost all spent time. The slowest part of the flow computation itself is still the initialization with 21 of the 31 seconds.

Time Stamps. The distance labels and counters per node are initialized each round. Using time stamps eliminates the need for initialization while adding a small overhead to DFS. The flow computation gets 2.4 times faster with 13 seconds instead of 31. After

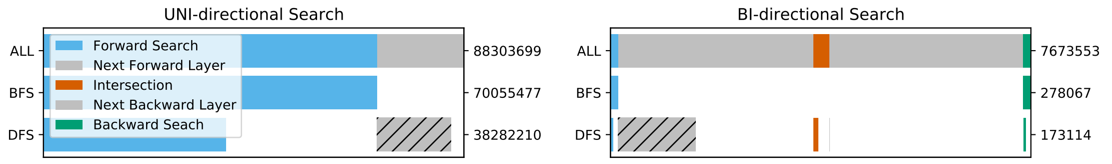


Figure 3.4: Average number of edges visited per flow computation for the terminal pairs used in Table 3.1, partitioned as in Figure 3.1. *Forward/Backward Search* represent the edges explored by the respective search. *Next Forward/Backward Layer* denote the edges that would be explored in the next step of the BFS. Edges in the *Intersection* originate from vertices in both upcoming BFS-layers. The BFS and DFS bars show the edges that are actually visited by the algorithm. The shaded area indicates the edges skipped by our last optimization (from DinitzStamp to DinitzOPT in Table 3.1) and is excluded in the sum on the right.

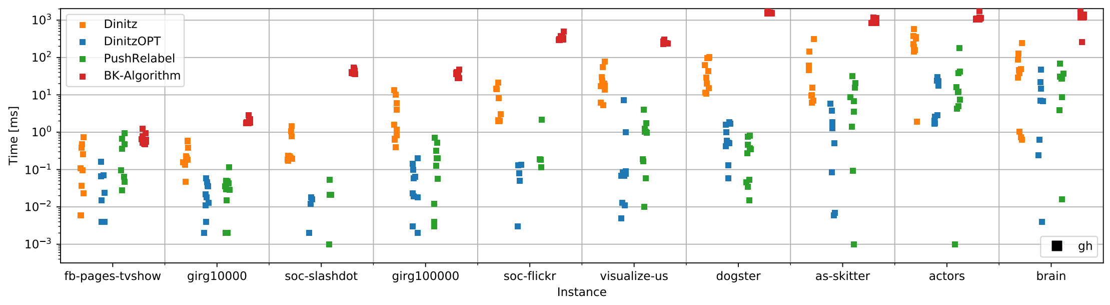


Figure 3.5: Runtime comparison of flow computations. The 10 terminal pairs per instance are uniformly chosen out of the $n - 1$ cuts required by Gusfield's algorithm.

introducing the time stamps, the DFS is the new bottleneck and makes up for about 80% of flow time.

Skip Next Forward Layer. This change prevents the DFS from visiting vertices beyond the last layer of the forward search that are not also seen by the backward search. In Figure 3.4 the skipped part is shaded. This optimization reduces the average search space for DFS during one round from almost 2% of all edges to just 0.2%. The improvement in search space is reflected by the profiler results. DFS is sped up from 10 seconds to just one second, which is faster than the BFS. The resulting time to compute all 1000 flows is 3.46 seconds, which is only 7 times slower than building the adjacency list in the beginning. In total, the time to compute the flows with the optimized Dinitz is 245 times faster than the unmodified Dinitz.

3.5.3 Gomory-Hu Trees

In the last sections, we observed that heterogeneous network structure yields easy flow problems that can be solved significantly faster than the construction of the adjacency list. This performance becomes important in applications that require multiple flows to be found in the same network. Gomory-Hu trees [GH61] fit this setting and have applications in graph clustering [FTT04]. A Gomory-Hu tree (GH-tree) of a network is a weighted tree on the same set of vertices that preserves minimum cuts, i.e., each minimum cut between any two vertices s and t in the tree is also a minimum s-t cut in the original network. Thus, they compactly represent s-t cuts for all vertex pairs of a graph. For the construction of a GH-tree, we use Gusfield's algorithm [Gus90] that requires $n-1$ cut-oracle calls in the original graph.

In this section, we evaluate the performance of max-flow algorithms for the construction of Gomory-Hu trees in heterogeneous networks. We will see that the terminal pairs required for Gusfield's algorithm yield easier flow problems than uniform random pairs. DinitzOPT is able to make use of this easy structure to achieve surprisingly low run times, and so is Push-Relabel when only considering the computation of the flow value. However, we find that the need to extract the source side of the cut hinders Push-Relabel to benefit from this performance.

Flow Computation on Gusfield Pairs. Figure 3.5 shows the same networks and algorithms as in Figure 3.2 but with terminal pairs sampled out of the $n-1$ flow computations needed by Gusfield's algorithm. The run times for all algorithms except the BK-Algorithm have high variance and are spread over up to four orders of magnitude for the larger instances. Although results for different terminal pairs vary greatly, BK seems to be the slowest algorithm followed by Dinitz. DinitzOPT and PR have comparable but significantly lower run times than the other algorithms. For example, 6 out of the 10 *gh* pairs measured for the `soc-slashdot` instance are solved by DinitzOPT and Push-Relabel faster than one microsecond which is the precision of our measurements. This suggests, that these algorithms are more sensitive to the varying difficulty of the flow computations for *gh* pairs. Our speedup over the Push-Relabel algorithm on *gh* pairs is not as pronounced as for the random pairs in Section 3.5.1. On the `dogster` instance PR is even faster than DinitzOPT on average.

To further investigate why *gh* pairs are this easy to solve, we analyze a complete run of all pairs needed by Gusfield's algorithm on the `soc-slashdot` instance. In Gusfield's algorithm, each vertex is the source once, thus the average degree of the source is the average degree of the graph (10.24). In contrast, the average degree of the sink is ca. 1500, which hinders the benefit of bidirectional search. Uni-directional Dinitz slows down by a factor of 15 when computing the flows with switched terminals. The average distance between two vertices in the original network is 4.16, but interestingly here the average distance from source to sink is 1.78. Out of the 70 k flow computations, 56 k are trivial cuts around one terminal. Computing a flow for a single s-t pair takes 2.76 rounds on average with the last round only to confirm that the flow is optimal.

DinitzOPT and Push-Relabel are both extremely fast on *gh* pairs. DinitzOPT takes 2.5

seconds to compute all $n = 70\,\mathrm{k}$ required flows, while PR needs 5 seconds. To obtain the 5 seconds for PR we exclusively measured the preflow computation, but PR is not limited by the time to compute the preflow. Actually, the entire computation of the Gomory-Hu tree on the `soc-slashdot` instance takes 12 minutes with Push-Relabel and 2.6 seconds with DinitzOPT. Instead of being caused by the Gusfield logic — which actually makes up less than 3% of the run time when using DinitzOPT as an oracle — the bottleneck when using PR as a cut oracle is not the flow computation, but initialization and extracting the cut. The drastic difference in run time is in part due to the optimizations we added to DinitzOPT to reduce the time between flow computations, while the Push-Relabel implementation recreates the auxiliary data structures, except the adjacency list, before each flow. However, in the following, we will see that a large amount of Push-Relabels run time is necessary to extract the cuts for Gusfield's algorithm.

Computing Cuts with Push-Relabel. In Gusfield's algorithm, we have to iterate over all vertices in the source side of the cut. For Dinitz algorithm we can obtain a cut by doing a BFS from the source. However, the PR algorithm only computes a preflow. We outline the following three approaches to extract the cut and show that each has major drawbacks.

Convert. Compute a preflow, convert it into a flow, then run BFS from the source.

T-Side. Compute a preflow, run BFS backwards from the sink, then take complement.

Swap. Compute a preflow from sink to source, then run BFS backwards from the source.

The most straightforward way to get a cut from a preflow is to convert it into a flow. Then, as for Dinitz, one partition of a min-cut can be identified by reachability from the source in the residual network. In previous works, the conversion from preflow to flow makes up only a small fraction of the running time [CG97; DM89]. For Gusfield pairs, however, Figure 3.6 shows that the conversion highly dominates the computation of the preflow. Only about 5 seconds of the 12 minutes of the complete run are spent in preflow computation.

To circumvent the conversion, we use the observation that one can obtain a cut directly from the preflow by finding all sink-reaching vertices in the residual network. Since Gusfield requires the source side of the cut, the complement of the found set of vertices can be used. Unfortunately, doing the backward search from the sink is even more expensive than the conversion. An explanation for this is the large sink side of the cut. Using this *T-side* approach to identify the cut for DinitzOPT takes 4.5 minutes which is a factor 100 slower than identifying the cut via the source side for DinitzOPT.

Making use of the fact that the source side of the cut is much smaller than the sink side, the drawbacks of the previous approach can be avoided in undirected networks by computing the preflow from sink to source. A cut can then be extracted by determining the vertices that can reach the original source in the residual network. The drawback of this method is that the preflow computation slows down massively from 5s to 47 minutes.

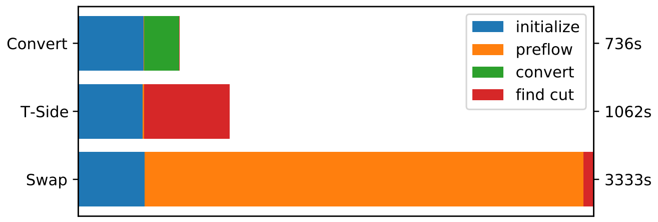


Figure 3.6: Distribution of spent time during Gusfield's algorithm on the `soc-slashdot` instance with three approaches to use the Push-Relabel algorithm as a min-cut oracle. We split the measurements into initialization, preflow, conversion, and cut identification. The time overhead for measurement, logging, and the logic of Gusfield's algorithm is included in the numbers on the right but excluded in the bars.

In conclusion, the *convert* approach is the fastest with just above 12 minutes followed by *T-side* with 18 minutes and *swap* with almost an hour. However, all three methods perform significantly worse than DinitzOPT, not because PR flow computations are slow, but both methods to avoid the four minutes run time of preflow-conversion imply even worse performance costs; either due to a breadth-first search that has to traverse almost the whole graph (T-side) or due to significantly slower preflow computations (Swap).

3.5.4 Performance on Homogeneous Networks

After evaluating the performance on heterogeneous networks we extend our experiments to networks of different structures. We consider the following networks: an Erdős-Rényi random graph [ER59] (`er100000`), an Erdős-Rényi random graph with uniform random weights in $[500, 10000]$ (`er100000_weighted`), an Erdős-Rényi random graph with super terminals (`er100000_super`), a generated layered network [Ahu+97] (`layered10000`), the road network of Pennsylvania (`roadNet-PA`), and a liver CT scan as a regular 6-connected grid (`liver.n6c100`). Further details regarding the datasets can be found in Section 3.5.5.

Figure 3.7 shows the performance of the flow algorithms on these instances. The performance on the Erdős-Rényi graphs is similar to our results for heterogeneous networks; the BK-algorithm is the slowest, followed by Dinitz, Push-Relabel, and DinitzOPT in this order. Note that a running time close to $O(\sqrt{n})$ was shown for bidirectional search on Erdős-Rényi random graphs [BN16]. Neither weights nor higher-degree terminals change how the algorithms compare to each other.

The layered network, which is specifically constructed to produce a computationally difficult flow instance [Ahu+97], is indeed more difficult than the others. In the layered network, Push-Relabel is at least five times faster than Dinitz. DinitzOPT is 10-20% slower than Dinitz. After all, our optimizations trade a small overhead during flow computation for the possibility of sublinear running time on particularly easy instances.

For the road network, the choice of the algorithm does not matter as much as for the other instances. The choice of the terminal pair, however, affects the performance

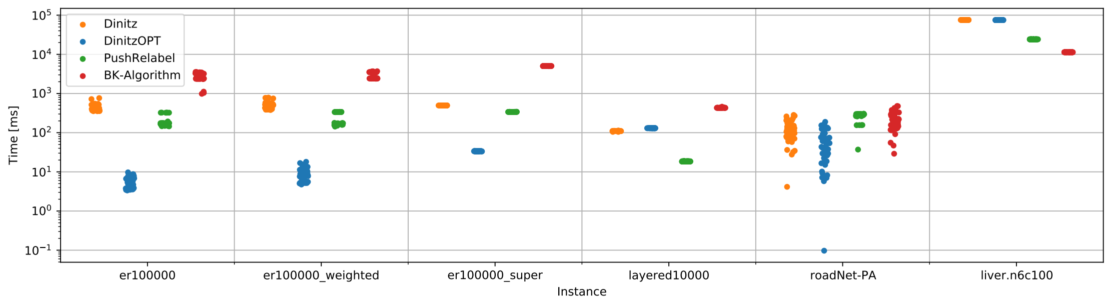


Figure 3.7: Run time of max-flow computations for various networks. Each point corresponds to one s-t flow. For each instance we computed 50 s-t flows. The instances `er100000_super`, `layered10000`, and `liver.n6c100` have designated terminals. For `er100000`, `er100000_weighted`, and `roadNet-PA` terminals are chosen uniformly at random. Unlike the experiments in Section 3.5.1, the algorithms rebuild their internal data structures including the adjacency list before each flow computation. This was necessary to prevent the BK-algorithm from reusing search-trees, which makes the instances with given terminal pairs trivial after the first run.

immensely. With a diameter of almost 800 and a very homogeneous degree distribution, the uniform random choice of terminal pairs produces problems of varying difficulty. Dinitz, BK, and DinitzOPT capitalize on the easier pairs, while Push-Relabel shows less variance between pairs.

Lastly, the liver scan produces different results than previous instances. The BK-algorithm was specifically designed for this kind of network structure and application. Unsurprisingly, the BK-algorithm performs best, followed by Push-Relabel, Dinitz, and DinitzOPT.

3.5.5 Data and Implementations

Table 3.2 lists the instances were used throughout this chapter. We obtained the datasets from the University of Koblenz (KONECT) [Kun13], the Network Repository website [RA15], as well as the Stanford Network Analysis Project (Snap) [LK14]. Furthermore, we used our GIRG generator with default parameters. We implemented the ER model and the layered network construction from Ajuja et al. [Ahu+97]. The parameters for ER are $n = 100000$ and $p = 0.02$. The parameters for the layered network are taken from the largest instance in their paper (W=71, L=141, d=10). Lastly, the `liver.n6c100` instance is from the University of Western Ontario. It is a regular 3D grid with 170x170x144 nodes, 6 edges per node, capacities up to 100, and a super sink/source. We converted all instances to a text-based edge list with zero-based indices except for Section 3.5.4 where we use the directed DIMACS format instead. The road network was undirected and is converted to the directed DIMACS format. The number of edges for the road network refers to the undirected version.

Table 3.2: Instances used in this chapter.

instance	directed	weighted	nodes	edges	avg. degree	source
fb-pages-tvshow			4K	17K	8.87	Network Repository
girg10000			10K	60K	11.99	generated
soc-slashdot			70K	360K	10.24	Network Repository
girg100000			100K	600K	12.00	generated
soc-flickr			514K	3.2M	12.42	Network Repository
visualize-us		✓	594K	3.2M	10.92	Network Repository
dogster			427K	8.5M	40.03	U. Koblenz
as-skitter			1.7M	11.1M	13.08	U. Stanford
actors		✓	382K	15.0M	78.69	U. Koblenz
brain			178K	15.8M	176.47	Network Repository
er100000	✓		100K	20M	199.94	generated
layered10000	✓		10K	100K	9.96	generated
roadNet-PA	(✓)		1.1M	1.5M	2.83	U. Stanford
liver.n6c100	✓	✓	4.1M	25M	6.04	U. Western Ontario

BK-Algorithm. We use the BK implementation from the web page of Vladimir Kolmogorov[5] that was written for the original paper [BK04]. Boost provides another version[6], but we found the original one easier to use because its interface is tailored towards multiple flow computations and provides easy and efficient access to the found cut. For each s-t flow we add edges with infinite capacity between s, t and the virtual terminals. After the flow is computed, we remove these edges again. This $O(1)$ work is included in time measurements. We apply the *reuse trees* feature and mark the changed terminals between flow computations accordingly. Memory is allocated on network construction and not per flow. We use 64-bit floating point numbers instead of integers to represent flow values and capacities to support applications (e.g. [FTT04]) that require non-integer capacities. The implementation requires additional checks to handle floating point imprecision. We applied this to all compared implementations (not just BK) and observed a performance drop of approximately 10% for all algorithms. Note that the range in which 64-bit floats exactly represent integral numbers even exceeds the range of 32-bit integers. However, precision issues are caused by the *infinity* capacity edges. To resolve this, the representation of infinity on these edges must be chosen according to the range of capacities.

Push-Relabel. The original implementation, used for example in [VB12], is no longer available[7]. We use the C++ version of the original implementation provided in Boost[8]. The Boost version is mostly the same code (up to same variable names) ported to C++, but is data structure agnostic. Therefore, we reimplemented the linearized adjacency list data structure used in the original implementation that we described in Section 3.4.

[5]`http://pub.ist.ac.at/~vnk/software.html`
[6]`https://www.boost.org/doc/libs/1_72_0/libs/graph/doc/boykov_kolmogorov_max_flow.html`
[7]was `http://www.avglab.com/andrew/soft.html`
[8]`https://www.boost.org/doc/libs/1_72_0/libs/graph/doc/push_relabel_max_flow.html`

3.6 Conclusion

We presented a modified version of Dinitz's algorithm with greatly improved run time and search space on real-world and generated scale-free networks. The scaling behavior appears to be sublinear, which matches previous theoretical and empirical observations about the running time of balanced bidirectional search in scale-free random networks. While these theoretical bounds apply during the first round of our algorithm, it is still unknown whether the analysis can be extended to account for the changes in the residual network. Our experiments, however, indicate that the search space remains small in subsequent rounds.

We observe that the low diameter and heterogeneous degree distribution lead to small and unbalanced cuts that our algorithm finds very efficiently. The flow computations required to compute a Gomory-Hu tree are even easier, making usually insignificant parts of the tested algorithms a bottleneck. For example, the preflow conversion leads to Push-Relabel being greatly outperformed by our algorithm in this setting.

4 Computing Directed Minimum Spanning Trees

This chapter is based on joint work with Maximilian Böther and Otto Kißig [BKW23]. The idea originated in a student project about the spread of infections organized by Karen Seidel at the Hasso Plattner Institute.

4.1 Introduction

The minimum spanning tree problem is well studied with various applications [GH85; SS84] and algorithms [Jar30; Kru56; Pri57]. The directed version, called the minimum spanning arborescence problem, has received much less attention. For a given root r, it aims at finding a directed spanning tree of minimum weight rooted at r. The applications include infection chain modeling [Jom+10] and the approximation of traveling salesperson instances [SS20]. Different versions and generalizations were studied [Geo03; KKT09; Kam14]. Sometimes multiple roots are given or it is required to find the best root. Historically, the problem was to find a set of non-overlapping trees with maximum total weight, called an optimum branching. As these versions are linear time equivalent [Edm67; Men+06], we focus on the minimum spanning arborescence problem a with given root.

The algorithm to find a minimum spanning arborescence was discovered independently by Edmonds [Edm67], Chu [Chu65], and Bock [Boc71]. Karp [Kar71] was the first to give a combinatorial proof of correctness. Following the literature, we call it Edmonds' algorithm. The algorithm runs in $O(nm)$ and forms the basis for later, more elaborate versions by Tarjan [Tar77; CFM79] running in $O(\min(n^2, m \log n))$ and Gabow et al. [Gab+86] running in $O(n \log n + m)$. We refer to the latter as the GGST algorithm. There exist parallel algorithms for different settings of distributed computing [Lov85; FO19]. They are based on Edmond's Algorithm as well but we will focus solely on the sequential setting. Both Tarjan's versions and GGST have the same complexity for very sparse and very dense graphs while the GGST version beats Tarjan's by a logarithmic factor for the regime in between. GGST likely is optimal since the problem of finding a minimum spanning arborescence is at least as hard as finding an (s,t)-shortest path [FO21] and comparison based sorting can be reduced to determining the order of contractions performed during Edmonds' algorithm [Gab+86]. However, a time of $O(m \log \log n)$ was obtained in the word RAM model with Tarjan's version [Men+06]. Moreover, Tarjan's version was shown to run in $O(n \log^2 n + m)$ on Erdős-Rényi graphs with random weights [Tar77; ER59].

To the best of our knowledge, no experimental evaluation of these algorithms, or even an implementation of GGST, exists. The latter is likely due to the rather technical description and the fact that the algorithm is not the main result of the corresponding paper. On the other hand, there exist some efficient (meaning $O(m \log n)$) implementations of Tarjan's version. The problem is a niche topic in coding competitions such as the International Collegiate Programming Contest (ICPC). Unfortunately, they are hard to find because most of them are only documented as submissions in online judge systems. The only ready-to-use library implementations run in $O(n^2)$. This paper provides accessible descriptions and implementations as well as a detailed evaluation. Our code is open source and can be found in our public repository[1]. The core contributions of this paper include

- five Tarjan implementations with different underlying data structures, one of which beats existing solvers on most instances,
- a high-level description of the GGST algorithm with several optimizations/simplifications,
- an efficient implementation of the GGST algorithm,
- and a detailed experimental evaluation on a large number of real-world and synthetic networks.

In Section 4.2 we describe Edmonds' algorithm along with the two versions by Tarjan [Tar77] and Gabow et al. [Gab+86]. Section 4.3 describes the existing and new implementations as well as optimization techniques. The experimental evaluation is presented in Section 4.4. We conclude in Section 4.5.

4.2 Edmonds' Arborescence Algorithm

We discuss Edmonds' algorithm in Section 4.2.1, Tarjan's version in Section 4.2.2, and the GGST version in Section 4.2.3. The latter two yield just the weight of the optimal solution, not the actual edges. Reconstructing the edge set is discussed in Section 4.2.4.

4.2.1 Edmonds' Original Version

Edmonds' algorithm works as follows. For each vertex $v \neq r$, pick the cheapest incoming edge $\pi(v)$. If the set of these $n-1$ edges contains no cycles, it is an arborescence; otherwise, it is possible to show that there is an optimal solution that contains all chosen edges except one for each cycle. To determine which edge of each cycle to remove, Edmonds' algorithm contracts each cycle. Note that a vertex is a part of at most one cycle. The weight of all edges going into a cycle C is reduced as follows. An edge pointing at vertex $v \in C$ is reduced by the weight of $\pi(v)$, i.e., the weight of the cheapest edge incoming into v. We then compute a solution on the contracted graph. The resulting solution has an incoming edge for each cycle C we contracted. This edge corresponds to an original edge (u, v) with $v \in C$, which we use to replace the cycle edge $\pi(v)$ we picked earlier.

The correctness is based on the following fact. Adding a constant Δ to all incoming edge weights of a vertex changes the weight of each arborescence by Δ, since each solution

[1]`https://github.com/chistopher/arbok`

picks exactly one of those. This means that the edge cost changes performed during the algorithm preserve the optimal solution. Moreover, the cycle edges all get a cost of zero. Thus, the final cost is the same, no matter which edge is replaced.

4.2.2 Tarjan's Version

Tarjan proposed a version of Edmonds' algorithm that, given the right data structures, runs in $O(m \log n)$ or $O(n^2)$ [Tar77]. It features two major improvements. First, the cycle expansion and removal of one edge per cycle is detached from the main algorithm and seen as a postprocessing step. The algorithm tracks all chosen edges as a superset of the solution, which can be reconstructed afterward in linear time. The second change is to formulate the algorithm sequentially in such a way to avoid rebuilding the graph for each contraction. The approach goes as follows. While there is a vertex other than the root that was not processed yet, its cheapest incoming edge which is not a self-loop is added to the solution. If this edge forms a cycle with previously chosen edges, the cost of edges into the cycle is changed as in Edmonds' algorithm and the cycle vertices including their incoming edges are merged into a vertex representing the cycle. This vertex is then added to the queue of unprocessed vertices.

The algorithm requires data structures to find the cheapest incoming edge, recognize cycles of chosen edges, and track contractions. The latter two can be achieved with disjoint set union (DSU) data structures such as a disjoint set forest [GF64; Tar75]. To find cycles, a DSU maintains weakly connected components with respect to the chosen edges. Note that each vertex has at most one incoming chosen edge. Thus, an edge closes a directed cycle with previously chosen edges, if and only if, it connects two vertices in the same weakly connected component. A second DSU is used to manage contractions and map original vertices to contracted vertices. The endpoints of edges are not updated after each contraction. Instead, a DSU lookup is required each time the algorithm handles an edge.

The data structure to maintain incoming edge sets must support four operations: (1) add an element, (2) extract the minimum element, (3) change the weight of all elements in the set by a constant, and (4) merge two sets. If all operations take at most logarithmic time, the algorithm runs in $O(m \log n)$. Most mergeable heaps (e.g., hollow heaps, treaps, skew heaps) support operations (1), (2), and (4) and can be extended with lazy propagation to allow for operation (3). Alternatively, if operations 2-4 run in $O(n)$, e.g., when using an adjacency matrix, the algorithm runs in $O(n^2)$, which is better for dense graphs.

4.2.3 GGST Version

Gabow et al. [Gab+86] further refine the version given by Tarjan to reduce the running time to $\mathcal{O}(n \log n + m)$. They use the last remaining degree of freedom, namely the order in which vertices are processed. The authors suggest to always process the vertex next from which the last chosen edge originated thus forming a path of processed vertices, called the *growth path*. To avoid special cases when the path reaches the root, they add

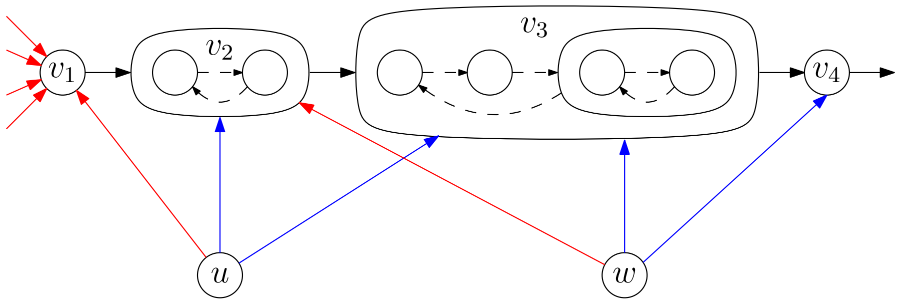


Figure 4.1: Visualization of the growth path with the first four vertices v_1, v_2, v_3, v_4. Additionally, the exit lists of two arbitrary vertices u, w are shown. Active edges are red and passive edges are blue.

dummy edges with cost 0 from the root to all other vertices making the graph fully connected. These edges do not affect the running time but they simplify the description since the algorithm becomes oblivious to the root and the additional edges can be removed in the reconstruction phase. The improved running time is achieved by exploiting the structure of the path and clever handling of associated edges. For each vertex (on or outside the growth path) an *exit list* contains outgoing edges pointing into the growth path. The elements of an exit list are sorted by the position of their target vertex in the growth path, i.e., the first edge points closest to the head of the path. The first edge in each exit list is called *active*; all others are *passive*. The active edges are maintained in a data structure we call an *active forest*. Figure 4.1 shows an example. In the following, we give a concise, yet comprehensive, description of the algorithm that differs from the original discussion in the level of abstraction and simplifies the logic and data structures.

The algorithm starts with an arbitrary vertex and repeatedly picks the cheapest incoming edge of the path head until the path covers the whole graph. In each iteration, the path is either extended or contracted. If the origin of the picked edge is not yet on the growth path, then it becomes the new path head. If it is already on the growth path, then the prefix of the growth path up to this vertex forms a cycle, which is contracted into a single vertex that becomes the new path head. The process is summarized in Algorithm 4.1. As in Tarjan's version, contractions are tracked with a DSU [GF64; Tar75] which handles the $find(u)$ calls.

Growth Path Extension. When the growth path is extended by a new vertex u, all incoming edges of u are introduced to the algorithm and inserted into their respective exit lists. Consider the insertion of an edge, say (x, u), into x's exit list. Since u has just become the new head of the growth path, the edge will be inserted at the front of the exit list. It will become active and, if the exit list was not empty, the previously active edge will become passive. Because u has just become part of the growth path, it is not a contracted vertex. However, x may be on the growth path and therefore x may be a contracted vertex. As such, x may have multiple outgoing edges to u, originating from different vertices inside x. To deal with this issue (and with multi-edges in the input), one

Algorithm 4.1: Minimum arborescence algorithm by Gabow et al. [Gab+86]

```
1  initialize growth path with arbitrary vertex;
2  insert its incoming edges into exit lists;
3  while not all vertices on growth path do
4      query min. incoming edge $(u, v)$ of path head from active forest;
5      remember $(u, v)$ for reconstruction;
6      if find(u) is not on growth path then
7          insert $u$'s incoming edges into exit lists;
8      else
9          delete prefix of path up to last occurrence of $find(u)$;
10         update incoming edge costs for all vertices on prefix;
11         delete outgoing edges of prefix from exit lists;
12         merge prefix in DSU and Active Forest;
13         limit edges into the cycle to at most 1 per origin;
14     insert $find(u)$ at front of path;
```

checks if the first edge in the exit list already points to u and if so, only keeps the cheaper one. This limits the exit list to at most one edge pointing to u. Thus we maintain the invariant that an exit list never contains two edges to the same vertex.

Growth Path Contraction. When a prefix of the path forms a cycle, it is contracted just as in Edmonds' algorithm. That is, the prefix is removed from the path, incoming edges into the cycle are reduced in cost, edges resulting in self-loops are deleted, the cycle vertices are contracted in the DSU as well as in the active forest, and multi-edges are removed (see lines 9 - 13 in Algorithm 4.1).

The cost reduction (line 10) is done with the DSU which can be modified to track an offset for each vertex [Gab+86]. Whenever the current cost of an edge is needed, a DSU lookup analogous to a $find$ is made to get the offset of the target vertex.

Self-loops are outgoing edges from the cycle. So by deleting all edges in exit lists of cycle vertices, self-loops are avoided (line 11). This also deletes edges pointing further down the path but these are irrelevant to the algorithm. They can only become incoming edges of the head if, in the future, the path is contracted up to their target, and in this case they would be self-loops.

Edges that became multi-edges by the contraction are consolidated (line 13). For each vertex with more than one edge pointing into the cycle, the prefix of their exit list that points into the cycle is deleted except for the cheapest of those edges. If a vertex has more than one edge pointing into the cycle, at least one of them is passive. Thus, such vertices can efficiently be found by maintaining, for each vertex on the growth path, a list of incoming passive edges, called a *passive list*. If each edge stores a handle into the passive list and exit list it is in, then these lists can be maintained without asymptotic overhead. We propose a more efficient method in Section 4.3.4.

Active Forest. The active forest maintains all currently active edges and it must be updated accordingly. It stores for each vertex the outgoing active edge and a set of incoming active edges. We associate an active edge with the vertex it originates from. The active forest is able to

INSERT an active edge for a vertex that does not yet have one in $O(1)$,

REPLACE the active edge of a vertex by another one that points closer to the growth path head or points to the same vertex but has less weight in $O(1)$,

DELETE the active edge of a vertex in $O(\log n)$,

MERGE the sets of incoming active edges for the first two vertices of the growth path in constant time, and

QUERY the minimum incoming active edge of the path head in $O(\log n)$ amortized time.

This is implemented as follows. Each vertex stores its incoming active edges in a Fibonacci heap [FT87], which enables the operations INSERT, DELETE, MERGE, and QUERY by just mapping them to the corresponding Fibonacci heap operations. The REPLACE operation could be implemented as a DELETE followed by an INSERT. Unfortunately, this results in a running time of $O(\log n)$. Instead, Gabow et al. [Gab+86] suggest to reuse the internal heap node representing the old edge. The node is *moved* from the heap the old edge is in to the heap where the new edge should be and receives the new edge as a key. This *move* takes $O(1)$ time and is the crucial point where the logarithmic factor over Tarjan's version is saved. The move operation is possible by restricting QUERY, REPLACE, and MERGE to the structure of the growth path. In general, no mergeable heap data structure is known that lifts these restrictions and still supports something like a constant time move [Men+06].

However, the *move* has two major problems for which we need to understand some internals about Fibonacci heaps. A Fibonacci heap is a forest whose roots are kept in a list called the *root list* of the heap. Each tree maintains the heap property, i.e., the key of a child node is higher or equal to the key of its parent. The key in our case is the weight of the corresponding active edge. Also, a Fibonacci heap usually maintains the minimum key of nodes in the root list to allow queries in constant time. The first problem of the move operation is that the cached minimum of a root list cannot be updated in constant time if the current minimum is moved out of that list. Therefore, we do not maintain the minimum. Instead, the QUERY operation rebuilds the root list, which is a common operation for Fibonacci heaps usually done upon extraction of the minimum, resulting in an amortized $O(\log n)$ running time. The second problem is that moving an internal heap node actually moves the whole subtree rooted at this node. Descendants of the node are displaced into the wrong heap and, moreover, changing the key of the moved node can violate the heap property. To fix the displacement, every time a Fibonacci heap operation would put a node into the root list it is returned to the root list of the heap the node actually belongs to, which we call the *home heap* of that node. That is, the *home heap* of a heap node is the heap of the target vertex of the corresponding active edge. Finding

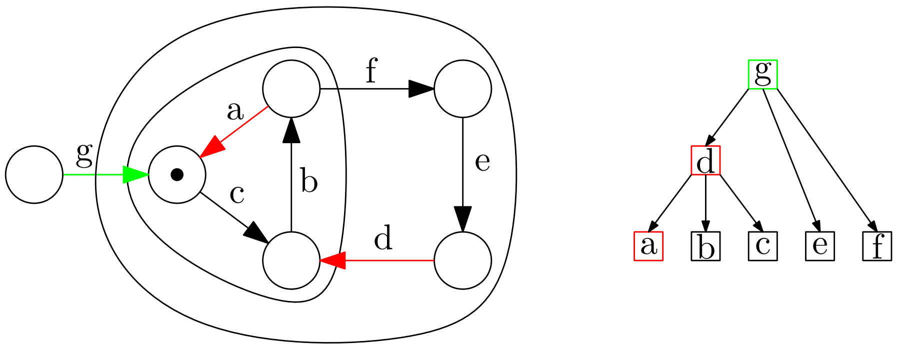


Figure 4.2: Example graph (left) and corresponding reconstruction forest (right). Assume Gabow starts at the vertex marked with a dot and the edges are labeled alphabetically in the order they were added to the growth path. The first step of the reconstruction process is indicated by colors.

the home heap requires a DSU lookup because the target vertex might be contained in a contracted vertex. Gabow et al. [Gab+86] prove the following three invariants to address the violated heap property and the correctness of the home heap fix. (1) The root of any tree is always in its home heap. (2) The heaps maintain an additional heap property w.r.t. their home heaps ordered by the position in the growth path. That is, the home heap of a parent node is at least as close to the growth path head as the home heaps of its children. (3) The original heap property is never violated between two nodes that are in their home heap. Only displaced nodes can temporarily violate the heap property.

Time Complexity. The growth path is extended at most n times. Since contracting a cycle of length l reduces the total number of vertices in the graph by $l-1$, there are at most $n-1$ contractions and the summed length of all contracted cycles is less than $2n$. Thus, QUERY, DELETE, and MERGE are called $O(n)$ times on the active forest. Furthermore, the following operations happen at most once per edge and can all be done in constant time. Insertion into an exit list, the active forest, or passive list, REPLACE in the active forest, deletion from an exit list, and deletion from a passive list. The DSU imposes no additional overhead since, if there are at least $n \log n$ calls to $find$, each individual one takes amortized constant time [Tar75]. In total this yields a running time of $O(n \log n + m)$.

4.2.4 Arborescence Reconstruction

Although Tarjan [Tar77] proposed to split off the reconstruction phase from the main algorithm, the reconstruction method given in the paper is incorrect. A note by Camerini et al. outlines a working method [CFM79]. Consider a new graph called the *reconstruction forest* where the nodes are the edges that are picked by the arborescence algorithm. In the forest, an edge that was picked as an incoming edge to a contracted vertex has directed arcs to the edges that constitute the top-level cycle of the contracted vertex. A leaf in

Table 4.1: Overview of arborescence algorithms. Tarjan+Path means they implement Tarjan's version but adjust the order in which vertices are processed to form a path as in the GGST version.

Solver	Author/Source	Variant	Data Structure	Runtime
FELERIUS	David Stangl	Tarjan	Skew Heap [ST86]	$O(m \log n)$
SPAGHETTI	Takanori Maehara	Tarjan+Path	Skew Heap [ST86]	$O(m \log n)$
YOSUPO	Kohei Morita	Tarjan+Path	Pairing Heap [Fre+86]	$O(m \log n)$
LEMON	LEMON 1.3.1	Tarjan+Path	Adjacency List	$O(n^2)$
ATOFIGH	Ali Tofigh	Tarjan	Adjacency List	$O(n^2)$
MATRIX	this paper	Tarjan	Adjacency Matrix	$O(n^2)$
TREAP	this paper	Tarjan	Treap [SA96]	$O(m \log n)$
HOLLOW	this paper	Tarjan	Hollow Heap [Han+17]	$O(m \log n)$
SET	this paper	Tarjan	Red Black Tree [GS78]	$O(m \log^2 n)$
PQ	this paper	Tarjan	Binary Heap [Cor+09]	$O(m \log^2 n)$
GGST	this paper	GGST	Fibonacci Heap [FT87]	$O(n \log n + m)$

the reconstruction forest corresponds to the first picked incoming edge of a vertex of the original graph. Figure 4.2 shows an example. The reconstruction process repeatedly selects a root of the forest. The corresponding edge becomes part of the final solution. The target vertex in the original graph of the selected edge has an associated leaf in the reconstruction forest. The process deletes the path from this leaf to the selected root from the forest, then proceeds with the next root.

A very concise implementation is possible by noting that the order in which the main algorithm picks edges is a reverse topological order of the reconstruction forest. Thus, all roots are found by iterating over the picked edges in reverse and skipping already deleted ones. Further required information is the leaf of each original vertex and the parent for each node in the reconstruction forest. The former is computed by iterating over the picked edges to find the first occurrence of each target. The latter must be saved by the main algorithm each time it contracts a cycle.

4.3 Implementation

This section introduces different solvers for the minimum arborescence problem and highlights their key points as well as optimizations and deviations from the abstract description in Section 4.2. We compare 11 solvers; our five versions of Tarjan's approach using different data structures, five external Tarjan-based solvers, and our Gabow implementation (see Table 4.1). External solvers fall into two categories, namely coding competition code and library solvers.

4.3.1 Competition Codes

Coding competitions occasionally feature arborescence tasks that require an efficient implementation for sparse graphs. These implementations are often not as maintainable or usable as library solvers but they are written with a high focus on performance. The online judge platform Library Checker[2] contains a test set for the minimum arborescence problem. We include the jury solution by the maintainer Kohei Morita as well as the fastest submission by David Stangl. We denote them by YOSUPO[3] and FELERIUS[4] according to their pseudonyms on popular contest websites. Furthermore, there is a competition-style implementation by Takanori Maehara[5], which we denote by SPAGHETTI. With around 130 lines of code, it is the most concise implementation. However, it lacks the reconstruction phase and a proper memory management.

4.3.2 Library Solvers

The two library implementations we consider are LEMON and ATOFIGH, both running in $O(n^2)$. The former is part of the LEMON library for graph algorithms[6]. We use the latest release 1.3.1 from 2014. They save incoming edges in arrays. The merge is done by iterating over all incoming edge lists of cycle vertices while collecting the cheapest edge into the cycle for each origin. They reuse the same collecting array each time and clear the used entries afterward, such that the merge is not $O(n)$ but linear in the number of merged edges. Thus, the solver is faster the fewer edges are involved in each contraction. The second library implementation, ATOFIGH, was written by Ali Tofigh and Erik Sjölund[7] using the Boost Graph Library [Sch11]. They also represent incoming edge sets as dynamically growing arrays. However, the arrays are sorted by origin vertex and the merge is done with the linear time merge routine usually known from merge sort. It was modified to remove multi-edges by only keeping the cheapest one for each origin. The same performance considerations apply. Note that there exist sparse networks where these merge strategies yield quadratic running time.

4.3.3 Our Tarjan-based Solvers

Our Tarjan code shares the logic for the algorithm and reconstruction and differs only in the data structure to manage the sets of incoming edges (see Section 4.2.2). The MATRIX solver maintains an adjacency matrix and performs the operations in linear time. The HOLLOW and TREAP solvers use our implementations of Hollow heaps [Han+17] and Treaps [SA96], respectively, which both support lazy propagation to update weights. The hollow heap is not required to implement the usual decrease key operation as it is not required by the algorithm, which allows for implementing the merge operation efficiently,

[2]`https://judge.yosupo.jp/problem/directedmst`
[3]`https://codeforces.com/profile/yosupo`
[4]`https://codeforces.com/profile/Felerius`
[5]`https://github.com/spaghetti-source/algorithm`
[6]`https://lemon.cs.elte.hu/trac/lemon`
[7]`https://github.com/atofigh/edmonds-alg`

by simplifying some bookkeeping tasks. The SET and PQ variants use the `std::set` and `std::priority_queue` data structures from the C++ standard template library. They are typically implemented as a red-black tree [GS78] and a binary heap [Cor+09], respectively. Since the set and priority queue interfaces do not support a fast merge operation, we use the well known *smaller into larger* technique. That is, for a merge we iterate over the smaller of the two sets and add the elements individually to the larger set. An element switches sets at most $O(\log n)$ times, each time into a set that is at least twice as large, and a switch takes $O(\log n)$. This sums up to $O(m \log^2 n)$ for all merges combined. Since the elements are moved individually, weight updates do not need lazy propagation but they are handled by an offset for each set that is applied when an element enters or leaves the set.

4.3.4 Our GGST Solver

The solver features three optimizations compared to the description in Section 4.2.3. First, no dummy edges are inserted. Instead a new path is started each time the root is reached. Second, we replace linked lists by dynamic arrays where possible. Exit lists, passive lists, and the growth path are only modified at the front, so an array can be used by saving them in reverse. Actually, the usage of passive lists as previously described requires arbitrary deletions and thus cross references for each edge to the position in the list. Our third optimization is to remove the need for cross references by simplifying the deletion patterns. The only time the algorithm deletes edges is during the contraction of a cycle[8]. Outgoing edges are deleted by clearing complete exit lists and mirroring the deletions across passive lists. Incoming multi-edges are deleted by clearing complete passive lists and mirroring the deletions across exit lists. We modify the two steps to make synchronization between exit and passive lists easier and restrict modifications to the front of the lists.

When outgoing edges of a cycle are deleted, some of these edges are self-loops and some point further down the growth path. Instead of mirroring the clearing of the exit lists by deleting corresponding entries from passive lists, we suggest to entirely skip the removal from the passive lists. This, of course, keeps invalid entries in the passive lists. However, a passive list is only read during a contraction to identify multi-edges into the cycle. At this time, the invalid entries point into a prefix of the path but at the time of deletion pointed down the path. Thus, they became self-loops which can be identified and skipped. Since a passive list is cleared after identification of self-loops, each invalid entry is seen only once.

We propose to implement the consolidation of multi-edges as follows. For each passive edge into the cycle, compare the first two edges in the exit list of the origin of the passive edge and delete the more expensive one. This "delete one of the first two edges" operation is done for each origin as often as this origin has passive edges into the cycle. Since this origin's exit list starts with an active edge pointing into the cycle, followed by all the passive edges into the cycle, the cheapest edge of this prefix will remain at the front of

[8]The original description by Gabow et al. has more deletions. We simplified the algorithm in this regard.

the exit list. Gabow et al. propose a similar strategy but delete either the first edge or the currently inspected passive edge (instead of the second in the exit list), which requires for each passive edge a way to obtain its handle in the exit list.

4.3.5 Alternative Reconstruction Method

The FELERIUS solver features an alternative method for reconstruction more closely related to the original idea of Edmonds' algorithm. Recall that Edmonds' algorithm contracts each cycle C and when picking an incoming edge into the contracted vertex, it replaces one of the cycle edges. That is, the edge into the contracted vertex corresponds to an original edge (u, v) and replaces the cycle edge incoming to v. The difficulty when adapting this to Tarjan's version is that endpoint indices of edges are not explicitly updated after each contraction. Thus, one has to deal with the possibility that the cycle vertices are contracted vertices representing previous cycles. In this case, v might be contained in a cycle vertex $v' \in C$ rather than being part of the cycle itself. This is, e.g., the case in Figure 4.2 where the edge g replaces the edge d. Stangl tackles this challenge as follows. Since Tarjan maintains an incoming edge for each vertex during the main algorithm, the reconstruction phase processes the cycles from last to first and performs the necessary replacements. When a cycle is processed, the edge (u, v) that was picked as incoming for this cycle can be found as the incoming edge to the vertex representing the cycle. To find the cycle edge it should replace, a persistent DSU is used to query the cycle vertex v' containing v at the time just before the cycle was contracted. To make the DSU persistent, Stangl drops path compression [Tar75] from the data structure which means each *find* call takes $O(\log n)$. However, the main algorithm as well as the reconstruction perform only $O(n)$ *find* calls thus leaving the total running time unchanged.

Another issue during implementation is that, after contracting a cycle, it is represented by one of its cycle vertices. The representative is chosen by the DSU among the cycle vertices according to the union-by-size strategy [Tar75]. The picked edge incoming to the contracted vertex thus overrides the cycle edge of this representative. The representative and the edge that was (mistakenly) replaced are saved during the main algorithm and restored in reconstruction just before the actual edge is determined that should be replaced.

4.4 Experiments

In this section we evaluate the solvers listed in Table 4.1. The solvers, data preparation scripts, plotting code, execution logs, and timing data are available in our public repository.

4.4.1 Setup and Datasets

The experiments were performed on a server with two 8-Core Intel Xeon™ Gold 6144 CPUs and 192 GB DDR4 memory on the openSUSE Leap 15.3 operating system. The implementations are written in C++ and adjusted to fit a common interface. The code was compiled with gcc version 10.3.0. Each run had a timeout of 30 minutes. We used a total of 656 networks from the following sources. The number of networks is in parentheses.

- **konect** (319). All directed networks smaller than 5GB from the KONECT project[9].
- **networkrepository** (75). A selection of sparse networks from the Network Repository project[10]. The project contains mostly undirected networks and does not label directed ones as such. We downloaded all networks (around 3000) and kept the ones that are labeled as directed in their respective file format.
- **girgs** (200). This data set contains geometric inhomogeneous random graphs. We used the efficient generator from Chapter 2 with default parameters except for n, deg, and seeds. We set $n = 10^4$ and average degrees from 50 to 2000 in steps of 100 with 10 networks per configuration. Edges are directed randomly.
- **antilemon** (5). A sparse family of networks crafted to be difficult for arboresence solvers. They require at least $n/2$ contractions with at least $n/2$ edges pointing into each contracted cycle. We generated networks with $n = 10^i$ for $i \in [2, 6]$.
- **fastestspeedrun** (47). Test cases of a programming task from the ICPC Northwestern Europe Regional Contest 2018[11]. They have up to 2500 vertices and are fully connected.
- **yosupo** (10). Test cases for the Directed MST problem on the Library Checker website. The networks are Erdős-Rényi graphs [ER59] with a random spanning tree from the root vertex as subgraph. Weights are sampled uniformly at random.

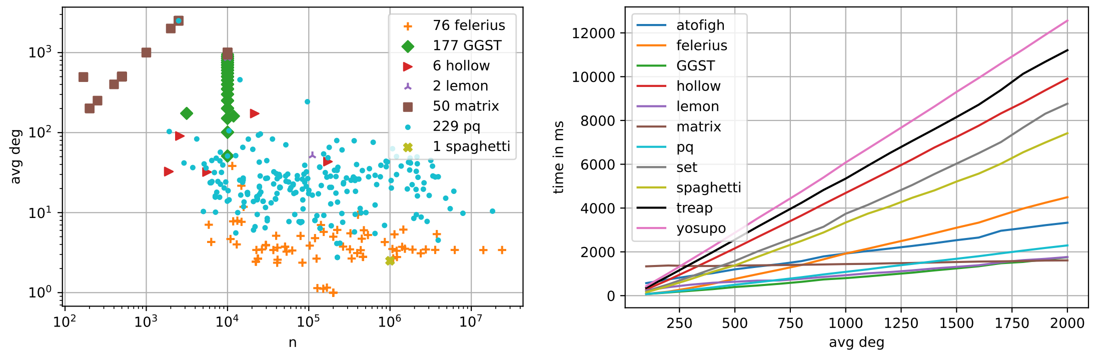
For unweighted networks, we sample integer weights uniformly at random. If an instance has no specified root, we restrict ourselves to the largest connected component and add a root vertex that connects to all original vertices with edges of weight infinity.

4.4.2 External Solver Integration

The LEMON solver does not compile with C++20 upwards because it uses allocator methods that were deprecated in C++17 and removed in C++20. We had to compile it separately from the other solvers. Furthermore, it performs reconstruction during the main algorithm. In Figure 4.4, its reconstruction time is the time to obtain the solution from their internal data structures.

The ATOFIGH solver contains a programming error in a radix sort subroutine where a right shift equal to the size of the left-hand operand type (`int` in our template instantiation) is performed. The C++ standard[12] states in Section 7.6.7 concerning shift operators "The behavior is undefined if the right operand is negative, or greater than or equal to the

[9] `http://konect.cc/`
[10] `https://networkrepository.com`
[11] `https://2018.nwerc.eu/`
[12] The standard must be purchased but a working draft is available at `http://www.open-std.org`



(a) Fastest solver per instance.

(b) Runtime scaling over growing GIRGs.

Figure 4.3: Left: For each instance the untied fastest solver if any. The legend includes the number of wins per solver. Right: The run time of the solvers on GIRGs with 10^4 vertices over growing density. Each data point is averaged over 10 GIRGs with the same density.

width of the promoted left operand" [ISO20]. Most compilers give a warning (if enabled) and default to 0 which actually works with the given implementation. Nevertheless, we fixed this error by changing $\leq$ to $<$ in the loop that iterates over the radix. The ATOFIGH solver also performs reconstruction during the main algorithm. In Figure 4.4, its reconstruction time is the time to obtain the solution from their internal data structures.

The YOSUPO solver uses `std::shared_ptr` for memory management. On very large instances this crashes due to a stack overflow caused by deep recursion in the destructor. On Linux machines, one can increase the stack limit to circumvent this problem which is what we do in our experiments.

The SPAGHETTI solver does not free allocated memory which gives it an advantage over the other solvers. We decided to keep the leak since a proper cleanup would require considerable changes to their code and performance. The SPAGHETTI solver is the only solver that does not support reconstruction.

4.4.3 General Performance

Figure 4.3a shows all instances with an untied fastest solver, i.e., a solver that is strictly faster than all others. The major reason for ties is that two or more solvers are faster than 1 ms which is the precision of our measurements. Overall, 115 instances are tied, 85 instances have at least two algorithms that solve the instance faster than 1 ms, 73 instances are solved in under 1 ms by at least six solvers, and 46 instances are solved in under 1 ms by all solvers.

On the untied instances, the PQ solver dominates with 229 wins, followed by GGST with 177, then FELERIUS with 76, and MATRIX with 50. Combined, these four solvers win more than 98 % of the untied instances. Moreover, there is a clear trend regarding the type of instance each solver is good at mirroring the theoretical complexities of the algorithms quite closely. The matrix-based Tarjan solver, MATRIX, is best for dense graphs, the heap-based Tarjan solvers, PQ and FELERIUS, are optimal for sparse graphs, and the GGST algorithm wins in between. For the sparse real-world instances, there is a clear cut between the PQ and FELERIUS solvers. The FELERIUS solver was specifically tuned to be fast on the yosupo instances which have barely more edges than vertices and thus wins on instances with average degree below 10. Furthermore, all but three of the 177 GGST wins are on GIRGs. We explicitly generated the GIRGs to fill the gap between the sparse real-world networks and the fully connected FASTESTSPEEDRUN instances. The most surprising result, however, is that the PQ solver using a binary heap performs exceptionally well although it should scale worse in the number of edges than the competitors by at least a logarithmic factor due to the missing merge operation. We identify three possible reasons for this behavior. First, a binary heap implementation is very efficient while the more complex logic of GGST and fewer cache efficient data structures of FELERIUS cause significant overhead. Second, realistic data is easy in the sense that the contractions, which are the theoretical bottleneck of the PQ solvers, occur not as often or involve less edges and vertices. Finally, realistic networks are sparse and thus $O(n \log n)$ becomes indistinguishable from $O(m \log n)$, which is the remaining complexity of the binary heap implementation when ignoring the cost for contractions. Therefore, on sparse networks with few contractions, the three solvers GGST, FELERIUS, and PQ all have a complexity of roughly $O(n \log n)$ and it comes down to implementation details like memory layout, cache efficiency, and the level of code optimization. For the same reason MATRIX beats GGST on very dense instances where both solvers have a complexity of $O(n^2)$.

4.4.4 Scaling Analysis

To examine the effect of density on solver performance we use the girgs data set. The GIRG model produces realistic networks regarding degree distribution, clustering, and distances that resemble the real-world networks from the NETWORKREPOSITORY and KONECT data sets. Figure 4.3b shows the results. As expected, the MATRIX solver is not affected by the number of edges. It starts out as the slowest solver but beats all the others by the time the degree reaches 2000. All other solvers exhibit an approximately linear scaling in the number of edges which emphasizes again that logarithmic factors are hardly noticeable for reasonably sized inputs. Most notably, this includes the $O(n^2)$ ATOFIGH and LEMON solvers. These solvers heavily depend on the fact that the instance structure is easy and needs few contractions involving few edges. The LEMON solver is the second fastest solver only slightly outperformed by GGST indicating that GIRGs are even easier to solve than the real-world networks from the other data sets. The reason for this could be the randomized edge direction for the GIRGs. Another interesting fact is that the five solvers that scale the worst with growing density are YOSUPO, TREAP, HOLLOW, SPAGHETTI, and SET. These five have in common that they use pointer-based heap data structures to manage the edges. The other solvers use indices into a preallocated pool (FELERIUS), don't have a heap element for every edge (GGST), or don't use a heap to manage edges (LEMON, ATOFIGH).

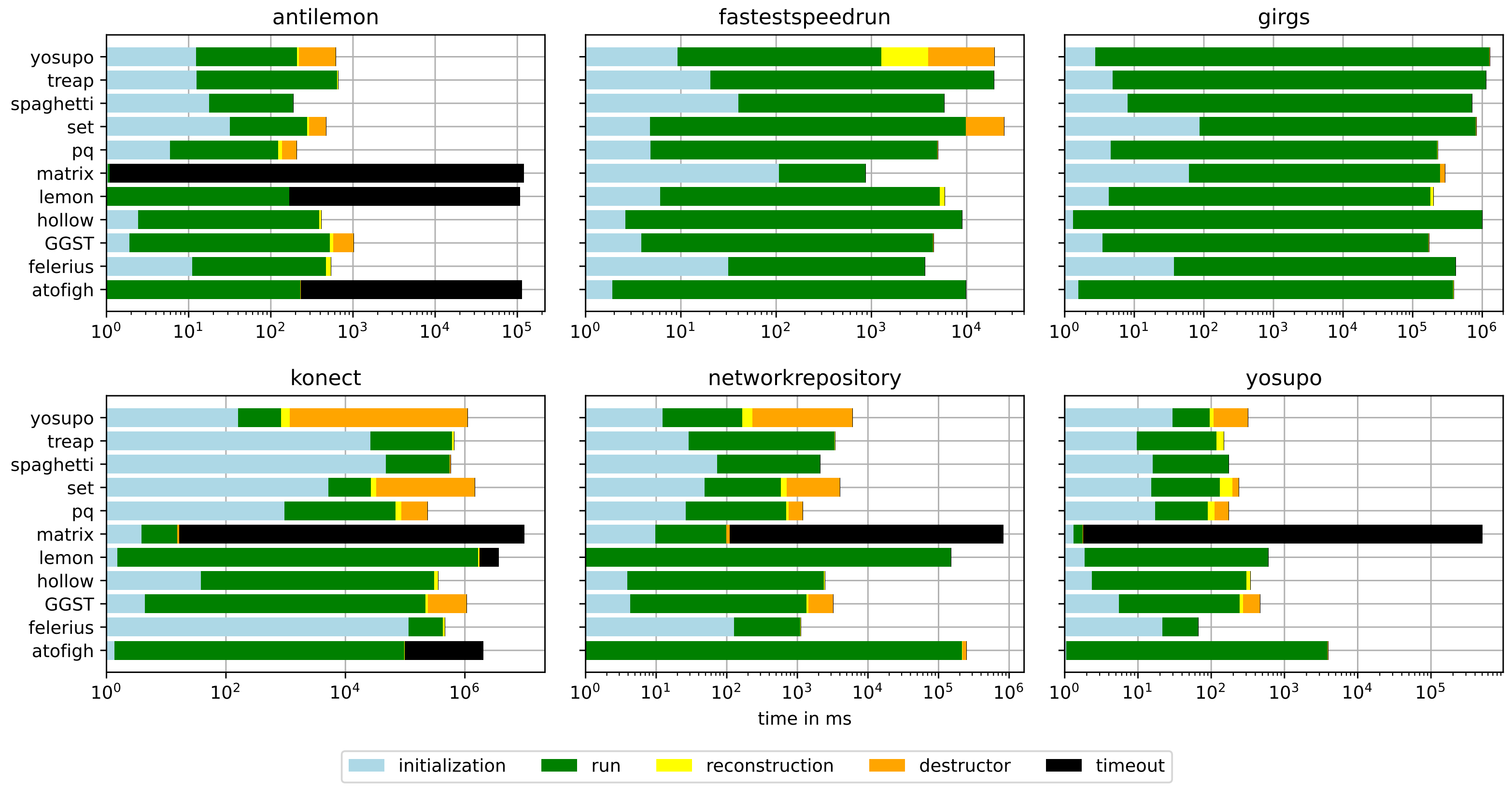


Figure 4.4: For each data set the summed run time over the contained instances per algorithm. The bars are divided into colored segments to show the fraction of time spent on each subroutine. For timeouts, all 30 minutes are counted as timeout no matter what was done in these 30 minutes. Note that the colored segments inside each bar are completely detached from the logarithmic x-axis.

4.4.5 Time Per Operation

Figure 4.4 shows the run times of the solvers divided into initialization, execution, reconstruction, and destructor subroutines as well as timeouts. The ATOFIGH solver crashed by exceeding the available memory on the largest ANTILEMON graph and 8 road networks from the KONECT data set, which are originally from the 9th DIMACS Implementation Challenge on shortest paths. These crashes are treated as timeout. The MATRIX solver is only executed on graphs with less than 10^5 vertices and treated as timeout otherwise. The LEMON solver timed out on three DIMACS graphs and the largest ANTILEMON graph.

On the real-world networks from the KONECT and NETWORKREPOSITORY data sets, the quadratic solvers perform much worse than the other algorithms. Of course, MATRIX cannot handle large graphs but also ATOFIGH and LEMON occasionally encounter a difficult instance. This overshadows their good performance on the many easy instances since we only consider the summed run time here. The LEMON solver performs well on the GIRG data set and the MATRIX solver dominates the fully connected FASTESTSPEEDRUN instances. Otherwise, the quadratic solvers are never among the fastest. In particular, these three solvers are more than two orders of magnitude slower than the FELERIUS solver on the NETWORKREPOSITORY graphs where the FELERIUS solver is the fastest on all instances. The ANTILEMON instances were crafted as worst-case instances for LEMON and ATOFIGH which is clearly visible in the results. On this data set, the PQ solver outperforms the others. Unsurprisingly, the FELERIUS solver performs best on the YOSUPO data which it was optimized for.

Before evaluating the run times of individual subroutines, we note that LEMON and ATOFIGH perform most of the initialization and reconstruction operations in the main phase of the algorithm while SPAGHETTI has neither reconstruction nor memory management. With that in mind, our experiments show that the reconstruction phase takes only a fraction of the run time independent of the solver or data set. Furthermore, the initialization, which includes allocating memory and building internal data structures, takes a considerable amount of time for all algorithms. The high initialization time can be explained by the fact that just inserting the edges into the heap data structures takes $O(m \log n)$ and as such is one of the theoretical bottlenecks of most implementations. There exist linear time constructions for some of the data structures (e.g. treaps, binary heaps, skew heaps) but for consistency across solvers, we build them by repeated insertions. Finally, the high destructor time of the YOSUPO solver is due to their use of `std::shared_ptr` instead of manual memory management.

4.5 Conclusion

In this paper we discussed the Tarjan and GGST versions of Edmonds' algorithm for the minimum spanning arborescence problem. We outlined existing solvers, provided our own implementations, and compared their practical performance. Our implementation of the GGST algorithm is the first public implementation and our description simplifies the original one in several aspects. Our experiments show that the compared solvers perform well on real-world data while scaling experiments suggest that realistic networks are substantially easier than worst-case instances. Even solvers with an $O(n^2)$ worst-case complexity often perform almost linear in the number of edges. However, they are not as consistent. They time out when the instance contains difficult structures, which occasionally happens even on real-world networks. Furthermore, we find that differences in complexity by logarithmic factors are mostly irrelevant in practice. Our $O(m \log^2 n)$ Tarjan implementation using a binary heap beats the other solvers on most real-world networks although our $O(n \log n + m)$ GGST implementation is two logarithmic factors faster asymptotically. This, again, emphasizes that real-world instances often do not force the worst case of an algorithm and complex logic and data structures can produce significant overhead. For future work, it would be interesting to examine what makes realistic instances easy and possibly show a better running time on a random model like hyperbolic random graphs similar to the result for Erdős-Rényi graphs.

5 A Branch-and-Bound Algorithm for Hitting Set

This chapter is based on joint work with Thomas Bläsius, Tobias Friedrich, and David Stangl [Blä+22c]. The idea for this work and the majority of the implementation originate in the master thesis by Stangl [Sta20]. The aim of the thesis was to transfer heuristics from SAT solving to other NP-hard problems. Specifically, a learning approach was to direct the branching decisions of the solver. The result was that the solver learned to branch on the vertices of highest degree, which is already an established heuristic. While interesting to see such a heuristic emerge without it being explicitly implemented, using just the heuristic instead of the learning approach was slightly faster.

5.1 Introduction

Hitting set naturally emerges from many problems appearing in various domains, e.g., transportation [Wei98], model-based diagnosis [Rei87], data profiling [Bir+20], or biology [ITK00]. Unfortunately, hitting set is NP-hard. In fact, it is among the first 21 NP-complete problems [Kar72].

Beyond its NP-completeness, there is a wide range of theoretic results on hitting set, including exact algorithms [SC10], approximation results [Sla97; CV07; DS14], parameterized algorithms [Abu10; Fer06], and parameterized approximation algorithms [BF12]. Moreover, several variants of the problem have been studied, e.g., weighted variants [Fer06], geometric variants, where the instance represents geometric objects [CV07], implicit hitting set, where the instance is not explicitly given but implicitly by an oracle that reveals sets not yet hit [Cha+11; MK13], and the enumeration variant, where one has to find all inclusion-wise minimal hitting sets instead of just the minimum [GV17].

Due to its importance for various applications, hitting set has also been studied from a practical perspective. A lot of engineering work has been dedicated to the above mentioned enumeration variant; see the survey by Gainer-Dewar and Vera-Licona [GV17] for an overview and the paper of Murakami and Uno [MU14] for the state-of-the-art algorithm. For the optimization problem of finding a minimum hitting set, there are results on heuristic algorithms, e.g., [Boj14; CKW10], as well as heavily parallelized brute-force approaches using GPUs [Car+17; Car+15].

Concerning clever algorithmic techniques for solving hitting set exactly, there is the seminal work of Weihe [Wei98] proposing two rules for data reduction that perform very well on instances coming from rail networks [Blä+19a]. More recently, Bevern and Smirnov [BS20] proposed alternative reduction rules for d-hitting set (restricting the size of each set to at most d) and evaluated them on instances coming from the cluster vertex deletion problem. Though reduction rules are a crucial component in designing efficient algorithms, one generally still needs an algorithm to solve the remaining instance. Concerning such an algorithm, the current state-of-the-art is somewhat unsatisfactory. In 2000, Caprara, Toth, and Fischetti [CTF00] did an exhaustive study of all prevalent solvers at the time and conclude: "This shows that the state-of-the-art general-purpose ILP solvers are competitive with the best exact algorithms for SCP[1] presented in the literature, and that their performance can sensibly be improved by an external preprocessing procedure." ([CTF00]) Later, de Kleer [Kle11] conducted an empirical study on the effect of Weihe's reduction rules [Wei98] in a simple branch-and-bound algorithm. However, the algorithm does not outperform a general-purpose ILP solver. To the best of our knowledge, using an ILP solver, potentially after preprocessing, remains the state-of-the-art to this day.

In this paper, we engineer and evaluate a branch-and-bound algorithm that beats this state-of-the-art. On our test set of 929 instances where the ILP solver reported a non-zero[2] running time, we reach a median speedup factor of more than 25. For three quarters of these instances, we have a speedup of more than one order of magnitude.

The basic building blocks of our branch-and-bound algorithm are bounds on the solution size and data reduction rules. They are described in Section 5.2. We note that most bounds and reduction rules we use have been considered before, either for hitting set or in a different context. For the different lower bounds we give a theoretical analysis that completely characterizes how they relate to each other; see Section 5.2.3. In Section 5.3 we specify our overall algorithm and provide details on how to efficiently implement it. Our evaluation in Section 5.4 is based on 4256 instances from different domains. Beyond the overall running time of our algorithm, we give a detailed evaluation of how much the different building blocks contribute to the final result. Finally, we discuss the performance impact of structural properties of the input instance in Section 5.5 using the generative network model geometric inhomogeneous random graphs [BKL19]. Our implementation is publicly available[3].

5.2 Basic Building Blocks

In this section we describe lower and upper bounds as well as reduction rules and introduce the hitting set problem and our notation.

5.2.1 Problem Definition

Formally, a *hypergraph* $\mathcal{F}$ is a set family over a *vertex set* V, i.e., every $F \in \mathcal{F}$ is a subset of V. We call these elements $F \in \mathcal{F}$ *hyperedges*. For brevity, we will refer to them as *edges*. The number of vertices in the hypergraph is $|V|$, and likewise the number of edges $|\mathcal{F}|$. Additionally, we use $\|\mathcal{F}\|$, called the hypergraph size, to refer to the sum of all edge

[1]SCP stands for "set cover problem", which is equivalent to the hitting set problem.
[2]Gurobi reports running times below 0.01 as 0.
[3]`https://github.com/Felerius/findminhs`

sizes, i.e., $\|\mathcal{F}\| = \sum_{F\in\mathcal{F}}|F|$. For a vertex $v \in V$, we denote the set of edges containing v as $\mathcal{F}(v)$. We call $\deg(v) = |\mathcal{F}(v)|$ the *degree* of v. Note that the sum of vertex degrees is equal to $\|\mathcal{F}\|$.

We say that a vertex *hits* an edge if it is contained in it. Based on this, we call a vertex subset $H \subseteq V$ a *hitting set* of $\mathcal{F}$ if all edges in $\mathcal{F}$ are hit by at least one vertex in H. Formally, $H \subseteq V$ is a hitting set of $\mathcal{F}$ if and only if $\forall F \in \mathcal{F} : H \cap F \neq \emptyset$. We call a hitting set *minimum* if no smaller hitting set for the same hypergraph exists. We refer to a hitting set as *minimal* if it contains no other hitting set as proper subset. The hitting set problem asks for a minimum hitting set of a given hypergraph.

5.2.2 Upper Bounds

For the upper bound, we use the simple greedy algorithm of repeatedly picking the vertex with the highest degree. This results in a $\log n$-approximation, works well in practice, and runs in linear time [GW97; Ski20]. We note that there are multiple LP-based upper (and also lower) bounds for which we refer to the overview by Caprara [CTF00].

5.2.3 Lower Bounds

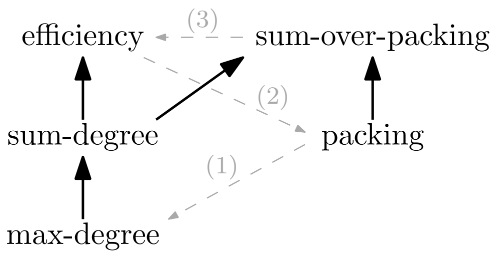
In contrast to upper bounds, good lower bounds are harder to achieve, but crucial for the pruning. Here we describe five lower bounds, some of which have been used for hitting set or other problems. Moreover, we prove a complete characterization of how the lower bounds relate to each other.

max-degree bound The max-degree bound uses that each vertex hits at most $d_{\max}$ many edges, where $d_{\max}$ is the highest vertex degree. Thus, at least $\left\lceil \frac{|\mathcal{F}|}{d_{\max}} \right\rceil$ vertices are required to hit all edges.

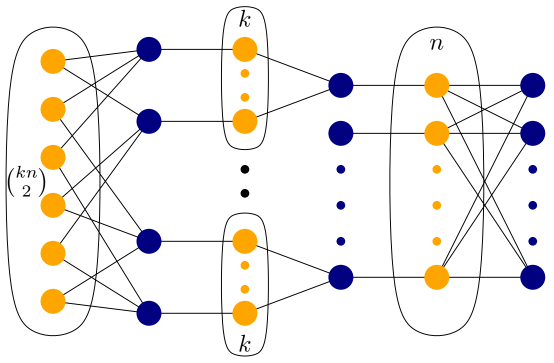
sum-degree bound Let $d_1, \ldots, d_n$ be the vertex degrees in descending order. Since vertices can only be chosen once, the max-degree bound can be improved to the smallest k for which $\sum_{i=1}^{k} d_i \geq |\mathcal{F}|$.

efficiency bound Consider any solution S. Let each vertex $v \in S$ charge its cost onto the edges it hits. That is, each edge $F \in \mathcal{F}(v)$ is charged $1/\deg(v)$ by v. The size of the solution can now be expressed as the sum of the cost of all edges, i.e., $|S| = \sum_{F\in\mathcal{F}} \sum_{v\in S\cap F} 1/\deg(v)$. The efficiency bound assumes the lowest cost for each edge individually, yielding $\left\lceil \sum_{F\in\mathcal{F}} \min_{v\in F} \frac{1}{\deg(v)} \right\rceil$ as a lower bound.

packing bound A set P of pairwise disjoint edges constitutes a lower bound, because each vertex appears in at most one of those edges. Thus at least $|P|$ vertices are required to cover them. Finding the best packing bound is actually an independent-set problem on the intersection graph of the edges $\mathcal{F}$. Using an independent set of conflicts as a lower bound is a known technique applied by recent solvers for other hard problems [Got+20a].



(a) Relations between lower bounds.



(b) Instance family represented as a bipartite graph.

Figure 5.1: Left: Hierarchy of lower bounds. A bold arrow from a to b means that a is dominated by b (Lemma 2). Dashed arrows are labeled as in Lemma 3 and indicate that there exists an instance where a is smaller than b. Note that for any pair of bounds, there is either a directed bold path (indicating dominance) or a directed path containing exactly one dashed edge (indicating non-dominance). Right: This instance family is used in Lemma 3. Hyperedges are blue and vertices are orange. The sum-over-packing bound is worse than efficiency for $k = 2$ and efficiency < packing for $k = 1$.

sum-over-packing Any packing P can be used to strengthen the sum-degree bound, as the packing requires to select $|P|$ vertices to cover the edges in P, which might not be the vertices of highest degree that the sum bound would use otherwise. In the remainder, we focus on how many edges of $\mathcal{F} \setminus P$ we cover. The vertices selected to cover P can cover at most $b_P = \sum_{F \in P} \max_{v \in F}(\deg(v) - 1)$ edges in $\mathcal{F} \setminus P$. If b_P is smaller than $|\mathcal{F} \setminus P|$, we have to pick additional vertices to cover the remaining $|\mathcal{F} \setminus P| - b_p$ edges. To this end, let $d_1, \ldots, d_n$ be the descending vertex degrees in $\mathcal{F} \setminus P$, excluding the vertex of highest degree of each edge in P (these vertices have already been selected and cannot be selected twice). With this, the improved sum bound is $|P| + k$ where $k \geq 0$ is the smallest number for which $\sum_{i=1}^{k} d_i \geq |\mathcal{F} \setminus P| - b_P$.

A lower bound a dominates another bound b, if $a \geq b$ for all problem instances. If a bound has multiple choices (e.g., the packing bounds), we consider the choice that leads to the highest bound. Two bounds are incomparable if neither dominates the other, i.e., there is an instance where $a < b$ and another instance where $a > b$. Figure 5.1a shows the relations between the lower bounds stated in the following lemmas.

Lemma 2. *The sum-over-packing dominates the sum-degree and the packing bound. The sum-degree bound dominates the max-degree bound and is dominated by the efficiency bound.*

Proof. We first show that the sum-over-packing bound dominates the sum-degree, and the packing bound. The packing bound cannot be larger because the sum-over-packing bound adds a non-negative number to it. Also the sum-over-packing bound over an empty packing is exactly the sum-degree bound.

The max-degree bound can be expressed as the smallest k for which $\sum_{i=1}^{k} d_1 \geq |\mathcal{F}|$ showing that the sum-degree bound dominates the max-degree bound.

It remains to show that the efficiency bound dominates the sum-degree bound. The sum-degree bound is given by the k highest degree vertices whose degrees sum up to just above $|\mathcal{F}|$. Let $v_1, \ldots, v_k$ be these vertices sorted descending by degree. We partition the edges of $\mathcal{F}$ into k sets $E_1, \ldots, E_k$ with the following two properties. First, $|E_i| = \deg(v_i)$ for $i < k$ and $1 \leq |E_k| \leq \deg(v_k)$. Second, the maximum vertex degree for every edge in E_i is at most $\deg(v_i)$. Such a partition can be achieved as follows. Assign to E_1 all edges containing v_1. For larger i, assign to E_i all edges that contain v_i that have not yet been assigned. Moreover, add further edges arbitrarily, until E_i contains $\deg(v_i)$ edges. The first property, concerning the sizes of E_i, clearly holds. For the second property, observe that in step i, all edges containing vertices $v_1, \ldots, v_{i-1}$ have already been assigned. Thus, all unassigned edges, and thereby all edges ending up in E_i, only contain vertices of degree at most $\deg(v_i)$.

With this partition, we get that the efficiency bound is larger than $k-1$, because

$$\begin{aligned} \sum_{F \in \mathcal{F}} \min_{v \in F} \frac{1}{\deg(v)} &= \sum_{i=1}^{k} \sum_{F \in E_i} \min_{v \in F} \frac{1}{\deg(v)} \\ &\geq \sum_{i=1}^{k} \sum_{F \in E_i} \frac{1}{\deg(v_i)} \\ &= k - 1 + \frac{|E_k|}{\deg(v_k)} > k - 1, \end{aligned}$$

and thus the rounded-up efficiency bound is at least as high as the sum-degree bound. □

Lemma 3. *The packing bound is incomparable with the max-degree, sum-degree, and efficiency bound. The efficiency bound is incomparable with the sum-over-packing bound.*

Proof. We give examples of instances where (1) packing < max-degree, (2) efficiency < packing, and (3) sum-over-packing < efficiency; see Figure 5.1a. With the domination relations from Lemma 2, the incomparability of the packing bound follows from the instances where packing <(1) max-degree ≤ sum-degree ≤ efficiency <(2) packing. Similarly, the second part of the lemma follows from the instances where efficiency <(2) packing ≤ sum-over-packing <(3) efficiency.

In the example for (1), there are three vertices a, b, c and three edges $\{a, b\}, \{a, c\}, \{b, c\}$. Every packing contains at most one edge. In contrast the max-degree bound is $\lceil 3/2 \rceil = 2$.

We give a parametrized example for (2) and (3). The construction is shown in Figure 5.1b. There are three types of edges. In the center, there are n disjoined edges, the *center edges*, that constitute a maximum packing. To the right there are n edges, the *right edges*, that share n vertices. To the left are $k \cdot n$ edges, the *left edges*, that pairwise share a vertex of degree two. Each of the center edges contains one of n high degree vertices shared by the right edges. Each center edge also contains k vertices of degree two that are also contained in one of the $k \cdot n$ left edges.

For $k = 1$, the efficiency bound is (for edges from left to right) $n/2 + n/(n+1) + n/(n+1) < n$ while the largest packing, e.g., the center edges, has size n. Thus, the efficiency bound can be smaller than the largest packing showing (2).

For $k = 2$ the efficiency bound is $2n/2 + n/(n+1) + n/(n+1) > n$. To prove (3), it remains to show that the sum-over-packing bound is at most n for each possible packing. Note that the argument must hold for *each* packing and not just maximum packings because a maximum packing does not necessarily lead to the highest sum-over-packing bound. If a packing includes a right edge, this prevents the inclusion of all other right edges and all center edges. If a packing includes a left edge, this prevents the inclusion of all other left edges and exactly one center edge. The only disjoined packings possible are either only center edges, one left edge and the rest center edges, or one left and one right edge. Thus, each packing P has size at most n and contains at least $|P| - 2$ center edges, which have vertices of degree $n + 1$.

Now we show that the sum-over-packing bound is either constant or limited by the packing size. For sufficiently large n and any packing S that is larger than six, the packing contains at least four center edges. The sum-over-packing bound adds nothing to the size of the packing because the maximum degrees of the vertices in four center edges add up to more than $|\mathcal{F}|$. That is, the sum-over-packing bound is $|S|$, because $\sum_{F \in S} \max_{v \in F} \deg(v) > 4n = |\mathcal{F}|$. For any packing S smaller than six, the remaining instance after deleting S and its max-degree vertices still contains $\Omega(n)$ nodes of degree at least n. Four of them suffice to obtain a degree sum larger than $|\mathcal{F}|$. Therefore the sum-over-packing bound is at most $|S| + 4 \leq 10$. In each case, the sum-over-packing bound is at most n, thus smaller than the efficiency bound. □

5.2.4 Reduction Rules

Our algorithm uses the following reduction rules. The domination rules were first described by Weihe [Wei98]. The costly discard rule is a standard technique in branch and bound but has, to the best of our knowledge, not yet been used for hitting set. The unit edge rule is widely known and stated, e.g., by Shi and Cai [SC10].

Unit Edge Rule. If there is an edge of size one, then pick the contained vertex.

Edge Domination Rule. If there are two edges e_1, e_2 with $e_1 \subseteq e_2$, then delete e_2.

Vertex Domination Rule. If there are two vertices v_1, v_2 such that $\mathcal{F}(v_1) \subseteq \mathcal{F}(v_2)$, then delete v_1.

Costly Discard Rule. If discarding a vertex raises the lower bound to or above the current upper bound, then pick this vertex.

We do not consider the complement of the costly discard rule (costly inclusion) because including a vertex cannot raise the packing lower bound by more than one. The rule could only take effect when the upper and lower bound differ by one, in which case the instance is almost solved anyway.

We note that Shi and Cai [SC10] proved that branch-and-bound runs in time $O(1.23801^n)$ when using the first three of the above reduction rules in addition to two rules based on edges of size two. We omitted these two rules from our solver, as they rarely take effect.

5.3 The Branch-and-Bound Algorithm

In this section we describe the structure and efficient implementation of our solver as outlined in Algorithm 5.1. In every step, our algorithm branches on the inclusion or exclusion of a vertex in the solution, thereby creating two new instances that are solved recursively. Before each branch, we apply the following two steps. First, an approximate solution for the current instance is found using a greedy algorithm. The best found solution so far represents the current upper bound and is globally maintained as a result of the algorithm. It is used to prune branches where no better solution can be achieved. Second, reduction rules are repeatedly applied until either no reduction is possible or a lower bound allows the current branch to be pruned.

Algorithm 5.1: Solve Recursively.

```
1 update upper bound // greedy
2 while reductions or pruning possible do
3     compute lower bounds
4     if lower bound $\geq$ upper bound then return
5     apply first applicable reduction
6 select branching vertex $v$ by highest degree
7 branch on choosing $v$ // inclusion branch
8 branch on discarding $v$ // exclusion branch
```

5.3.1 Operation Summary and Reduction Order

Initially we compute a greedy upper bound in time $O(||\mathcal{F}||)$ but do not repeat it in the reduction loop. Then, reductions and bounds are checked one after another (Algorithm 5.1, line 2-5). They are processed in ascending order by runtime to prevent expensive operations when possible. After one reduction is applied, the search starts from the top of the list again. In general, lower bound pruning happens before reductions. Although the max-degree bound is dominated by the efficiency bound, we include it due to the lower computational complexity. The order of lower bounds and reductions is as follows.

1. max-degree bound in $O(|V|)$
2. efficiency bound in $O(||\mathcal{F}||)$
3. packing bound in $O(||\mathcal{F}|| + |\mathcal{F}| \log |\mathcal{F}|)$

4. sum-over-packing bound in $O(||\mathcal{F}||)$
5. unit edge rule in $O(|\mathcal{F}|)$
6. costly discard rule with efficiency bound updates for all vertices in $O(||\mathcal{F}||)$
7. costly discard rule with packing updates for all vertices in $O(||\mathcal{F}|| + |\mathcal{F}| \log |\mathcal{F}|)$
8. costly discard rule with repack for 3 vertices in $O(||\mathcal{F}|| + |\mathcal{F}| \log |\mathcal{F}|)$
9. edge domination rule in $O(|\mathcal{F}| \cdot ||\mathcal{F}||)$
10. vertex domination rule in $O(|V| \cdot ||\mathcal{F}||)$

In the following, we discuss the branching strategy and implementation details of the instance representation, the bound computation, and the reduction rules.

5.3.2 Branching Strategy

As mentioned above, we branch on the inclusion or exclusion of a vertex. The remaining degrees of freedom are the vertex to branch on and the order in which the two branches are processed. We found the latter to be irrelevant while the former crucially affects search space and performance. Our solver always branches on the vertex with highest degree in the remaining instance and processes the inclusion branch first.

5.3.3 Instance Representation

During the algorithm the instance needs to be updated regularly. To avoid copying the instance, we maintain one data structure representing the current instance throughout the algorithm. There are three places in the algorithm where the instance is modified. First, the greedy algorithm iteratively deletes vertices. Second, the reduction rules reduce the instance. Third, when branching, a vertex is excluded (it is deleted) or is included (it and all its edges are deleted). In all cases, changes have to be rolled back appropriately.

To support these operations, we maintain the vertices of each edge and the edges of each vertex in sorted order at all times. Maintaining the order speeds up set-like operations, e.g., union of two edges, and is required by the reduction rules for edge and vertex domination.

For this, we implement a data structure called *ordered subset list* that manages a subset $S \subseteq \{s_1, \dots, s_n\}$ of n strictly-ordered objects $s_1 < s_2 < \dots < s_n$. Assuming the s_i are sorted in advance, it supports the following operations.

Name	Description	Time
init()	Initialize $S = \{s_1, \dots, s_n\}$	$O(n)$
del(i)	Delete s_i from S	$O(1)$
undo()	Undo last (not undone) deletion	$O(1)$
iter()	Traverse S in increasing order	$O(\|S\|)$
iterrev()	Traverse S in decreasing order	$O(\|S\|)$

This can be implemented by storing the subset S itself in a doubly-linked list. Additionally, we have an array A that points for each i to the list entry corresponding to s_i. When deleting an element from S, its list entry can be found in constant time via A. It is removed from the list, but the list item itself remains in memory and A keeps the pointer to it. To allow for later reinsertion, we maintain a stack of indices of deleted list entries. As the list entry itself has not been modified at the time of deletion, its previous and next entries are intact and thus we can reinsert it into the list in constant time in the position it was before its deletion.

5.3.4 Upper Bound Computation

The greedy algorithm picks the highest degree vertex and deletes it and its edges from the instance until all edges are hit. Since modifications, i.e., deleting edges and vertices, are done in time linear in the number of changes (see Section 5.3.3), this totals to at most linear time until a hitting set is found. Finding the vertex with highest degree in each step is done with a bucket heap [Cor+09; Ski20] that stores the vertex degrees. The data structure allows constant time operations due to the limited range of the stored values. Since degrees are only lowered during the procedure and the total vertex degree is $||\mathcal{F}||$ the greedy algorithm takes linear time in the size of the instance.

5.3.5 Packing Bound Computation

Finding a maximum packing of disjoined edges is an independent set problem and thus computationally expensive. Recent solvers for the quasi-threshold editing and cluster editing problem, which use the same idea of packing conflicts, apply the min-degree heuristic to find a good packing [Got+20a; HH15]. In our context, the degree in the conflict graph of an edge F from the original instance would be the number of other edges that share at least one vertex with F. We approximate this and sort all edges by $\sum_{v \in F} \deg(v)$ in ascending order. Then, we go through the edges and add the current edge to the packing if possible. When adding an edge to the packing, each contained vertex is marked. An edge F can be added if all contained vertices are unmarked, which can be checked in $|F|$. In total, the initial packing is computed in $O(||\mathcal{F}|| + |\mathcal{F}| \log |\mathcal{F}|)$.

We implemented the 2-improvement heuristic for independent set to grow the packing [ARW12]. The heuristic is a local search that repeatedly tries to replace an element from the packing with two new ones. Although this technique is effective [Got+20a], we found it to be too slow and too rarely applicable to justify the high computational cost (see Figure 5.6). Our implementation runs in $O(|P| \cdot ||\mathcal{F}||)$ per improvement where P is the current packing.

5.3.6 Efficient Costly Discard Rule

The costly discard rule states that a vertex must be picked if discarding it raises some lower bound to or above the current upper bound. That is, if we were to branch on that vertex, the exclusion branch would be pruned immediately. The rule has two degrees of freedom:

first, the vertex it is applied to and, second, the lower bound that is used. For maximum effectiveness of the reduction, we would like to check the rule for all vertices and lower bounds. However, computing all lower bounds from scratch $|V|$ times is computationally expensive. We restrict it to the efficiency and packing bound. In the following, we discuss how to compute these bounds efficiently for all vertices at once.

Costly Discard with Efficiency Bound. For the efficiency bound, checking the costly discard rule for all vertices at once can be done in $O(||\mathcal{F}||)$ as follows. First, the efficiency bound is computed for the current instance. While doing so, for each edge the two vertices with highest degree are saved. When a vertex v would be discarded from the instance, only edges v is contained in can change their contribution to the bound. Such an edge F changes the contribution $\min_{u \in F} 1/\deg(u)$ only if v was the vertex with highest degree in F. In this case the contribution depends on the vertex with second highest degree in F, which was identified earlier. In total, discarding v changes the contribution of at most $\deg(v)$ edges that can be updated in constant time each. Over all vertices this sums up to $||\mathcal{F}||$.

Costly Discard with Packing Bound. For the packing bound, a similar approach of dynamically updating a packing bound $|V|$ times can be used to check the rule for all vertices and constitutes item 7 in Section 5.3.1. Discarding a vertex v removes it from all edges. The edges that are relevant are those that intersect the union of the current packing exactly in vertex v. That is, they could now be included in the packing after v's removal. We say that such an edge is blocked by vertex v. Each edge is blocked by at most one vertex. After the initial packing is constructed, we create for each vertex v a list of edges that are blocked by this vertex and sort each list individually by the highest degree of a contained vertex (excluding v). These lists are found and sorted in $O(||\mathcal{F}|| + |\mathcal{F}| \log |\mathcal{F}|)$. When checking if a vertex qualifies for the costly discard rule with the packing bound, we traverse the list of blocked edges for this vertex and greedily add them to the packing if possible. After the rule is checked, we remove the added edges to restore the initial packing. Since each edge F is in at most one list and can be added to the packing in time $O(|F|)$, the costly discard rule can be checked for all vertices in time $O(||\mathcal{F}||)$ when using the lists. The creation of the lists dominates the running time with $O(||\mathcal{F}|| + |\mathcal{F}| \log |\mathcal{F}|)$, which is the same as the time it takes to compute the initial packing.

Unfortunately, the updated packings are worse than if they were computed from scratch. Therefore, we additionally choose the c vertices of highest degree, for which we check the costly discard rule with a completely new packing each (see item 8 in Section 5.3.1). Our experiments in Section 5.4.5 suggest that $c = 3$ is a reasonable choice.

5.3.7 Efficient Domination Rules

The domination rules can be checked naively by comparing each set with all others to find inclusions. This implies a running time of $O(|V| \cdot ||\mathcal{F}||)$ and $O(|\mathcal{F}| \cdot ||\mathcal{F}||)$ which is quadratic in the number of edges or vertices, respectively. In fact, under the strong exponential time hypothesis the reduction cannot be done in less than quadratic time in the worst case [BCH16]. In practice, however, the sub- and superset reductions can be sped up significantly using *set tries*, a data structure described by Savnik [Sav13].

A set trie manages a collection $\mathcal{T}$ of sets over $[n]$ and supports the following operations which require that the given sets are sorted and can be traversed in linear time.

Name	Description	Time
add(S)	add S to $\mathcal{T}$	$O(\lvert S\rvert)$
hasSubset()	does $\mathcal{T}$ contain a subset of S	$O(\lvert S\rvert + \lVert\mathcal{T}\rVert)$
hasSuperset()	does $\mathcal{T}$ contain a superset of S	$O(\lvert S\rvert + \lVert\mathcal{T}\rVert)$

For the dominated edge rule we create an empty set trie and iterate through the edges in increasing order of their size. For each edge, we check whether a subset of it exists in the set trie. If so, the edge is dominated. Otherwise the edge is added to the set trie. Note that this process can be continued after the first dominated edge to find all of them.

For the dominated vertex rule, the process is similar. An empty set trie is created that stores sets of edges. Then, vertices are iterated in decreasing order of their degree. Recall that $F(v)$ is the set of edges containing the vertex v. If the set trie contains a superset of $F(v)$, the vertex v is dominated. Otherwise, $F(v)$ is added to the set trie.

5.4 Evaluation on Public Hitting-Set Instances

In this section, we evaluate our branch-and-bound algorithm experimentally on a range of instances used in previous publications. First, in Section 5.4.2, we compare its run time to the state-of-the-art ILP-solver Gurobi [Gur21] and analyze how much time is spent in which part of the algorithm. Moreover, we examine the contributing factors for the size of the search space during a run. The next three sections evaluate the used techniques as well as substantiate our design decisions by providing experimental grounds to argue in favor of our choices regarding the solver configuration. We investigate details regarding the performance and effectiveness of lower bounds in Section 5.4.3, upper bounds in Section 5.4.4, and reduction rules in Section 5.4.5. Specifically, we evaluate the set and order of used lower bounds, the number of checked vertices in the costly discard repack rule, the frequency of greedy invocations, and the order in which reductions are applied.

5.4.1 Experimental Setup

Our implementation is written in the Rust programming language and is available in our public GitHub repository[4] along with all datasets, logs, run results, and evaluation scripts. All auxiliary packages, including the versions used, are listed in the repository as part of the Cargo project format. For the evaluation, we used version 1.53.0 of the Rust compiler with link-time optimization enabled. The experiments were run on a Gigabyte R282-Z93

[4]`https://github.com/Felerius/findminhs`

(rev. 100) server at 2.6GHz base speed with 1024GB DDR4 (3200MHz) memory. Runs had a timeout of 24h.

We use instances from four sources.

`UCC` `[Bir+20]` contains 134 instances, two for each of 67 databases. In the first type of instance, the hitting sets correspond to the unique column combinations of the database. The second type of instances are the transversal hypergraphs of the first type.

`CVD` `[BS20]` contains cluster vertex deletion instances, derived from weighted graphs of protein similarities [Rah+07; Böc+07]. In the reduction step from weighted graphs to unweighted graphs, we use all edges with non-negative weights. This is consistent with the code linked in the paper of Bevern and Smirnov [BS20], but differs from the statement in the paper itself, which only uses edges with positive weights. Like the authors, we restict us to hypergraphs with at most 10^6 edges, resulting in a total of 3952 instances.

`EN1` `[MU14]` contains 159 instances that were previously used to evaluate algorithms for enumerating minimal hitting sets. These contain several classes of instances, including real-world and generated instances. The original data set contains 172 instances of which we omitted 13 whose size $||\mathcal{F}||$ exceeds $3 \cdot 10^7$, as the RAM required to run experiments on them proved to be prohibitive.

`EN2` `[GV17]` contains eleven additional instances that have been used to evaluate enumeration algorithms. Five of them are derived from metabolic reaction networks and six from interventions in cell signaling networks.

We distinguish between the randomly generated instances (`rnd`) from the `EN1` dataset and application specific instances (`appl`) due to their different structure. The instances displayed in the various plots are filtered depending on the context. Figure 5.2 includes all 4256 instances. Subsequent plots are restricted to the 136 instances (58 `rnd`, 78 `appl`) that finish in 24 hours (excludes 6) and are non-trivial (excludes 4114), that is, instances where our solver runs at least one second in its default configuration. For experiments that compare different configurations, only the instances finishing in *all* configurations are used. Effected by this is Figure 5.6 where three instances were dropped due to timeout in some configuration. Additionally, ten instances where dropped in Figure 5.9b because they had no forced vertices and two instances where dropped in Figure 5.5b because they never had a branch pruned due to bounds.

5.4.2 Runtime Performance and Search Space

Figure 5.2 shows the run time of our solver in comparison with Gurobi. Gurobi is at version 9.1.2, restricted to a single thread, and without a memory restriction. We note that there are instances where Gurobi uses almost 50GB of memory. Following Caprara [CTF00], instances were reduced with the domination rules before running Gurobi on them. The

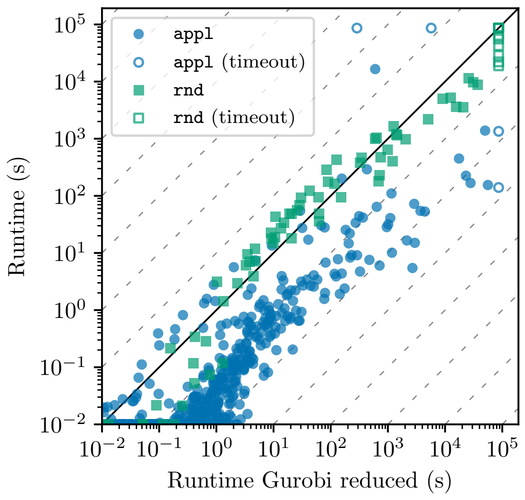


Figure 5.2: Run time of our solver compared to Gurobi with preprocessing. Times are rounded up to 0.01s.

reduction process is included in the reported run times. Preliminary experiments showed this to be faster than running the ILP alone.

Our solver is significantly faster than Gurobi on non-random instances; on three quarter of non-random instances at least one order of magnitude. Contrasting this, there are only three instances (random and non-random) where Gurobi is faster by more than a factor of 4. Run times for random instances are competitive. Gurobi is approximately 1.5 times faster on smaller instances while we are consistently faster on instances that take more than 30 minutes to solve. In total, there are 8 instances that finish in the timeout only for our solver. There are 2 instances that finish just for Gurobi.

Figure 5.3 shows the fraction of run time that is spent in each step of the algorithm. As expected from the asymptotic considerations in Section 5.3, the domination rules dominate the run time although they are executed last and thus avoided when possible. Random instances spent most time in edge domination since they have many edges and few vertices. Non-random instances spend most time in vertex domination. Still, when taking both classes of instances together, the total time spent is spread over different subroutines and, in the median, no individual task takes more than 20% of the total time. Greedy, packing lower bound, and the costly discard repack reduction rule show comparable times. Although, the repack reduction essentially computes three packings, it is processed later in the loop than the packing lower bound, which explains why the rule does not take three times as much time as the packing. Finally note that the column for *other* is vanishingly small. It includes, e.g., the instance manipulation and rollbacks as well as logging, timing, and branching.

After evaluating the runtime in general, the question poses itself, what key factors contribute to the performance and which instances are the most difficult. For this, we

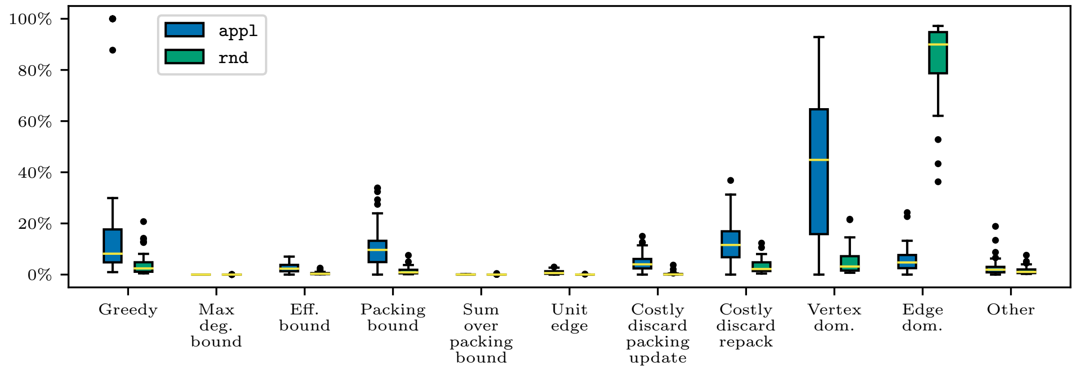


Figure 5.3: For each instance, the run time share of each operation was measured relative to the total run time of the instance. Note that the time for efficiency costly discard is included in the efficiency bound time.

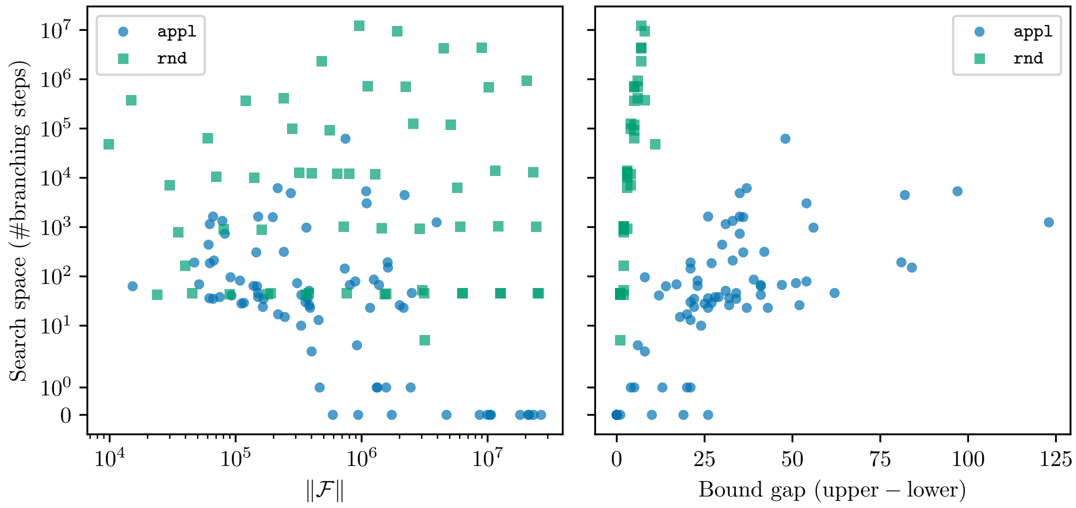
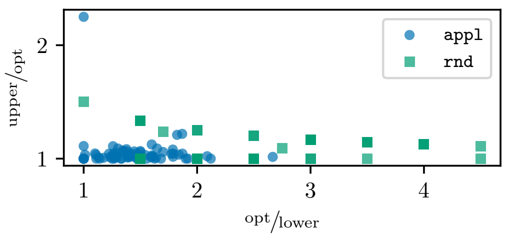
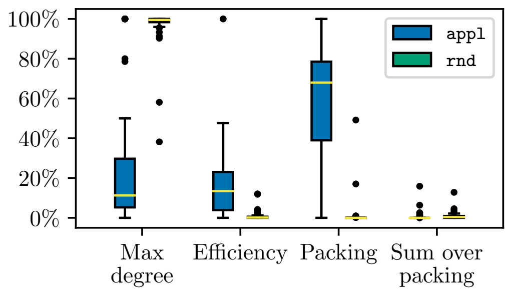


Figure 5.4: Search space compared to instance size (left) and difference between upper and lower bound (right). Each instance represents one sample.



(a) Quality of upper and lower bounds.



(b) Number of prunes by bound.

Figure 5.5: Left: For each instance, the initial lower bound relative to the optimum on the x-axis and the upper bound relative to the optimum on the y-axis. Right: For each instance, the breaks from the reduction loop split by the responsible bound. Values are relative to the total number of breaks for the instance.

measure the search space, that is the number of branching decisions of the solver, instead of its run time. Figure 5.4 shows the size of the search space depending on the instance size and the difference between the initial upper and lower bound. The lack of samples with low search space and instance size is an artifact due to the exclusion of instances that are solved in less than a second. There appears to be no clear correlation to indicate that instance size reflect difficulty. On the other hand, the bound gap is usually a good estimate for the difficulty of an instance to a given branch-and-bound solver, because the solver lowers this gap during execution and is finished if the difference reaches zero. The random instances exhibit a distinct exponential growth in search space with growing gap. Overall, the gap between upper and lower bound is the key factor that determines the performance of the algorithm.

To find out which bound is responsible for the gap, Figure 5.5a explores the overall quality of lower and upper bounds compared to the optimum. The initial upper bounds are already close to the optimum with all but three being less than 1.3 times the optimum. Lower bounds are spread out more. Some instances have lower bounds that are not even a third of the optimum. Interestingly, the instance with with worst upper bound has a perfect lower bound. In conclusion, this shows that lower bounds as well as upper bounds can be improved but upper bounds are closer to the optimum already. Thus, a significant performance improvement could be gained from better lower bounds.

5.4.3 Lower Bound Effectiveness

Figure 5.5b counts how often each bound was responsible for pruning a branch in the search tree. For random instances, the max-degree bound is sufficient and is responsible for almost all prunes. Application instances make use of several bounds. Max-degree still helps but, excluding a few instances, accounts for less than 30%. The efficiency and the packing bounds then catch branches that are missed by the previous bounds with the

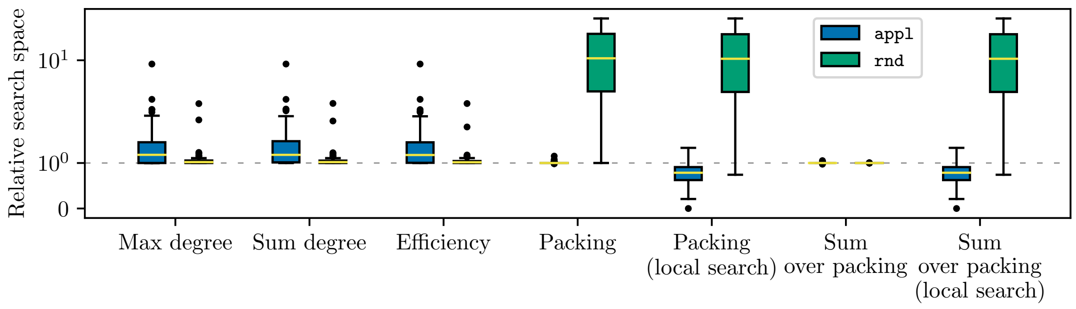


Figure 5.6: Search space when only using a certain bound, relative to default settings. Note the logarithmic y-axis with special handling for zero.

packing bound being significantly more successful than the efficiency bound despite it being run after it.

For the efficiency bound, the outlier at 100% is due to an instance with a trivial search space of one node that is pruned by the efficiency bound. Still, there are a few instances where the max-degree bound fails often while the efficiency bound is responsible for 40% to 60% of all prunes.

There are even more instances which almost exclusively depend on the packing bound. This can be explained due to the packing bound representing a different approach to obtain the lower bound than the max-degree or efficiency bound and thus performs well on a different kind of instances. The max-degree bound performs well if the high degree vertices do not share many edges, i.e., if the instance can be covered by selecting few high-degree vertices. On the other hand, many edges containing multiple high-degree vertices makes it possible to have many edges containing only vertices of lower degree, which facilitates large packings.

Regarding the last bound, note that the sum-over-packing bound rarely applies. However, Figure 5.3 shows that its run time is negligible since the previously computed packing is reused. For some instances the bound actually prunes a significant number of branches.

With the question answered to what extent our chosen lower bounds contribute to pruning, it remains to show that it is their combination and not one bound alone that is responsible for the overall performance. Figure 5.6 shows the relative search space when using only one bound compared to using our default configuration for the solver. Max-degree, sum-degree, and efficiency all behave similar, that is worse than with all bounds. On the other hand, for random instances only using the packing bound leads to significantly higher search space. These instances, however, are solved easily when using the sum-over-packing instead of the packing alone. In fact, using only sum-over-packing is almost always as good as using the combination of bounds. Nonetheless, it still makes sense to use the other bounds before: The packing bound is free as we have to compute a packing anyway to apply sum-over-packing. Moreover, the simpler bounds can be computed more quickly than a packing (recall Figure 5.3) but are often sufficient as can

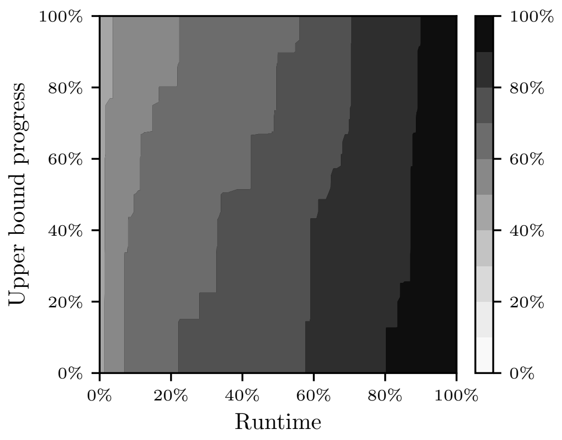


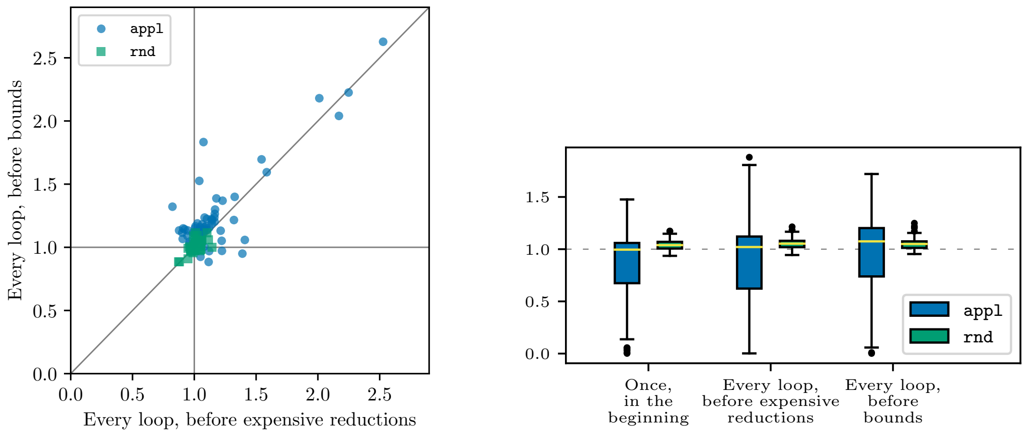
Figure 5.7: For a given time in the solving process and an amount of progress from the initial upper bound towards opt, the number of instances that have reached this progress at that time.

be seen in Figure 5.5b.

As mentioned in Section 5.3.5, we implemented versions of the packing with local search and included them in Figure 5.6. In the following we consider the influence of adding local search to packing or sum-over-packing. For application instances, adding the local search to the packer or sum-over-packing bound results in a slight improvement over the combination of all other bounds. However, preliminary testing showed local search to be too computationally expensive to justify the reduction in search space it yields. On random instances, adding local search to packing does not change the search space, while adding it to sum-over-packing surprisingly increases the search space. Further investigation revealed that a packing that was augmented with local search is enlarged to the point that taking the highest degree vertex of each edge in the packing constitutes for enough total degree to cover the whole instance. In this case, the sum-over-packing bound is equal to the packing bound.

5.4.4 Upper Bound Effectiveness

The greedy upper bound is used to initialize and, during a run, improve the best solution found so far. Figure 5.7 shows the progression and gradual lowering of the upper bound during execution. Each individual instance starts in the lower left corner at (0,0) and progresses to the upper right corner at (100,100). Under the assumption that this progression is linear we would get white above and black below the main diagonal. However, the plot can be better described as growing darker from left to right. This verticality means that during a run, upper bounds progress is often achieved all at once. Also the upper bound is surprisingly good with more than 40% of the instances already having a perfect upper bound to start with. After 25% of the run time the optimum has



(a) Runtime of greedy modes relative to each other.

(b) Runtime of greedy modes relative to no greedy.

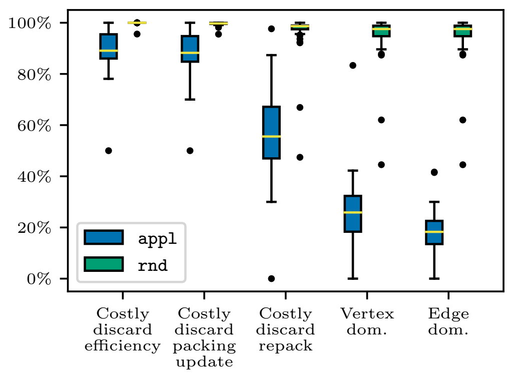
Figure 5.8: Left: Run times of two non-default greedy modes relative to the run time when using greedy once per node in the search tree. Each instance is a data point. Right: Relative run time for all three greedy modes compared to not using greedy upper bounds.

been found (100% progress) for more than 60% of instances. The remaining 75% of the run time are used for proving its optimality.

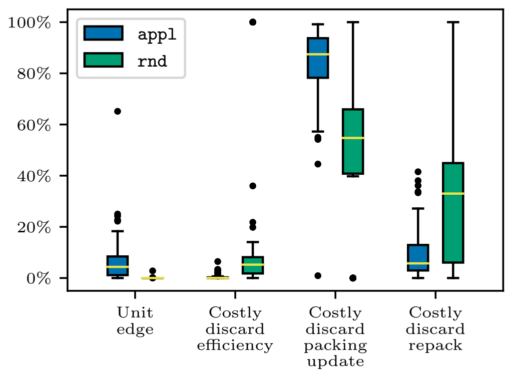
Still, the solver could be implemented without greedy at all. The upper bound would then be initialized to contain all vertices in V and updated when the branching reaches a hitting set. Running the greedy subroutine has the benefit that it helps to find solutions before reaching the associated leaf in the search tree and thus facilitates earlier pruning. In the following, we compare four different frequencies of running greedy to recompute the upper bound. The alternatives are to not use greedy at all, run greedy once before the reduction loop (which is what we do in the final configuration of our algorithm), run greedy every iteration of the loop (i.e., as item zero in Section 5.3.1), or to run it every loop just before the expensive reductions (i.e., between items 7 and 8 in Section 5.3.1).

Figure 5.8a compares the latter three alternatives. The baseline in the plot (and default configuration for the solver) is to run it once before the loop. The axes show the relative run time compared to this baseline. A point in the lower left quadrant means that the baseline is the worst out of the three; a point in the upper right quadrant that it is best, while the rest ranks it in the middle. Samples are concentrated in the center but slightly tilted to the upper right. Additionally, there are no outliers that favor running greedy every loop while there are outliers that heavily slow down when deviating from our baseline.

Figure 5.8b shows run times of all alternatives that use greedy compared to not using greedy at all, that is, initializing the upper bound to V and find better solutions only in leaf nodes while branching. Surprisingly, the median favors not using greedy at all



(a) Number of reduction iterations that reach a given point.



(b) Number of forced vertices found.

Figure 5.9: Left: For each instance, the number of reduction loop runs that reach a given reduction. Values are relative to all runs that do not break due to bounds. The forced vertex rule is excluded as it is the first and thus always reached. Right: For each instance, the number of found forced vertices by reduction rule. Values are relative to the total number of forced vertices for the instance.

but the outliers are heavily in favor of using greedy. Running greedy once before the reduction loop is never slower than 1.5 times the run time of no greedy, 1.02 times slower in the median, and up to 600 times faster at best, which makes it a good choice for the final configuration of our algorithm. Note that the benefits of greedy are restricted to application instances. On random instances there are no heavy outliers in favor of any configuration.

5.4.5 Reduction Effectiveness

Before we establish how effective each specific reduction is, we first investigate how often each reduction is actually reached in the reduction loop as shown in Figure 5.9a. Differences between two adjacent reductions express the success rate of the left one. Random instances, again, behave completely different than application specific instances. They almost always execute all reductions, because all reductions before the last one are unsuccessful. For application instances, all rules contribute somewhat. Although the domination rules have the highest run time share (recall Figure 5.3) they are only executed in 20–30% of loop iterations for most instances. The most frequent end to an iteration are a successful costly discard rule with packing updates or repacking. These rules are reached in more than 80% of iterations for most instances and the next rule (vertex domination) is checked 30% of the time in the median. Note that an iteration only ends in the repack step if the costly discard rule was unsuccessfully checked with a packing update before succeeding through a repack, giving evidence to the usefulness of repacking.

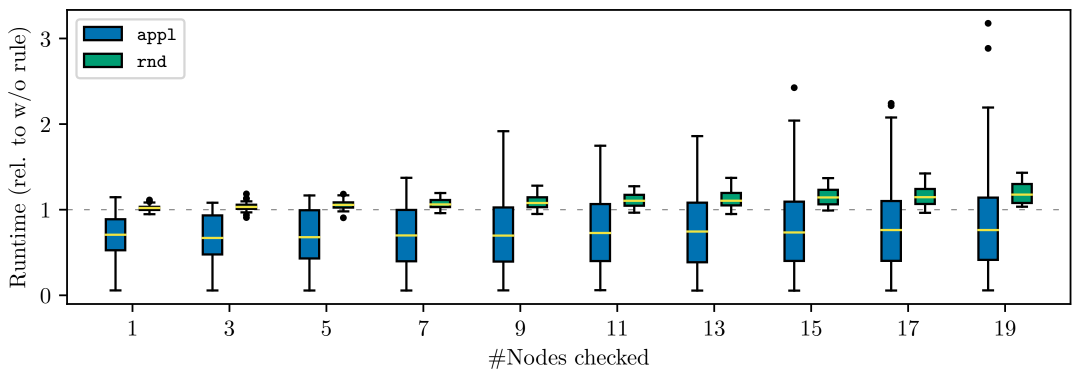
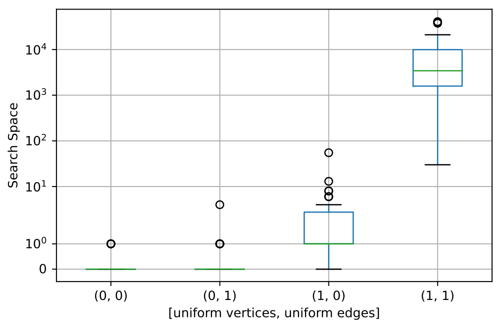


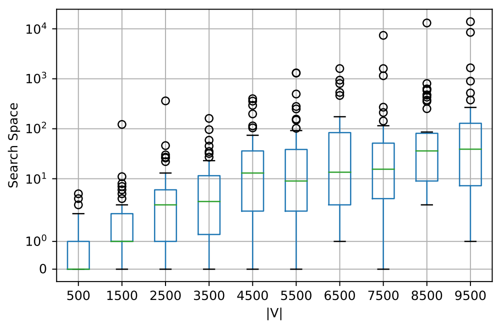
Figure 5.10: Run time when using different settings for the costly discard packing from scratch rule. All values are relative to the run time when that rule is disabled.

Figure 5.9b compares the effectiveness of the reductions that force a vertex to be included in the solution. The effectiveness is measured by the number of forced vertices. The unit edge rule and the costly discard rule with the efficiency bound catch almost no vertices compared to the two packing rules. Since they are very easy to compute and find a good amount of application on some instances they are nonetheless worth trying. Above 80% of forced vertices in application instances are found with the costly discard packing update rule. This constitutes no contradiction to Figure 5.9a because, although the repack rule is applicable as often, the packing update rule can find multiple forced vertices at once. The fact that the repack rule still finds a large amount of all forced vertices while being applied only after packing update failed to find anything, again, emphasizes that the updated packings are not as good as the packings that are constructed from scratch. The results for random instances are less expressive because only a small number of reductions are applicable, as established in the previous paragraph. Nevertheless, they follow the same trend as the other instances but rate packing update lower and packing repack higher.

One degree of freedom for the repacking is the number of vertices for which we apply it. Recall from Section 5.3.1 that we repack for the $c = 3$ vertices of maximum degree in our final configuration. Figure 5.10 shows the run time for different values of c relative to the run time when not repacking. One can make two main observations. First, for random instances, the cost of repacking outweighs the gain leading to slightly increasing median run times for increasing c. Second, repacking helps significantly on non-random instances but there is no additional gain in repacking more than three times, which has the lowest median. Thus, by using $c = 3$, we obtain a good balance between increasing run time for random instances only slightly while obtaining big speedups for some application instances.



(a) Search space by distribution combination.



(b) Search space by number of vertices.

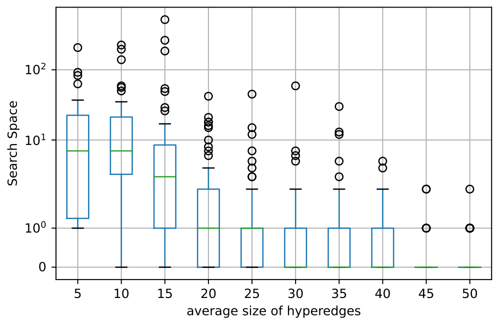
Figure 5.11: Left: Search space of the solver when choosing a homogeneous or heterogeneous distribution for vertex degrees and hyperedge sizes, respectively. Right: Search space of the solver for growing number of vertices and $|\mathcal{F}| = 5000$. Vertex degrees are heterogeneous and hyperedge sizes homogeneous.

5.5 Evaluation on Geometric Inhomogeneous Random Graphs

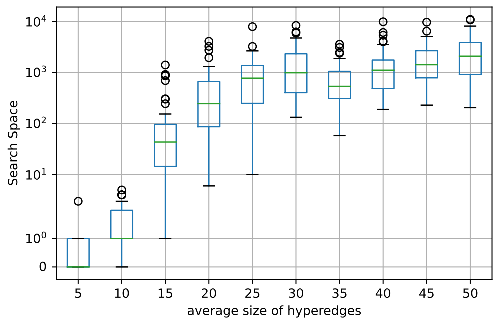
In this section, we discuss the performance impact of different structural properties of the input instance. To obtain realistic instances with controllable structural properties, we use geometric inhomogeneous random graphs (GIRGs) [BKL19]. This lets us isolate the effect of a single structural property on the algorithm performance while keeping other properties of the instance fixed. We discuss — in this order — the degree distribution, the hyperedge size distribution, the ratio of vertices to hyperedges, the average hyperedge size, the instance size, as well as the amount of locality.

We use the efficient generator introduced in Chapter 2 and modify it to generate bipartite graphs, which can be interpreted as hitting set instances. This approach comes with two challenges. First, the generator may generate trivial instances due to empty hyperedges, since their size is controlled in expectation. Second, the estimation method for the model parameter that controls average degree does not work for bipartite graphs. We circumvent the latter problem by using an exponential search followed by a binary search to get a graph with the desired density. Then we discard empty hyperedges. We monitor the amount of discarded edges to (manually) guarantee that only a very small fraction is discarded. We set the internal parameters of the GIRG model to temperature 0, dimension 2, and a power-law exponent of 2.8. Each box of the box plots summarizes 50 graphs generated with the same set of parameters but different seeds. The modified generator, plotting code, raw results, as well as execution logs are publicly available[5].

[5]`https://github.com/chistopher/sat-girgs`



(a) Search space for heterogeneous vertex degree.



(b) Search space for heterogeneous edge size.

Figure 5.12: Search space by hyperedge size. In (a) $|V| = 5000, |\mathcal{F}| = 5000$. In (b) $|V| = 200, |\mathcal{F}| = 200$.

5.5.1 Degree Distributions

Vertex degrees and hyperedge sizes are two degrees of freedom when generating instances. Viewed as a bipartite graph, these correspond to the degree distributions of the two partitions. While there are numerous reasonable possibilities for the distributions, we focus on homogeneity vs. heterogeneity represented by a Poisson binomial and a power-law distribution, respectively. Figure 5.11a shows the search space of the solver for all four combinations. Heterogeneity drastically reduces the search space. Moreover, the instances with just a heterogeneous vertex degree are easier than the ones with just heterogeneous hyperedge sizes. This fits nicely with the data sets from the previous section. There, the appliation instances tend to have more heterogeneous vertex degrees and more homogeneous hyperedge sizes. In detail, for application instances with $\|\mathcal{F}\| > 100$, the coefficient of variation for edge degrees is 1.03 on average, while it is 0.46 for hyperedge sizes.

5.5.2 Vertex to Edge Ratio

Figure 5.11b considers the effect of the ratio between vertices and edges on performance. We focus on homogeneous hyperedge sizes and heterogeneous vertex degrees to mirror the application instances. Note that the instance size is significantly larger than in Figure 5.11a. With this choice of distribution, most instances would be trivial otherwise. The figure shows the search space for instances with 5000 hyperedges with average size of ten over a growing number of vertices. The figure indicates the trend that having a large amount of vertices in comparison to the number of hyperedges makes the instances more difficult. Aside from the larger input size, which should not matter as much, the trend can be explained by the fixed hyperedge size. Since the average hyperedge size is fixed to ten, more vertices implies smaller vertex degrees which in turn implies larger solution sizes. Large solutions are generally difficult to find, since they require either more

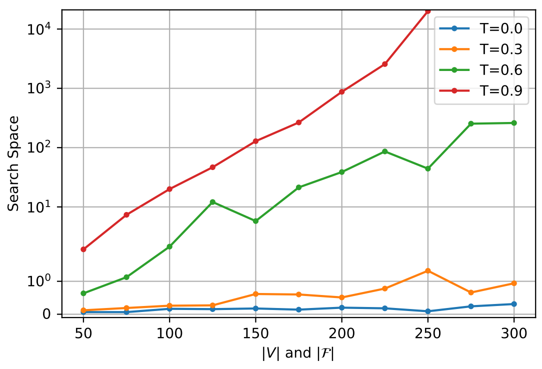


Figure 5.13: Search space by instance size for different temperatures. Vertex degrees are heterogeneous and hyperedge sizes homogeneous.

successful reductions or more branching. Preliminary experiments confirmed a similar trend for other distributions as long as at least one of them is heterogeneous.

5.5.3 Hyperedge Size

Figure 5.12 shows the search space over a growing average size of hyperedges. Interestingly, the results vary drastically depending on which distribution is chosen as the heterogeneous one. For heterogeneous vertex degrees larger hyperedges make the instance easier (see Figure 5.12a). In contrast, larger hyperedges make the instance harder for heterogeneous hyperedge sizes (see Figure 5.12b). The former is due to the trivial one or two vertex solutions that are reached far earlier when vertex degrees are heterogeneous. We explain the latter by the domination rules, which are less effective the larger the hyperedges. Note, however, that extremely small and extremely large hyperedge sizes should both be easy in the limit. Too small hyperedges of size zero or one lead to trivial instances while too large hyperedges lead to tiny (one or two vertex) solutions. Thus, the different trends in difficulty that are visible in the plots are due to the range of values that we investigated; in this case 5-50.

5.5.4 Locality

The temperature parameter T of the GIRG model controls the clustering, that is, it controls the extent to which the geometry is respected during generation. Higher values of T means less clustering. Figure 5.13 shows the search space over growing instance size for different values of T. Due to high variance this plot is averaged over 200 repetitions per data point instead of 50. As the algorithm was built for realistic instances, which usually have a high amount of clustering, it is not surprising that the search space grows rapidly when there is less clustering. A lack of clustering not only makes the instance harder but higher T also makes the solver scale worse with instance size. We suspect the domination rules to be the reason for this drastic impact on performance because they benefit from high clustering.

5.6 Conclusion

We provide a fast branch-and-bound solver that beats a modern ILP solver, which is the state-of-the-art for solving the minimum hitting set problem. Our implementation provides a baseline for future work in this direction. We explain the basic building blocks of our algorithm — which are lower bounds, upper bounds, and reduction rules — and experimentally evaluate their run time and efficiency to find a good configuration of used rules and bounds. We confirm the effectiveness of Weihes reduction rules noted in previous works. Another crucial part of the algorithm turns out to be the quality of lower bounds. The parameter-dependent *Costly Discard Rule* builds upon lower and upper bounds and contributes significantly to the performance of our algorithm. We find that the algorithm behaves differently on random inputs. Lastly, we find that the performance heavily depends on the heterogeneity of the degree distribution as well as the amount of locality. In the future, it would be interesting to determine why non-random instances are easier for our solver than random instances and if their structure can be exploited to design even faster algorithms for practical instances.

6 A Branch-And-Bound Algorithm for Cluster Editing

This chapter is based on joint work with Thomas Bläsius, Philipp Fischbeck, Lars Gottesbüren, Michael Hamann, Tobias Heuer, Jonas Spinner and Marcus Wilhelm [Blä+22a]. As a team, we participated in and won the exact and the heuristic track of the 2021 PACE challenge. Although both submissions were a team effort, Lars Gottesbüren, Michael Hamann, and Tobias Heuer focused more on the heuristic solver [Blä+21a], while the rest of the team focused on the exact solver [Blä+21b]. The evaluation of the exact algorithm was done after the participation in the challenge.

6.1 Introduction

In graph clustering, the goal is typically to partition the vertices into clusters such that there are many edges inside and few between clusters. The most clear-cut cases are so-called *cluster graphs* in which each connected component forms a clique. Thus, with one cluster for each connected component, there are no edges between clusters and all possible edges inside clusters exist. The *cluster editing problem* asks to use as few edge insertions and deletions as possible to transform a given graph into a cluster graph; thereby computing a clustering.

The cluster editing problem is NP-hard [KM86] and thus we cannot expect to solve it efficiently in general. Nonetheless there are algorithmic approaches using reduction rules [CC12; CM12; Guo09] or search trees [Böc+09; HH15]. The theoretically fastest known algorithm is by Böcker [Böc12] with a running time of $O(1.62^k + n + m)$, where k is the number of edits (edge insertions plus deletions) and n, m are the number of vertices and edges of the graph, respectively. To encourage development and implementation of practical algorithms, the challenge of PACE 2021 [Kel+21] was to solve cluster editing. Our solvers won the exact [Blä+21b] and heuristic [Blä+21a] track.

In this paper, we describe the details of our exact solver [Blä+21b] and present an in-depth evaluation. Roughly speaking our solver is a branch-and-bound algorithm: Whenever possible, we apply reduction rules to shrink the instance. When no reductions apply, we branch on the decision whether to put a pair of vertices in the same or in different clusters. To reduce the size of the resulting search tree, we compute lower bounds on the optimal solution and prune subtrees where the lower bound exceeds the upper bound computed by our heuristic solver.

Our lower bounds are based on so-called *packings* of small substructures for which we know optimal solutions. This approach has been used before to solve related problems [Got+20b] and for cluster editing in particular by packing paths of length 3 [HH15]. We generalize this approach to support larger substructures and weighted[1] instances; see Section 6.3.2. For the reduction rules, we use various rules from the literature [Bas+16; BBK11; CC12; CM12; Got+20b; Guo09] as well as newly developed ones; see Section 6.3.3. One type of reduction rule are so-called *forced choices* that essentially look ahead one branching step, e.g., if putting two vertices in different clusters would yield a lower bound exceeding the upper bound, one must put them in the same cluster. Thus, the lower bounds and the reduction rules are intertwined in the sense that better lower bounds lead to more applications of the forced choices rules. In Section 6.4 we evaluate how effective and efficient different reductions and lower bounds are, using the instances from the PACE challenge. Additionally, we evaluate our algorithm on geometric inhomogeneous random graphs [BKL19], which lets us study scaling behavior of our solver and its efficiency depending on certain instance properties. Our main findings are summarized as follows.

- The instances we can solve are usually already solved by just the reduction rules, i.e., once we have to apply branching we usually do not find a solution within reasonable time.
- The forced choices reduction rules are by far the most effective rules. This identifies good lower bounds as the key ingredient of our algorithm.
- Using packings of stars instead of paths of length 3 is still computationally feasible and yields substantially better lower bounds.
- The solver performs better on graphs with low average degree and if the graph is well-clusterable.
- The upper bounds computed by our heuristic solver are exceptionally good. In fact, the heuristic solver found an optimal solution on all instances where we know it.

6.2 Preliminaries

Let $G = (V, E)$ be a simple, undirected graph. The *cluster editing problem* asks to transform G into a disjoint union of cliques with the least number of edge edit operations. An edit is the deletion of an existing edge or the insertion of a missing edge. As a graph is a disjoint union of cliques if and only if it does not contain an induced path on three vertices (a P_3), the problem can also be seen as P_3-free editing.

The *weighted cluster editing problem* replaces the set of edges with a symmetric cost function $s : V \times V \mapsto \mathbb{Z}$. If $s(uv) > 0$, the pair uv is considered an edge with a deletion cost of $s(uv)$. For $s(uv) < 0$ the pair uv is a non-edge with an insertion cost of $-s(uv)$. A vertex pair with $s(uv) = 0$ is called a zero-edge that can be inserted or deleted for free. A solution to the weighted cluster editing problem is a partition of vertices. We associate a solution K with its corresponding equivalence relation $\equiv_K$. The cost of K for

[1]Though the input is unweighted, our reduction rules as well as the branching lead to weighted instances.

the instance (V, s) is defined as the total cost of edges between clusters and non-edges inside clusters. That is,

$$\mathrm{cost}(K, s) = \sum_{\substack{s(uv)>0 \\ u \not\equiv_K v}} |s(uv)| + \sum_{\substack{s(uv)<0 \\ u \equiv_K v}} |s(uv)|.$$

6.3 The Branch-and-Bound Algorithm

Our algorithm uses branch-and-bound to solve the decision variant of cluster editing, which asks if there exists a solution with a cost of k or less. The optimization problem is solved by calling the decision variant with increasing values of k. At its core our algorithm is a simple recursive subroutine that computes a kernel by applying reductions, returns if the lower bound for the remaining instance is above k, and otherwise branches on the inclusion or exclusion of an edge in the solution, introducing *permanent* and *forbidden* edges into the instance. We select the edge to branch on by highest edit cost and tiebreak by the number of P_3s that overlap this edge. As outlined by Böcker et al. [Böc12; Böc+09], the endpoints of permanent edges can immediately be merged to obtain an equivalent weighted instance[2] of smaller size.

Since branching creates weighted instances, we initially apply a series of reductions that are only possible for unweighted instances before we run the recursive branch-and-bound algorithm. Furthermore, we split the initial instance into connected components and solve them separately because an optimal solution never connects them.

In the following we discuss the different parts of our algorithm. In Section 6.3.1, we mention our approach to obtain upper bounds. Section 6.3.2 introduces and generalizes the concept of lower bounds via conflict packings. Finally, we list the used reduction rules and explain their application in our algorithm in Section 6.3.3.

6.3.1 Upper Bounds

An upper bound for the optimal solution is crucial for any branch-and-bound algorithm to identify and prune branches that cannot lead to an optimal solution. For this, we use our heuristic solver that won the heuristic track of the 2021 PACE challenge [Blä+21a]. The heuristic solutions are optimal on all 173 of the 200 test instances for the exact track we were able to solve (see Section 6.4). We refer to the solver description [Blä+21a] for details about the algorithm.

6.3.2 Lower Bounds

For lower bounds, we use an idea from recent solvers for this and other similar problems [Blä+22c; Got+20b; HH15]. The idea is to find a large set of vertex-pair disjoint P_3s. Recall that cluster editing can be seen as P_3-free editing. We call such a set a *conflict-packing* or *packing* for short. Since each P_3 in a packing needs at least one edit

[2]Forbidden edges have a weight of negative infinity.

to resolve and no edit overlaps with more than one conflict, the size of the set is a lower bound on the required number of edits.

Finding a maximum disjoint set of conflicts is an independent set problem, which is hard to solve in general. We are not aware of complexity results for independent set on this specific kind of intersection graph. Hartung et al. [HH15] use the commonly known *small degree* heuristic and some random perturbation to find a maximal set of P_3s. Gottesbüren et al. [Got+20b] also use the concept of a conflict packing in their algorithm for the quasi-threshold editing problem, i.e., $\{C_4, P_4\}$-free editing. They propose to use a local search with random replacements and the 2-improvement heuristic for independent set to grow the packing [ARW12].

With these heuristics, good P_3 packings can be found in reasonable time. However, P_3 packings have two major drawbacks. First, they are defined only for unweighted instances while the best known branch-and-bound algorithms for cluster-editing work on weighted instances [Böc12; Böc+09]. Second, each P_3 has two edges and one non-edge. Therefore a P_3 packing can never be larger than $|E|/2$ while many difficult instances require more than $|E|/2$ edits. We propose a framework to circumvent both these drawbacks by generalizing conflict packings.

In the following we formalize the concept of packing arbitrary substructures for which we know lower bounds into weighted instances. We call two cost functions $a, b : V \times V \mapsto \mathbb{Z}$ *conflicting* if there is a vertex pair uv that is a non-edge in (V, a) and an edge in (V, b) or vice versa, i.e., if $|a(uv) + b(uv)| < |a(uv)| + |b(uv)|$. Let $a + b$ denote the element-wise addition of the functions a, b, that is, $(a + b)(uv) = a(uv) + b(uv)$. We call a set of cost functions P a *packing* for the instance (V, s) if (1) they are pairwise non-conflicting, (2) they are non-conflicting with s, and (3) they do not exceed s in any vertex pair, that is $\sum_{p \in P} |p(uv)| \leq |s(uv)|$. Note that $\sum_{p \in P} |p(uv)| = |\sum_{p \in P} p(uv)|$ because of property (1). Also note that property (1) actually follows from (2) and (3).

Lemma 4. *Let a, b, c be three cost functions. If they are pairwise non-conflicting, then $(a + b)$ and c are non-conflicting.*

Proof. Let uv be a vertex pair with $c(uv) > 0$. If $c(uv) > 0$, then $a(uv) \geq 0$ and $b(uv) \geq 0$ since both are non-conflicting with c. Thus $(a + b)(uv) = a(uv) + b(uv) \geq 0$ which means that $(a + b)$ is non-conflicting with c on this vertex pair. The case for $c(uv) < 0$ works analogously. The case where $c(uv) = 0$ cannot lead to a conflict for c with any other function. □

Lemma 5. *Let a, b be two non-conflicting cost functions. If for all vertex pairs uv, $|a(uv)| \leq |b(uv)|$, then $\text{opt}(V, a) \leq \text{opt}(V, b)$.*

Proof. Let K be a solution for (V, a). Lemma 5 follows from the fact that all vertex pairs that are edited in $\text{cost}(K, a)$ are present in $\text{cost}(K, b)$ with greater or equal absolute value. □

Lemma 6. *For non-conflicting cost functions a, b, $\text{opt}(V, a) + \text{opt}(V, b) \leq \text{opt}(V, a + b)$.*

Proof. Let K be an optimal solution of $(V, a+b)$. Since a, b and $a+b$ are all non-conflicting (due to Lemma 4), there is never a non-edge in one instance that is an edge in the other. Thus we get,

$$\begin{aligned}\operatorname{opt}(V,a)+\operatorname{opt}(V,b) &\leq \operatorname{cost}(K,a)+\operatorname{cost}(K,b)\\ &= \sum_{\substack{a(uv)<0\\ u\equiv_K v}} |a(uv)| + \sum_{\substack{a(uv)>0\\ u\not\equiv_K v}} |a(uv)| + \sum_{\substack{b(uv)<0\\ u\equiv_K v}} |b(uv)| + \sum_{\substack{b(uv)>0\\ u\not\equiv_K v}} |b(uv)|\\ &= \sum_{\substack{(a+b)(uv)<0\\ u\equiv_K v}} |(a+b)(uv)| + \sum_{\substack{(a+b)(uv)>0\\ u\not\equiv_K v}} |(a+b)(uv)|\\ &= \operatorname{cost}(K,a+b) = \operatorname{opt}(V,a+b).\end{aligned}$$

□

Theorem 1. *For packing P of the instance (V, s), $\sum_{p\in P} opt(V,p) \leq opt(V,s)$.*

Proof. Let $c : V \times V \mapsto \mathbb{Z}$ be the element-wise addition of all functions in P which is non-conflicting with s by Lemma 4. Moreover, Lemma 6 implies that $\sum_{p\in P} \operatorname{opt}(V,p) \leq \operatorname{opt}(V,c)$. Since P is a packing for (V, s) the third property of packings states that for all vertex pairs uv: $|c(uv)| \leq |s(uv)|$. Therefore, Lemma 5 results in $\operatorname{opt}(V,c) \leq \operatorname{opt}(V,s)$. □

The theorem states that we can pack structures together and sum their lower bounds to obtain a lower bound for the initial instance. A P_3, for example, is represented by a cost function that is zero throughout except for its three (non-)edges. Therefore, the concept generalizes P_3 packings to weighted instances. Moreover, our formulation of a packing allows for other structures than P_3s. Recall that P_3 packings have the drawback that they cannot exceed $|E|/2$. To remedy this, we have to find other structures that have a better lower bound to edge ratio. Actually, a star S_k with k leaves (thus k edges and $\binom{k}{2}$ non-edges) cannot be solved with less than $k-1$ edits. Coincidentally, a P_3 is a star with two leaves. One can even generalize from stars to complete bipartite graphs $K_{a,b}$ which cannot be solved in less than $a \cdot (b-1)$ edits. A star S_k is just a $K_{1,k}$. So there is a tradeoff between structures that are easy to find and pack and structures that have strong lower bounds.

We implemented a P_3 packing, a star packing, and a $K_{a,b}$ packing. In preliminary experiments, we observed that the quality of star and $K_{a,b}$ packings were similar while star packings were easier, and thus slightly quicker, to compute. We thus focus on star bounds in the following. Our implementation of the star packing builds upon the P_3 packing described by Gottesbüren et al. [Got+20b]. They go through all items in the packing and try to replace one with two currently not in the packing. To not get stuck in a local optimum, they also randomly replace an item with one other item with a small probability when it cannot be replaced by two new ones. We make three major changes. First, we introduce more mutations that change the lower bound by exactly one. For P_3s, the packing grows by removal of one P_3 and insertion of two new ones in its place. We never insert or remove stars with more than two leaves. Instead, we add the option to add/remove a leaf. Second, when possible we merge a star with another existing star instead of mutating it. The merge increases the lower bound of the packing by one. Third, we relax the termination condition for the local search. They stop if the packing does not grow for five iterations. In contrast, we continue while the average number of improving iterations is still above one in five, i.e., five times the number of improving iterations is greater or equal the number of total iterations. This leads to better packings for instances that benefit from longer local search while still being fast on instances that quickly hit a local maximum. Finally note that the packing is weighted but we only pack or modify unweighted structures. For performance, however, we associate an integer weight with each star to represent multiple identical overlapping stars.

6.3.3 Reduction Rules

There exist various reduction rules [Bas+16; BBK11; CC12; CM12; Got+20b; Guo09] and we introduce additional ones (Forced Choices Single Merge and Clique-Like Subgraph). In the following, we discuss the reduction rules used by our solver and go into detail on how our solver applies the rules.

Twin Simple [Guo09]. This rule merges vertices with identical neighborhoods and is part of the unweighted $4k$ kernel based on critical cliques [Guo09]. The rule originally only works for unweighted instances. We generalize it to a pair of vertices in the weighted setting as follows. We can merge u and v with $s(uv) \geq 0$ if their edit cost to every other vertex differs by the same constant positive factor, i.e., there exists a $c > 0$ such that $s(uw) = c \cdot s(vw)$ for every other vertex w. The correctness proof is analogous to the unweighted case and goes roughly as follows. If v being in a certain cluster produces cost X, then u being in this cluster produces cost cX. Thus, the cheapest cluster for v is also the cheapest cluster for u, though there could be multiple equally cheap clusters. In the latter case it is nonetheless still not worse to put u and v together as $s(uv) \geq 0$. We note that applying this rule repeatedly to an initially unweighted instance merges all critical cliques.

Twin Complex [BBK11, Rule 5]. Let u, v be two nodes that are connected with an edge. The rule considers, for all possible ways to separate them into different cliques, the worst case cost of moving one into the clique of the other. If deleting the edge uv is at least as expensive as this worst case, then there is an optimal solution with u, v in the same clique and the edge can be contracted. The rule is checked with a dynamic programming (DP) approach [BBK11].

Unfortunately, the DP degenerates when dealing with forbidden edges, i.e., edges with cost $-\infty$. In the following, we discuss why this problem exists and what we did to fix it. If u and v have a similar neighborhood, then there is no worst case where both, moving u into v's clique or vice versa, are expensive. Intuitively, the rule works because one of the two options is always cheap. Now consider a vertex w with non-edges to u and v. If another reduction finds the edge uw to be forbidden, i.e., $s(uw) = -\infty$, then two things happen to the DP. First, the solutions that put u and w in the same clique can be ignored, which is beneficial as it makes it more likely that v can be moved into the clique of u. Second, the worst case will put w and v together, which makes it impossible to move u into the cluster of v. Thus, due to the second implication, knowing that uw is forbidden can have a detrimental effect on the applicability of the reduction rule. In fact, the DP degenerates to the point that not even true twins (except for the forbidden edge to w) can be merged. To circumvent this problem, we remember the edit cost for edges that are marked forbidden throughout the whole algorithm. In the DP we then use the original weights (getting rid of the downside due to the second aspect) but still skip solutions that put forbidden node pairs in the same clique (still using the upside of knowing uw is forbidden for the first aspect).

Induced Cost Forbidden/Permanent (icf,icp) [BBK11]. Let Δ denote the symmetric difference. The induced costs for setting a vertex pair to forbidden (icf) or permanent (icp) are

$$\begin{aligned}\mathrm{icf}(uv) &= \sum_{w \in N(u) \cap N(v)} \min\{s(uw), s(vw)\} \\ \mathrm{icp}(uv) &= \sum_{w \in N(u) \Delta N(v)} \min\{|s(uw)|, |s(vw)|\}.\end{aligned}$$

If the induced cost of setting a vertex pair to forbidden (icf) exceeds the current budget, then the pair must be merged. If the induced cost of setting a vertex pair to permanent (icp) exceeds the current budget, then the pair must be forbidden.

Heavy Non-Edge [BBK11, Rule 1]. If $s(uv) < 0$ and $|s(uv)| \geq \sum_{w \in N(u)} s(uw)$, i.e., inserting the edge uv is at least as expensive as isolating u by cutting all of its edges, one can set uv to forbidden, forcing u and v to be in different clusters.

Heavy Edge, Single End [BBK11, Rule 2]. If $s(uv) \geq \sum_{w \in V \setminus \{u,v\}} |s(uw)|$, i.e., deleting uv is at least as expensive as editing all other pairs involving u, one can merge u and v.

Heavy Edge, Both Ends [BBK11, Rule 3]. If $s(uv) \geq \sum_{w \in N(u) \setminus \{v\}} s(uw) + \sum_{w \in N(v) \setminus \{u\}} s(vw)$, i.e., deleting uv is at least as expensive as deleting all other edges adjacent to u and v, one can merge u and v, as it is always better to let u and v form their own cluster of size 2 than to separate them.

Distance Three Rule [Bas+16]. Two vertices with distance three or more cannot be in the same cluster in an optimal solution. Therefore, all vertex pairs with distance three or more are initially marked as forbidden. This does not apply to weighted instances.

Forced Choices, all Pairs [Got+20b]. If setting an edge to forbidden or permanent would raise the lower above the upper bound, then the opposite edit must be performed. In other words, we identify an edge where, if branched on it, one branch would be pruned immediately.

A naive implementation of this rule is too slow as it requires a quadratic number of lower bound (i.e. packing) computations [Got+20b]. However all these packings are similar. Given a packing lower bound for the instance, we locally modify the packing for each vertex pair to obtain the required bounds. Because a packing changes only locally, this can be done significantly faster than computing $\binom{n}{2}$ packing lower bounds from scratch.

Forced Choices, Single Merge. Updating the lower bounds as in the previous rule is usually worse than computing the bounds from scratch. Thus, we additionally identify a constant number of edits that are unlikely to be included in the optimal solution and compute lower bounds for them from scratch. Specifically, we choose five vertex pairs and test if editing them to non-edges would be too expensive so that the rule produces a merge when applicable. We do not test for the converse because merges are far more rewarding than finding a single forbidden edge. To choose the five pairs, a heuristic estimates in advance which pairs would produce the highest cost when set to non-edges. Criteria for this heuristic are the cost of the edit as well as the number of overlapping P_3s with the vertex pair before and after the edit.

Clique-Like Subgraph. Given the instance (V, s) and the subset $C \subset V$, we define the *C-subinstance* (V, s_C) by setting $s_C(u, v) = s(u, v)$ if $u \in C$ or $v \in C$ and $s_C(u, v) = 0$ otherwise, i.e., if $u, v \in V \setminus C$. The rule states that, if the C-subinstance has no better solution than to isolate C into a singleton cluster, then we can isolate C in the original instance. The rule can be checked by exactly solving the C-subinstance. We note that the instance (V, s_C) is likely easier than (V, s) due to the following observation. When looking at the graph with vertex set V with an edge between u and v if $s(u, v) > 0$, then we expect the closed neighborhood $N[C]$ of C to be rather small. Moreover, for a vertex $u \in V \setminus N[C]$ and any other vertex $v \in V$, we have $s_C(u, v) = 0$. Thus, we know that there is an optimal solution of (V, s_C) that has u as singleton, which reduces the instance to only the vertices in $N[C]$.

To show that the rule is correct, we prove the following theorem.

Theorem 2. *Let (V, s) be an instance of* Weighted Cluster Editing*, and let $C \subset V$ be a set of vertices. If an optimal solution for the C-subinstance isolates C into its own cluster, then there is an optimal solution for (V, s) that does so as well.*

Proof. Let $\mathcal{P}$ be any solution of (V, s). We construct a new partition $\mathcal{P}^\star$ that isolates C such that $\text{cost}(\mathcal{P}^\star, s) \leq \text{cost}(\mathcal{P}, s)$. For every cluster $A \in \mathcal{P}$, the partition $\mathcal{P}^\star$ contains $A \setminus C$. Additionally $\mathcal{P}^\star$ contains C.

For a pair $u, v \in V$, we say that $\mathcal{P}$ *splits* u and v if u and v are in different clusters of $\mathcal{P}$; otherwise $\mathcal{P}$ *joins* u and v. Similarly, for $C \subseteq V$, we say that $\mathcal{P}$ *splits* C if $\mathcal{P}$ splits at least one pair of vertices in C. Otherwise, if C is contained in a cluster of $\mathcal{P}$, then $\mathcal{P}$ *joins* C. We regularly need to sum over only negative or only positive cost. To simplify notation in these cases, let $s^+, s^- \colon V \times V \to \mathbb{N}$ be defined as $s^+(u,v) = \max(0, s(u,v))$ and $s^-(u,v) = \max(0, -s(u,v))$. The cost of the solution $\mathcal{P}$ is then defined as

$$\operatorname{cost}(\mathcal{P}, s) = \sum_{\substack{u,v \in V \\ \mathcal{P} \text{ splits } u,v}} s^+(u,v) + \sum_{\substack{u,v \in V \\ \mathcal{P} \text{ joins } u,v}} s^-(u,v).$$

For the subset $C \subset V$ and its complement $T = V \setminus C$ it will be useful to split the sum in $\operatorname{cost}(\mathcal{P}, s)$ by vertex pairs within C, pairs between C and T, and pairs within T. We have

$$\begin{aligned}\operatorname{cost}(\mathcal{P}, s) = &\sum_{\substack{u,v \in C \\ \mathcal{P} \text{ splits } u,v}} s^+(u,v) \;+ \sum_{\substack{u,v \in C \\ \mathcal{P} \text{ joins } u,v}} s^-(u,v) \;+ \\ &\sum_{\substack{(u,v) \in C \times T \\ \mathcal{P} \text{ splits } u,v}} s^+(u,v) \;+ \sum_{\substack{(u,v) \in C \times T \\ \mathcal{P} \text{ joins } u,v}} s^-(u,v) \;+ \\ &\sum_{\substack{u,v \in T \\ \mathcal{P} \text{ splits } u,v}} s^+(u,v) \;+ \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ joins } u,v}} s^-(u,v).\end{aligned} \tag{6.1}$$

Now we can bound the cost of $\mathcal{P}^\star$ in the original instance. Consider the six sums of $\operatorname{cost}(\mathcal{P}^\star, s)$ as in Equation (6.1). The first sum evaluates to 0 as $\mathcal{P}^\star$ does not split pairs in C. Similarly, the fourth sum evaluates to 0, as $\mathcal{P}^\star$ does not join vertices from C with vertices from T. For the second and third sum, we can drop the condition that $\mathcal{P}$ joins and splits u, v, respectively, as $\mathcal{P}$ joins all pairs in C and splits all pairs between C and T. Finally, $\mathcal{P}^\star$ splits a vertex pair $u, v \in T$ if and only if $\mathcal{P}$ does, i.e., we can exchange $\mathcal{P}^\star$ with $\mathcal{P}$ in the fifth and sixth sum. Thus, writing the remaining sums in order $5, 6, 2, 3$, we obtain

$$\begin{aligned}\operatorname{cost}(\mathcal{P}^\star, s) &= \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ splits } u,v}} s^+(u,v) \;+ \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ joins } u,v}} s^-(u,v) \;+ \sum_{u,v \in C} s^-(u,v) \;+ \sum_{(u,v) \in C \times T} s^+(u,v) \\ &= \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ splits } u,v}} s^+(u,v) \;+ \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ joins } u,v}} s^-(u,v) \;+ \operatorname{opt}(V, s_C) \\ &\le \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ splits } u,v}} s^+(u,v) \;+ \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ joins } u,v}} s^-(u,v) \;+ \operatorname{cost}(\mathcal{P}, s_C) \\ &= \operatorname{cost}(\mathcal{P}, s).\end{aligned}$$

The first equality follows from the premise of the theorem that there is an optimal solution for the C-subinstance that isolates C. Any solution for the subinstance that isolates C has to pay for all non-edges in C as well as all *edges* from C to $V \setminus C$. Moreover the solution that keeps all other vertices as singletons incurs no additional cost beyond this. Therefore the premise can alternatively be stated as

$$\sum_{u,v \in C} s^-(u,v) \; + \sum_{(u,v) \in C \times T} s^+(u,v) = \operatorname{opt}(V, s_C). \tag{6.2}$$

For the inequality, we have $\operatorname{opt}(V, s_C) \leq \operatorname{cost}(\mathcal{P}, s_C)$ because $\mathcal{P}$ is also a solution for (V, s_C). Regarding the last equality, note that $\operatorname{cost}(\mathcal{P}, s_C)$ coincides with $\operatorname{cost}(\mathcal{P}, s)$ except that the last two terms of Equation (6.1) evaluate to 0 for $\operatorname{cost}_S(\mathcal{P}, s_C)$, i.e.,

$$\operatorname{cost}(\mathcal{P}, s) = \operatorname{cost}(\mathcal{P}, s_C) \; + \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ splits } u,v}} \operatorname{cost}^+(u,v) \; + \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ joins } u,v}} \operatorname{cost}^-(u,v).$$

In conclusion, we obtain $\operatorname{cost}(\mathcal{P}^\star) \leq \operatorname{cost}(\mathcal{P})$, which proves the claim. □

Unused Reductions. There are other reduction rules in the literature, which did not make it into our solver for different reasons, which we briefly discuss in the following.

The Clique-Like Subgraph rule is similar to Rule 4 by Böcker et al. [BBK11], which is based on min-cuts. Since the min-cut rule can be computed more efficiently it could be useful in a solver, but we found it to be ineffective during preliminary testing. It has a similar drawback as the Clique-Like Subgraph rule, in that it does depend on the choice of a subset C. This drawback has been lifted in a recent result, which presents an algorithm to efficiently find all subsets to which the rule is applicable [Sch+22]. Thus, the inclusion of the min-cut rule in future solvers should be considered.

We implemented the reduction rules by Cao and Chen [CC12] leading to a kernel of size $2k$, which is the smallest known kernel for cluster editing. However, our preliminary experiments showed that these reduction rules were dominated by other rules. Moreover, the rules explicitly exclude zero-edges, which we can get due to edge contractions, and it is not obvious how to adapt the rules to this setting.

Böcker et al. [BBK11] suggest the improvement to the Induced Cost Forbidden/Permanent rules to add a lower bound for the graph without u and v to the induced costs to obtain an even stronger rule. The P_3-packing bound exactly captures this idea, even if it does not explicitly compute icf / icp, because all triangles formed with uv have a combined cost of icf / icp and can all be included in the packing because they share only the vertex pair uv. Moreover, the P_3-packing bound is strictly stronger in a weighted setting because it can additionally use the residual cost of edges uw and vw for all $w \in V \setminus \{u, v\}$ after each P_3 constituting the icf / icp is removed. Although the Forced Choices All Pairs rule with P_3 packings as bound dominates this improved version of the icp/icf rules, we keep the icp/icf rules in the solver as a failsafe to detect possible errors in the involved implementation of the forced choices rule.

Finally, there are the rules used in the unweighted $4k$ and $2k$ kernels [CM12; Guo09]. We have not implemented or adapted them to a weighted setting except for our generalization of the Simple Twin rule. The kernels are based on critical cliques, i.e., a clique containing nodes with identical closed neighborhood and the Simple Twin rule merges all critical cliques when applied repeatedly. Moreover, some other rules from these kernels are captured by our implemented rules. E.g., the Induced Cost Forbidden rule dominates rule 1 from the unweighted $2k$-kernel [CM12]. Nevertheless, the rules could be useful in future work.

6.3.4 Reduction Order

Reductions are checked sequentially in a certain order. If one reduction was applied, the process rechecks all reductions starting with the first one. Before the loop repeats, a new lower bound is computed to check if the current branch can be pruned. We chose the order of reductions by decreasing effectiveness. The forced choices reductions are the most effective reductions and come first. Twin Complex is also very effective, but we do Twin Simple before that because it is faster and already catches some cases for Twin Complex. The remaining reductions run in $O(n^3)$ each with low constant factors so their order does not matter as much. The final order of reductions that are checked during the branching algorithm is:

1. Forced Choices, all Pairs (Star)
2. Forced Choices, all Pairs (P_3)
3. Twin Simple
4. Twin Complex
5. Induced Cost Forbidden/Permanent
6. Heavy Edge, Both Ends
7. Heavy Edge, Single End
8. Heavy Non-Edge

The initial instance is reduced differently. The Distance Three rule is applied once before the reduction loop since it is only applicable to unweighted instances. Then, the Clique-Like Subgraph reduction checks the clusters that are found by the heuristic solver. The optimal solution for the subinstances is computed with the exact solver itself. To keep the running time reasonable we skip subinstances with 50 vertices or more and run the solver with a timeout of 5 seconds. Finally, the other reductions are applied in a loop. During the loop, the order differs in three aspects from the order given in the list above. First, small connected components are brute-forced before the first item on the list. Second, Forced Choices Single Merge is added as a last reduction. Third, Force P_3 is applied before Force Star.

6.4 Experiments

In Section 6.4.1, we discuss the performance of our solver, the efficiency and effectiveness of the reduction rules, and the quality of lower and upper bounds. In Section 6.4.2 we perform scaling experiments and determine how structural properties affect the solver performance on geometric inhomogeneous random graphs (GIRGs) [BKL19]. As a generative network model, GIRGs can generate a series of similar instances that differ in a single property such as size, average degree, or clustering.

Setup. The experiments were run single threaded on a 4-Core Intel Xeon E5-1630v3 at 3.7GHz with 128GB DDR4 at 2133MHz. Each run has a soft timeout of one hour except for Figure 6.4a where it was 10 minutes per instance. Soft timeout means the current subroutine, is allowed to finish for the solver to terminate gracefully. To generate GIRGs, we use the efficient generator introduced in Chapter 2. The random components such as the generation of GIRGs or the local search to find a packing bound use the Mersenne Twister algorithm of the C++ standard template library. They are seeded as to produce deterministic results. In fact, each lower bound computation uses the same seed therefore producing the same output when given identical inputs. The code for the experiments, raw data, execution logs, instances, as well as the plotting code can be found in a branch of our public repository[3].

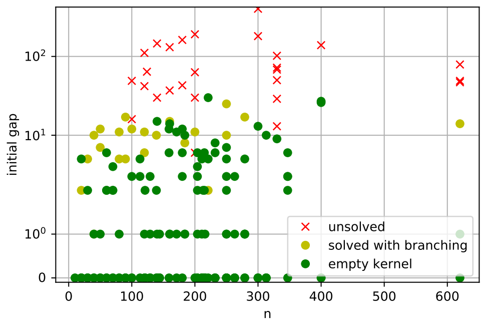
6.4.1 PACE Instances

In the following, we use the public and hidden instances from the 2021 PACE challenge to evaluate our solver. They represent a well balanced selection of instances from bioinformatics and data mining as well as randomly generated ones. Moreover, they are publicly available at the PACE website[4]. We discuss the effectiveness and efficiency of individual reduction rules as well as their combination used in the solver. Then, we compare the quality of the greedy upper bound to the lower bounds obtained by the P_3 and star packing, respectively.

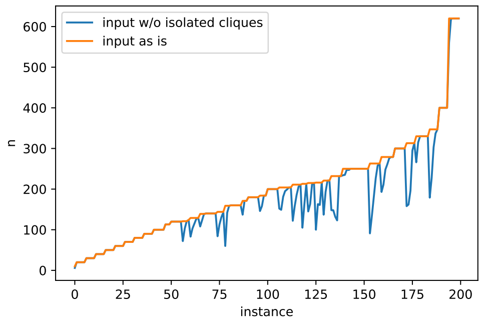
Solver Performance. In total, the algorithm solves 173 of the 200 instances from the PACE challenge with a timeout of one hour. Most instances finish significantly faster than that; 98 are solved in just one second and 160 finish in under a minute. Figure 6.1a shows for each instance if it was solved and whether reductions produce an empty kernel or branching was necessary. The axes correspond to the number of nodes and the initial gap between upper and lower bound. The gap is a good indicator of difficulty for our solver while the number of nodes seems unrelated to difficulty. All unsolved instances have a gap above 10. Surprisingly, 151 instances are solved with reductions alone. For a comparative evaluation of solver performance with other state-of-the-art algorithms, we refer to the official report of the 2021 PACE challenge [Kel+21]. On the hardware used

[3]`https://github.com/kittobi1992/cluster_editing/tree/experiments`

[4]`https://pacechallenge.org/`



(a) The initial gap between lower and upper bound.



(b) Vertices before/after removing isolated cliques.

Figure 6.1: For the 200 PACE instances, the gap between upper and initial lower bound (left) and the number of vertices per instance (right). In the left plot, the color indicates if an instance was solved by reductions only, needed branching, or remained unsolved in the given one-hour time limit.

in the actual challenge and a 30 min timeout, our algorithm solved 171 instances, while the second best submission solved 160.

Reduction Effectiveness. To evaluate the effectiveness of the reduction rules, we compute a kernel with each rule separately, i.e., apply the rule exhaustively with a soft timeout of one hour. This results in one kernel per combination of rule and instance. Before the kernel is computed we apply the Distance Three reduction rule which marks all vertex pairs in distance three or more as forbidden. Isolated cliques are removed from the input instance and once more from the final kernel. Figure 6.1b shows the size of the instances with and without the removal of isolated cliques. The results of the kernelization experiments can be seen in Figure 6.2. Each plot has a box for each reduction rule which represents the kernels made with this rule. There are two columns; the left one includes all instances while the right one only includes instances where the initial star bound does not match the upper bound. The instances with a gap between upper and lower bound represent more difficult instances for the solver thus making reductions more valuable on them. The rules *force p3* and *force star* refer to the Forced Choices, all Pairs reduction with the respective lower bound. The combination of the Induced Cost Forbidden and Permanent rules is labeled with *icx*. We also compute a kernel using all reduction rules (labeled as *all reds*) in the combination and order they are used by our solver to reduce the initial instance (see Section 6.3.3) excluding the brute-force of small components and the Clique-Like Subgraph reduction.

To evaluate the quality, we use three different measures. First, the number of vertices in the kernel, second, the number of edits that are already found, and third, whether the lower and upper bound get closer after computing the kernel. The number of vertices is a typical metric for kernel quality. For the second measure, the existence of an FPT algorithm for cluster editing indicates that the difficulty of the problem (the exponential

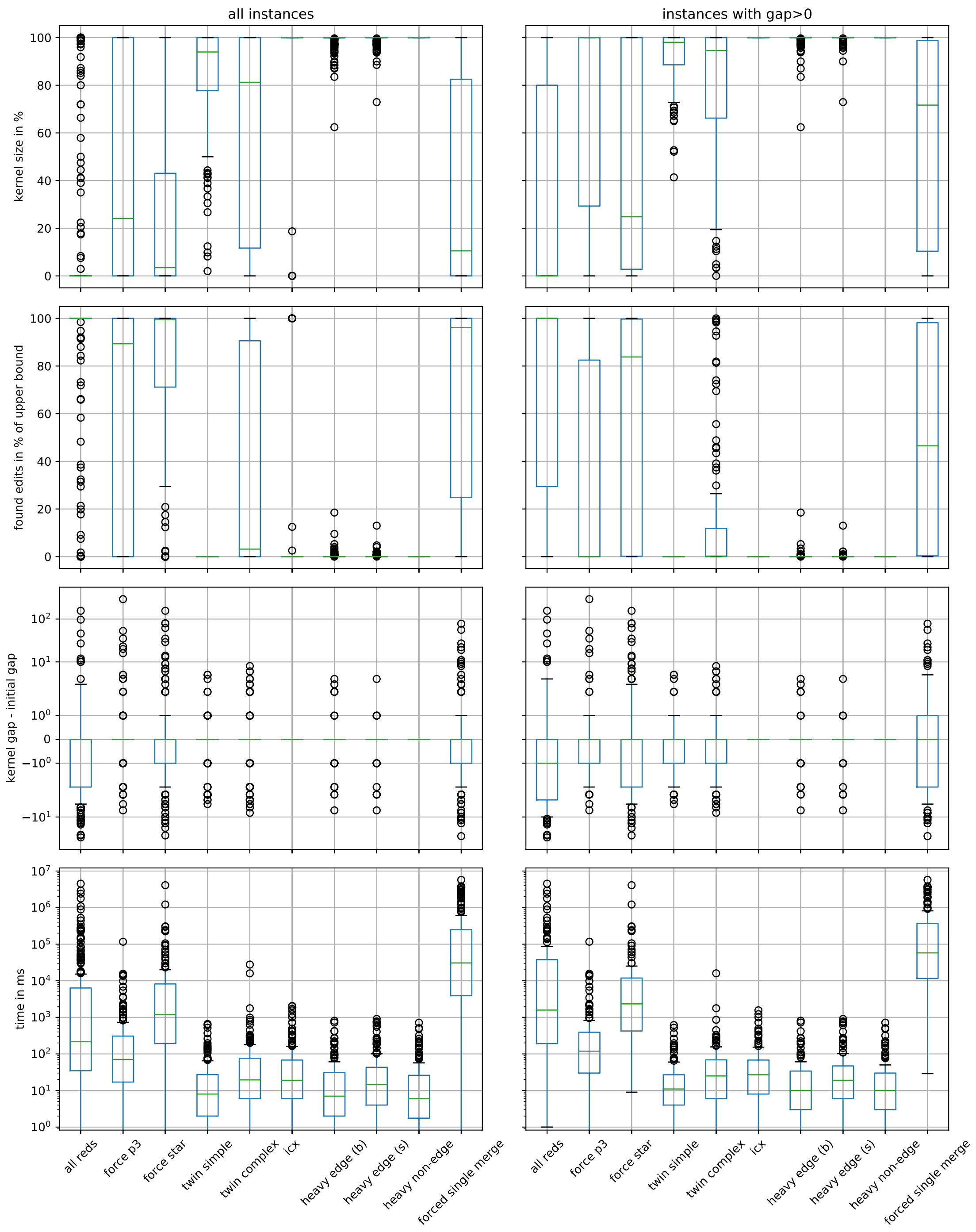


Figure 6.2: Kernels of PACE instances for each reduction rule. The rows show size, found edits, absolute gap change and time to compute the kernels. The left column includes all 200 instances while the right includes only the 121 instances with non-matching upper and lower bound. The label *all reds* refers to a combination of all other listed reduction rules.

part) is due to the size of the solution (the number of edits) rather than the size of the instance. In that sense, the percentage of edits that are already found during kernelization is a better indicator for progress towards a solution than instance size. The third measure, the gap between upper and lower bound, represents the difficulty of the kernel for any branch-and-bound solver using the lower bound that was used to compute the gap. With the change of the gap we estimate whether the kernel is easier or harder for our algorithm than the initial instance.

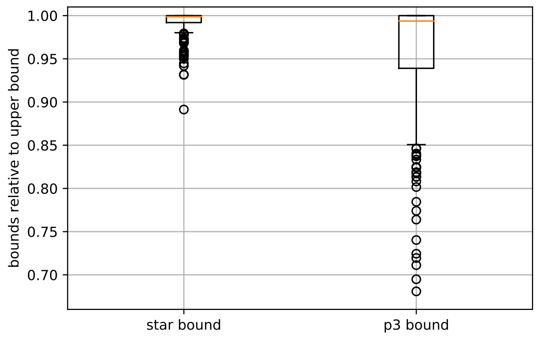
The first row shows the number of vertices in the produced kernels relative to the size of the initial instance without isolated cliques. The icx rules, both heavy edge rules and the heavy non-edge rule do not reduce the instance in the median. The heavy non-edge rule finds only forbidden edges. Therefore, the only way this rule could possibly reduce the number of vertices is by isolating a clique that is removed by our postprocessing. The simple twin and complex twin reductions are more effective with the complex twin producing smaller kernels. The non-zero gap instances prove to be harder for the twin reductions but the rules still find some application. The reductions based on forced choices produce the smallest kernels with an average size of 26% for force star, 44% for force P_3 and 36% for forced single merge. Best of all is the combination of all reduction rules that produces an empty kernel on more than 75% of instances and more than 50% of instances with a positive gap. On average, *all reds* reduces the instance to a size of 18%. While not explicitly shown, the instances with zero gap are interesting, too. The force star and the forced single merge reductions should produce an empty kernel in this case. While they indeed always apply initially, they do not always produce an empty kernel. This is because after the instance was reduced a few times it becomes weighted. Computing good packings for weighted instances becomes more difficult and might not be sufficient for the forced choices rules. In case of the forced single merge, the larger instances time out before the kernel is finished. Both these phenomenon happen rather rarely. On zero-gap instances the forced star produces a kernel size of 1.26% on average and 5.08% for forced single merge.

The second row shows the number of found edits that *must* be included in an optimal solution, i.e., the value that k is lowered by during kernelization. This value is normalized relative to the upper bound and represents another kind of progress towards solving the instance. The higher this value, the fewer choices remain to be fixed to solve the instance. Note that the values should be inverted when comparing with kernel size because for this plot 100% means solved while for instance size 0% means solved. Most reductions indicate similar results as for the kernel size. A notable exception is the simple twin rule. Since this rule only merges vertices with identical neighborhood, it never produces any cost but just reduces the instance. Interestingly, the progress made due to found edits is slightly better than the kernel size for the forced choices rules. For example, the force p3 kernel finds ca. 90% of edits in the median while the median kernel shrinks the instance by less than 80%. Since cluster editing is FPT in the number of edits, this is contrary to the expectation that the number of edits instead of the size of the instance should be responsible for the instance difficulty.

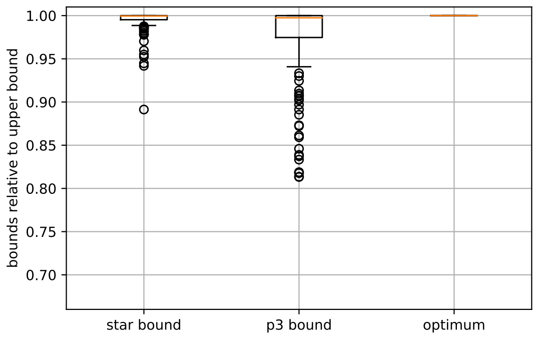
The third row shows the absolute change in gap after the kernel was computed, i.e., how much the difference between upper and lower bound has changed due to the kernelization. The y-axis uses symmetric log-scaling. A positive value means the gap grew and a negative value means the gap shrank. It might seem unintuitive that the gap can grow as previous lower bounds (before kernelization) still hold for the kernel. To explain this, consider two types of progress. The number of performed edits is *hard* progress, the lower bound represents *soft* progress. Once the total progress reaches the upper bound, the instance is solved. Hard progress is final but soft progress is temporary in the sense that, when actually solving the remaining instance, each reduction or branching step produces a new instance for which it might be more difficult to find good lower bounds. In general, there is no clear tendency for any reduction rule to only grow or only shrink the gap. The variance is very high to both sides. Thus, the inaccessibility of the kernel for soft progress sometimes outweighs the hard progress. In other cases it is the other way round or soft progress is even easier to achieve on the kernel. This is, e.g., the case for the simple twin rule, which makes no hard progress at all but has outliers to both sides. The median gap change is zero when looking at all instances. This is not surprising since 79 of the 200 instances already start with a gap of zero. For instances with a non-zero initial gap (the right column), the combination of all reductions actually reduces the gap by one for the median instance.

Reduction Efficiency. To evaluate the kernels by performance, we measure the time it takes to compute them. The last row of Figure 6.2 shows the results. The y-axis is logarithmic. Note that the highest outliers are approximately at $3.6 \cdot 10^6$ms which equals the soft timeout of one hour. All but the forced choices rules have a comparable run time with less than 100ms for more than 75% of the instances. Of these rules, just the twin complex and the icx rule have outliers over 1s and are in general the slowest of the non-forced choices rules. Note that the icx rule can be exhaustively applied in time $O(n^3)$ which is the same time one execution of the rule takes [Böc+09]. We instead apply the rule repeatedly since its run time is dominated by the forced choices rules. In the median, force p3 is slightly below 100ms, force star takes just above 1s, and forced single merge approximately one minute. All reductions combined are faster than force star but slower than force p3. Since all reductions produce by far the best kernels, this speaks for the order in which they are applied.

Bound Quality. Figure 6.3 compares the P_3 bound, the star bound, and the optimum solution. The values are given relative to the upper bound computed for the instance. Figure 6.3a aggregates this over all instances while Figure 6.3b contains only solved instances and additionally shows the optimum solution. Note that the y-axis begins at 0.7 which means that even the worst outlier is already fairly good. The first observation is that the optimum is at 1.0 relative to the upper bound with no variance between instances. In fact, the upper bound from our heuristic solver matches the optimum on all 173 instance we can solve. Therefore, the lower bounds can be considered relative to the optimum; at least for the right plot. In total, the star bound is significantly better than the P_3 bound with all but one instances having a bound at more than 90% of the upper



(a) Lower bounds via P_3, star packing (all instances).



(b) Lower bounds and optimum for solved instances.

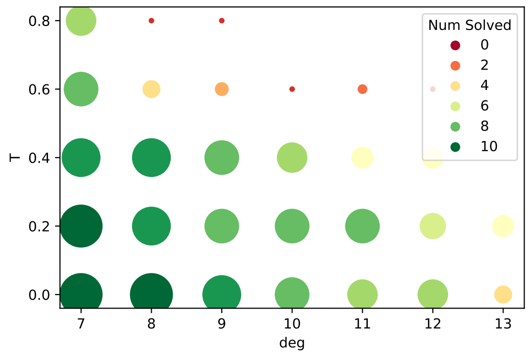
Figure 6.3: Packing lower bounds via P_3 and star packings on all instances (left) and on solved instances (right). The right plot additionally contains a column for the optimum. All values are relative to the respective upper bound for the instance. Note that the y-axis ranges from 0.7 to 1.0.

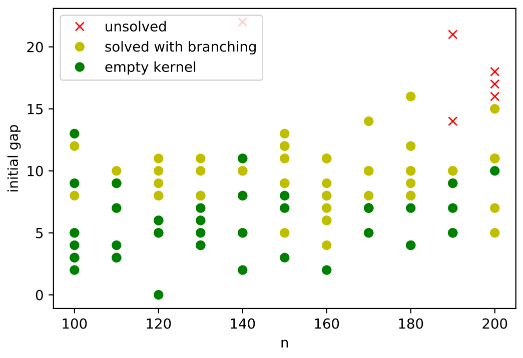
bound. The orange line for the median is only slightly lower for P_3 compared to the star bound but in this context this is huge. The number of edits in an optimal solution is approximately 1800 on average and goes as high as 27000. In contrast to this, we did not solve any instance with an initial gap of more than 30 (see Figure 6.1a), which is less than 2% of the 1800 edits needed on average. The average gap over all instances is 123 for P_3 and 15 for the star bound. Of course the average is heavily biased by the huge number of edits for the larger instances. Nevertheless, the 75th percentile ordered by absolute gap represents a gap of 43 for P_3 and 9 for the star bound, which makes the difference between solvable and unsolvable in this context.

6.4.2 Experiments on GIRGs

We use geometric inhomogeneous random graphs [BKL19] and our generator from Chapter 2 to benchmark the solver for a growing number of vertices, temperature, and average degree and to investigate the structure of an optimal solution. Unless noted otherwise, the number of vertices is 150, the average degree is ten, the power-law exponent describing the degree distribution is 2.9, the temperature parameter, which controls the degree of clustering, is zero (meaning high clustering) and the dimension of the ground space torus is two. For each set of input parameters for the GIRG model we generate 10 instances with different seeds.

Temperature and Average Degree. Figure 6.4a shows the effect of clustering and average degree on solver performance. For each combination of temperature and average degree, the plot shows how many of ten instances were solved in less than ten minutes. There is a clear threshold behavior that instances with both, high temperature and high average degree, are rarely solved. High average degree (13) and low temperature (0.0) is





(a) Number of solved instances by T and degree.

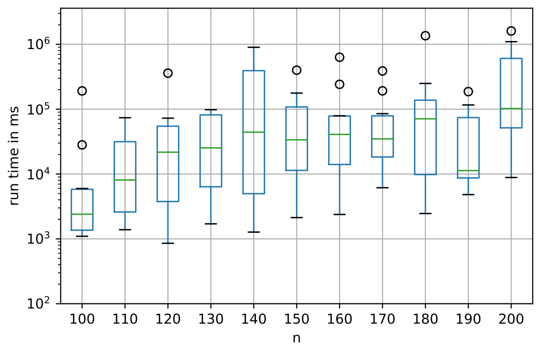
(b) The initial gap between lower and upper bound.

Figure 6.4: Solved instances by degree and T (left) and the initial gap on growing GIRGs (right). The colors in the right plot indicate if an instance was solved by reductions only, needed branching, or remained unsolved in the given time limit. The left plot maps color and size to solved instances.

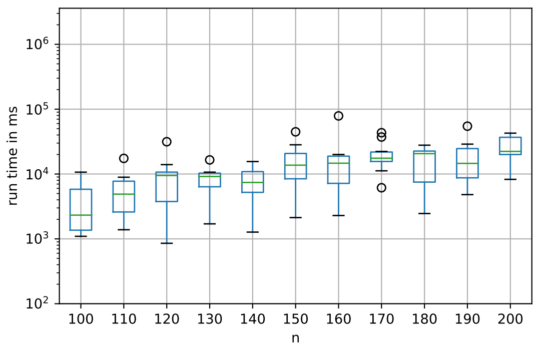
manageable with four of ten instances solved; high temperature (0.8) and low average degree (7) even more so with seven of ten instances solved. In contrast, the algorithm solved only 4 of the 80 instances with temperature at least 0.6 and average degree at least 10.

Graph Size. Figure 6.4b gives an overview of which instances could be solved with or without branching. The axes are the size of the graph and the initial gap between lower and upper bound. As expected, the instances that could not be solved have a higher gap. Compared to the PACE instances, fewer can be solved without branching and only one has matching initial upper and lower bounds. Nevertheless, 41 of these 110 instances are solved by reductions alone. An instance with only 140 vertices was not solved and has a substantially higher gap than the others of the same size. GIRGs with the same configuration can vary greatly in difficulty for our solver. Two instances with 190 nodes and three with 200 nodes are unsolved.

Figure 6.5a and 6.5b show the total run time of the solver and the time spent with initial reductions, respectively. Although the run time of the solver differs up to three orders of magnitude between instances of the same size, the median time to solve an instance grows from 100 to 140 vertices. After that, the growth becomes less pronounced such that the variance makes it hard to estimate a clear trend. Also the last two columns are biased because of the instances that timed out. We explain the high variance by the comparatively small range of n that can reliably be solved and the low number of samples. Thus, in the range from 150 to 200 vertices the random sampling of position and degree distribution affects the difficulty of the GIRG for our solver more than the size of the graph. Nevertheless, the time to compute the initial kernel grows steadily with increasing number of nodes (see Figure 6.5b).

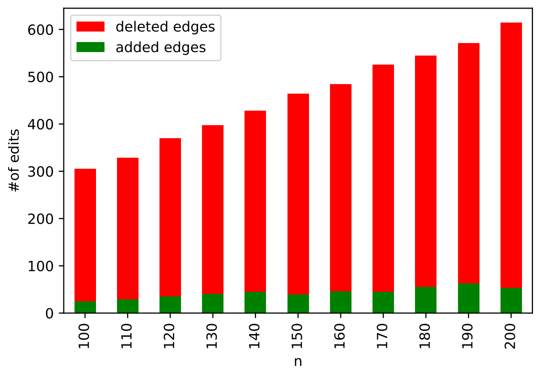


(a) Run time in milliseconds on solved instances.

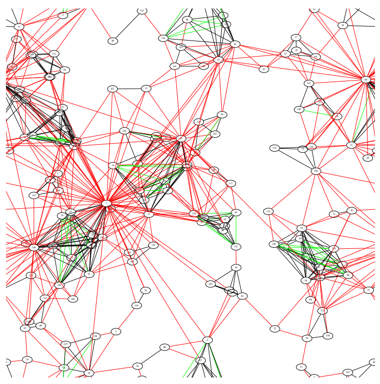


(b) Run time in milliseconds for the initial kernel.

Figure 6.5: The run time of the solver in total (left) and to compute the initial kernel (right).



(a) Average edit counts in an optimal solution.



(b) An optimal solution for a GIRG.

Figure 6.6: The number of edits in an optimal solution (left) and a possible optimal solution (right). The left plot shows the average over ten instances. The right GIRG was generated with default parameters and edits are indicated by color. Thus, the green edges are not in the generated GIRG.

Solution Structure on GIRGs. Figure 6.6b shows the edits of an optimal solution for a GIRG with the default parameters listed above. Note that the positions for the vertices are sampled in a unit torus where opposite borders are identified. Due to the scale-free degree distribution, some vertices have very high degree in the input instance. Most of those edges are deleted and the high-degree node is placed in the largest clique in close proximity. There are many small cliques in the resulting cluster graph and more edges are deleted than inserted. Figure 6.6a confirms this observation. The plot shows the number of deleted or added edges in an optimal solution over growing graph size. Each bar represents the average over all solved instances for this size. Due to the average degree of ten, the number of edges in the input is between 500 and 1000. The number of total edits grows approximately linear in the size of the graph and deletes about half of all edges. In fact, between 44 and 64 percent of the edges are deleted which seems to be independent of graph size. The number of inserted edges is comparatively low but also grows with growing number of vertices.

6.5 Conclusion

We present an exact branch-and-bound algorithm for the cluster editing problem. Moreover, we propose new reduction rules as well as formalize an improved technique to obtain lower bounds via subgraph packings, which contributes significantly to the success of the solver. We evaluate the lower bounds as well as various reductions rules on the instances of the 2021 PACE challenge. The lower bounds match the optimum on 79 of the 173 instances we were able to solve. For the reduction rules, by far the most effective ones are the rules that depend on lower bounds to identify forced choices, i.e., edge pairs that must or must not be edited in any optimal solution. They produce kernels with a small number of vertices and reduce k (the number of allowed edits) to an even greater extent. Combining all reductions used by the solver produces an empty kernel on more than 75% of all instances. We also investigate the effect of size, clustering, and density on our algorithm in a scale-free network model. While the size of the graph has a small effect on performance, the combination of high density and low clustering produces remarkably hard instances.

7 Conclusion and Outlook

In this thesis we explored the capabilities of the GIRG model in algorithm engineering. After implementing an efficient GIRG generator, we used it to engineer algorithms for maximum flow, minimum spanning arborescence, hitting set, and cluster editing. The GIRG model as well as the generator helped to inspire, design, evaluate, and better understand our algorithms on realistic networks. The following sections summarize the findings of the individual chapters and propose directions for future research before reaching a general conclusion in Section 7.6.

7.1 Generating GIRGs

The first step was to implement a generator that can efficiently produce instances of the model at the scale of large real-world networks, which the naive quadratic implementation is not capable of. Although a linear time algorithm was known for some time [BKL15; BKL19], its description is rather technical and the algorithm was never implemented before. This was partly due to HRGs, as a special-case of GIRGs, having a multitude of efficient generators (see Section 2.1.2). Unfortunately, most of these HRG generators are further restricted to the easier threshold case of the model ($T = 0$) and lack the capability to generate more difficult, noisy instances.

In Chapter 2, we refined and implemented the algorithm by Bringmann, Keusch, and Lengler [BKL19] to obtain the first subquadratic generator of the GIRG model. Our generator allows non-zero temperatures and its HRG special case outperforms all existing HRG generators in a sequential setting. The code supports parallelization. However, this was not the focus of our work and generators for HRGs with better scaling on multiple cores/compute nodes are available [Pen17; Fun+19].

In practice, a major open question was to determine the dependence of the input parameters of the model on the average degree of the resulting graph. All applications throughout this thesis need to fix the average degree in some way. For HRGs, there exists a method to calculate the input parameters to achieve a desired average degree asymptotically but this method becomes unreliable for small graphs and graphs with non-zero temperature. For example, to generate the HRGs visualized in Section 1.2.5, we input a desired average degree of 16 to obtain a graph with an actual average degree of 8. Furthermore, this method does not translate to the GIRG model. We determined the relation of the input parameters and average degree for the GIRG model, but it turned out to be difficult to solve analytically. Instead we provide an efficient estimation algorithm that produces accurate results and whose runtime is negligible in practice.

Using our generator and estimation algorithm, we were able to investigate the relation between HRGs and GIRGs. Since the theoretical inclusion of the HRG model in the GIRG framework is an asymptotic statement, the actual similarity of these models in practice was still an open question. We answered this question by experimentally showing that a large GIRG and HRG with matching parameters have more than 99% of their edges in common but the remaining difference does not fully converge to zero with a growing number of vertices.

For future work, further speeding up the generator appears unnecessary, because the current generator produces output faster than it can be written to disk. Instead, additional features like generating only a region of the ground space like an angular segment of a HRG are more valuable. Another contribution would be a better method to estimate the average degree of HRGs since the HRG variant of our generator currently relies on the imprecise estimation mentioned above.

7.2 Scale-Free Flow

In Chapter 3, we designed a flow algorithm specifically for scale-free networks. GIRGs played a crucial role for this algorithm, since the theory on the related HRG model provided the inspiration to use bidirectional breadth-first search. Moreover, GIRGs allowed a thorough evaluation that yielded evidence to support the claim of sublinear running time. In itself an algorithm with sublinear running time is only useful if it is executed on the same dataset multiple times since otherwise the time to read the input would dominate. Applications that require multiple cuts/flows in scale-free networks include the computation of a Gomory-Hu tree [GH61], clustering techniques [FTT04; LR04; Sch07; AL08; OZ14; VGM16; VKG19] as well as community detection and the study of community structure in general [FLG00; IK04; SJN06; Les+09]. For future work in this direction, it would be interesting to consider our experimental results from a theoretic point of view.

7.3 Minimum Spanning Arborescence

In Chapter 4, we conducted the first experimental evaluation of algorithms for the minimum spanning arborescence problem. To this end, we simplified and provided the first implementation of the GGST algorithm [Gab+86], which is the asymptotically fastest algorithm for the problem running in $O(n \log n + m)$. We also implemented efficient solvers based on Tarjan's algorithm [Tar77], which is at most a logarithmic factor slower than the GGST algorithm in theory.

An extensive evaluation on a large number of real world networks found that the overhead of the GGST algorithm still outweighs this logarithmic factor even after our optimizations and careful implementation. Surprisingly, we found that the version of Tarjan's algorithm that uses a min-heap is often the fastest even though it adds another logarithmic factor (on top of the previously mentioned) due to the missing decrease-key operation. While GGST was outperformed on real-world networks, we were able to produce realistic instances with the GIRG generator in a density regime were our GGST implementation beats the other available solvers.

The near-linear time of our implementations stands in contrast to the publicly available library solvers, which are Tarjan-based and take quadratic time in the worst case. Interestingly, the library solvers perform better than expected on many real-world instances and especially on GIRGs. We described and benchmarked a family of networks, which force them into their quadratic worst-case. For future work, it would be interesting to investigate why the quadratic worst-case of the library solvers does not occur for real-world networks. A nice result would be a parameter that is provably small on GIRGs and allows to describe the running time of these solvers more accurately.

Another direction is to simplify the GGST algorithm, which is still quite complex. A valuable contribution would be an algorithm that keeps the theoretical running time of the GGST algorithm (which is probably optimal) but with less overhead. In general, our work provides a baseline to evaluate and design arborescence algorithms in the future.

7.4 Hitting Set

In Chapter 5, we designed and evaluated a branch-and-bound algorithm for the hitting set problem. Unlike the maximum flow and minimum arborescence problems, hitting set is NP-hard. Thus our algorithm needs exponential time in the worst case. Nevertheless, it is able to solve large instances in reasonable time and outperforms ILP solvers, which are the state-of-the-art approach to solve hitting set.

The key contributing factor to its performance are lower bounds. We collected and formalized lower bounds used in the literature and established the complete inclusion hierarchy between them. One of the most effective techniques for obtaining lower bounds are packings. In this context, a packing describes a collection of sets to hit that are pairwise disjoint. Any valid solution must select at least as many elements as there are sets in the packing, because each element hits at most one set. Using these lower bounds and reduction rules based on them, the solver performs exceptionally good on real-world instances but struggles with random input.

To further investigate why realistic input is easier to solve, we again use GIRGs and our efficient generator developed in Chapter 2. At first glance, the hitting set problem seems unrelated to GIRGs because it is defined for a family of sets instead of graphs. However, an instance of hitting set forms a hypergraph and a hypergraph can be seen as a bipartite graph with the two sides being the hyperedges and vertices of the hypergraph. We adapted the GIRG generator to produce bipartite graphs and interpreted them as hypergraphs. A very similar setting was proposed to analyze the proof complexity of boolean satisfiability instances where clauses and variables form the bipartite incidence graph [Blä+21c]. The bipartite GIRG generator allowed us to further investigate the performance of the algorithm and we found that heterogeneity and locality are major contributors to the performance of the algorithm.

Theoretically analyzing the bipartite GIRG model seems a promising direction of future work. In particular, an estimation algorithm for the average degree would be desirable to avoid the currently used binary search. A different direction is to improve and broaden the scope of the solver. For example, a fast solver for weighted hitting set would be valuable in different contexts and could build on top of the findings of our unweighted solver.

7.5 Cluster Editing

The last problem we considered in this thesis is cluster editing. We provide a fast branch-and-bound algorithm that solves instances with hundreds of vertices in a few seconds. We use the packing technique — as in the hitting set solver — to obtain good lower bounds in practice. These lower bounds together with parameter-dependent reduction rules allow us to solve most of the instances of the 2021 PACE challenge without branching. To achieve these results, we generalized the packings to weighted instances and to packings of more elaborate structures. There is a tradeoff when choosing the complexity of the packed structures. The more complex the structure, the better the bound but also the harder it gets to compute a good packing. In our implementation, stars were most effective.

Using GIRGs we found that the time it takes the solver to process similar instances varies significantly and the size of the instance does not seem to affect this behaviour much. However, the amount of locality and the density of the graph drastically change the difficulty of an instance. GIRGs with little locality and a slightly raised average degree are almost impossible for the solver. Investigating further, we found that a solution on a GIRG needs to delete most of the edges and keeps only few larger clusters. This explains why higher density and missing locality makes the instances more difficult.

For future work, analyzing the packing technique would address multiple open questions. Even though these lower bounds are the most important part of our algorithm, it is still unclear if even the simplest packings can be computed in polynomial time since they reduce to an independent set problem on a restricted graph class. Both, a polynomial algorithm or a hardness result would certainly provide valuable insights. Even approximation results for different kinds of packings could directly translate to huge performance improvements of our solver. Another way to further improve the solver is to incorporate a recent technique for a reduction rule based on min-cuts [Sch+22].

7.6 Summary

To summarize our results from the four considered problems, we showed that the GIRG model can be useful in different contexts. GIRGs proved highly flexible and we used them for problems on weighted, directed, and bipartite graphs. On top of providing crucial benchmark instances for debugging and testing, GIRGs allowed us to obtain valuable insights. Most notably, we experimentally showed sublinear running time of our flow algorithm, investigated the solution structure of cluster editing, complemented our benchmark set of arborescence instances with a density for which there are no real-world networks available, and generated networks with adjustable locality and heterogeneity to reveal the effects of these properties on different algorithms. All this was done using our efficient generator without which, most of the experiments would not be possible.

Bibliography

[ABL09] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. "On the structure of industrial SAT instances." In: *International Conference on Principles and Practice of Constraint Programming.* Springer. 2009, pp. 127–141. DOI: 10.1007/978-3-642-04244-7_13.

[Abu10] Faisal N. Abu-Khzam. "A Kernelization Algorithm for d-Hitting Set." In: *Journal of Computer and System Sciences* 76.7 (2010), pp. 524–531. DOI: 10.1016/j.jcss.2009.09.002.

[AD85] Joachim H. Ahrens and Ulrich Dieter. "Sequential Random Sampling." In: *ACM Transactions on Mathematical Software* 11.2 (1985), pp. 157–169. DOI: 10.1145/214392.214402.

[Ahu+97] Ravindra K. Ahuja, Murali Kodialam, Ajay K. Mishra, and James B. Orlin. "Computational investigations of maximum flow algorithms." en. In: *European Journal of Operational Research* 97.3 (1997), pp. 509–542. ISSN: 0377-2217. DOI: 10.1016/S0377-2217(96)00269-X.

[AJB99] Réka Albert, Hawoong Jeong, and Albert-László Barabási. "Internet: Diameter of the world-wide web." In: *nature* 401.6749 (1999), p. 130.

[AL08] Reid Andersen and Kevin J. Lang. "An Algorithm for Improving Graph Partitions." In: *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms.* SODA '08. San Francisco, California: Society for Industrial and Applied Mathematics, 2008, pp. 651–660. URL: https://dl.acm.org/doi/10.5555/1347082.1347154.

[Ama+00] Luıs A Nunes Amaral, Antonio Scala, Marc Barthelemy, and H Eugene Stanley. "Classes of small-world networks." In: *Proceedings of the national academy of sciences* 97.21 (2000), pp. 11149–11152.

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows: Theory, algorithms and applications.* Prentice-Hall, Inc., 1993.

[Ans+16] Carlos Ansótegui, Maria Luisa Bonet, Jesús Giráldez-Cru, and Jordi Levy. "Community Structure in Industrial SAT Instances." In: *CoRR* abs/1606.03329 (2016).

[AOK15] Rodrigo Aldecoa, Chiara Orsini, and Dmitri Krioukov. "Hyperbolic Graph Generator." In: *Computer Physics Communications* 196 (2015), pp. 492–496. DOI: 10.1016/j.cpc.2015.05.028.

[ARW12] Diogo V. Andrade, Mauricio G. C. Resende, and Renato F. Werneck. “Fast local search for the maximum independent set problem.” In: *Journal of Heuristics* 18.4 (2012), pp. 525–547. DOI: `10.1007/s10732-012-9196-4`.

[BA99] Albert-László Barabási and Réka Albert. “Emergence of scaling in random networks.” In: *science* 286.5439 (1999), pp. 509–512.

[Bae18] Jeroen Baert. *Libmorton: C++ Morton Encoding/Decoding Library.* 2018. URL: `https://github.com/Forceflow/libmorton`.

[Bar+02] Albert-Laszlo Barabâsi, Hawoong Jeong, Zoltan Néda, Erzsebet Ravasz, Andras Schubert, and Tamas Vicsek. “Evolution of the social network of scientific collaborations.” In: *Physica A: Statistical mechanics and its applications* 311.3-4 (2002), pp. 590–614.

[Bar16] Albert-László Barabási. *Network science.* Cambridge university press, 2016.

[Bas+16] Lucas Bastos, Luiz Satoru Ochi, Fábio Protti, Anand Subramanian, Ivan César Martins, and Rian Gabriel S Pinheiro. “Efficient algorithms for cluster editing.” In: *Journal of Combinatorial Optimization* 31.1 (2016), pp. 347–371. DOI: `10.1007/s10878-014-9756-7`.

[BB06] Danielle Smith Bassett and ED Bullmore. “Small-world brain networks.” In: *The neuroscientist* 12.6 (2006), pp. 512–523.

[BBK11] Sebastian Böcker, Sebastian Briesemeister, and Gunnar W Klau. “Exact algorithms for cluster editing: Evaluation and experiments.” In: *Algorithmica* 60.2 (2011), pp. 316–334. DOI: `10.1007/s00453-009-9339-7`.

[BCH16] Michele Borassi, Pierluigi Crescenzi, and Michel Habib. “Into the Square: On the Complexity of Some Quadratic-time Solvable Problems.” In: *Electronic Notes in Theoretical Computer Science* 322 (2016), pp. 51–67. DOI: `10.1016/j.entcs.2016.03.005`.

[BF12] Ljiljana Brankovic and Henning Fernau. “Parameterized Approximation Algorithms for Hitting Set.” In: *Approximation and Online Algorithms (WAOA 2011).* 2012, pp. 63–76.

[BF22] Thomas Bläsius and Philipp Fischbeck. “On the External Validity of Average-Case Analyses of Graph Algorithms.” In: *European Symposium on Algorithms.* Vol. 244. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 21:1–21:14. DOI: `10.4230/LIPIcs.ESA.2022.21`.

[BFL14] Annika Baumann, Benjamin Fabian, and Matthias Lischke. “Exploring the Bitcoin Network.” In: *WEBIST (1).* 2014, pp. 369–374.

[BFW21] Thomas Bläsius, Tobias Friedrich, and Christopher Weyand. “Efficiently Computing Maximum Flows in Scale-Free Networks.” In: *European Symposium on Algorithms (ESA).* Vol. 204. 2021, 21:1–21:14. DOI: `10.4230/LIPIcs.ESA.2021.21`.

[Bia07] Ginestra Bianconi. "The entropy of randomized network ensembles." In: *EPL (Europhysics Letters)* 81.2 (Dec. 2007), p. 28005. DOI: 10.1209/0295-5075/81/28005.

[Bir+20] Johann Birnick, Thomas Bläsius, Tobias Friedrich, Felix Naumann, Thorsten Papenbrock, and Martin Schirneck. "Hitting set enumeration with partial information for unique column combination discovery." In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 2270–2283. DOI: 10.14778/3407790.3407824.

[BK04] Y. Boykov and V. Kolmogorov. "An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.9 (2004), pp. 1124–1137. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2004.60.

[BKL15] Karl Bringmann, Ralph Keusch, and Johannes Lengler. "Sampling Geometric Inhomogeneous Random Graphs in Linear Time." In: (Nov. 2, 2015). arXiv: 1511.00576 [cs.SI].

[BKL16] Karl Bringmann, Ralph Keusch, and Johannes Lengler. "Average Distance in a General Class of Scale-Free Networks with Underlying Geometry." In: (Feb. 18, 2016). arXiv: 1602.05712 [cs.DM].

[BKL19] Karl Bringmann, Ralph Keusch, and Johannes Lengler. "Geometric Inhomogeneous Random Graphs." In: *Theoretical Computer Science* 760 (2019), pp. 35–54. DOI: 10.1016/j.tcs.2018.08.014.

[BKW23] Maximilian Böther, Otto Kißig, and Christopher Weyand. "Efficiently Computing Directed Minimum Spanning Trees." In: *Symposium on Algorithm Engineering and Experiments (ALENEX)*. 2023.

[Blä+18a] Thomas Bläsius, Cedric Freiberger, Tobias Friedrich, Maximilian Katzmann, Felix Montenegro-Retana, and Marianne Thieffry. "Efficient Shortest Paths in Scale-Free Networks with Underlying Hyperbolic Geometry." In: *International Colloquium on Automata, Languages, and Programming (ICALP)*. Vol. 107. 2018, 20:1–20:14. DOI: 10.4230/LIPIcs.ICALP.2018.20.

[Blä+18b] Thomas Bläsius, Tobias Friedrich, Anton Krohmer, and Sören Laue. "Efficient Embedding of Scale-Free Graphs in the Hyperbolic Plane." In: *IEEE/ACM Transactions on Networking* 26.2 (2018), pp. 920–933. DOI: 10.1109/TNET.2018.2810186.

[Blä+19a] Thomas Bläsius, Philipp Fischbeck, Tobias Friedrich, and Martin Schirneck. "Understanding the Effectiveness of Data Reduction in Public Transportation Networks." In: *Algorithms and Models for the Web Graph*. Springer International Publishing, 2019, pp. 87–101. DOI: 10.1007/978-3-030-25070-6_7.

[Blä+19b] Thomas Bläsius, Tobias Friedrich, Maximilian Katzmann, Ulrich Meyer, Manuel Penschuck, and Christopher Weyand. "Efficiently Generating Geometric Inhomogeneous and Hyperbolic Random Graphs." In: *European Symposium on Algorithms (ESA)*. Vol. 144. 2019, 21:1–21:14. DOI: 10.4230/LIPIcs.ESA.2019.21.

[Blä+21a] Thomas Bläsius, Philipp Fischbeck, Lars Gottesbüren, Michael Hamann, Tobias Heuer, Jonas Spinner, Christopher Weyand, and Marcus Wilhelm. "PACE Solver Description: KaPoCE: A Heuristic Cluster Editing Algorithm." In: *International Symposium on Parameterized and Exact Computation (IPEC)*. 2021, 31:1–31:4. DOI: 10.4230/LIPIcs.IPEC.2021.31.

[Blä+21b] Thomas Bläsius, Philipp Fischbeck, Lars Gottesbüren, Michael Hamann, Tobias Heuer, Jonas Spinner, Christopher Weyand, and Marcus Wilhelm. "PACE Solver Description: The KaPoCE Exact Cluster Editing Algorithm." In: *International Symposium on Parameterized and Exact Computation (IPEC)*. 2021, 27:1–27:3. DOI: 10.4230/LIPIcs.IPEC.2021.27.

[Blä+21c] Thomas Bläsius, Tobias Friedrich, Andreas Göbel, Jordi Levy, and Ralf Rothenberger. "The Impact of Heterogeneity and Geometry on the Proof Complexity of Random Satisfiability." In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Society for Industrial and Applied Mathematics, Jan. 2021, pp. 42–53. DOI: 10.1137/1.9781611976465.4.

[Blä+22a] Thomas Bläsius, Philipp Fischbeck, Lars Gottesbüren, Michael Hamann, Tobias Heuer, Jonas Spinner, Christopher Weyand, and Marcus Wilhelm. "A Branch-And-Bound Algorithm for Cluster Editing." In: *Symposium on Experimental Algorithms (SEA)*. Vol. 233. 2022, 13:1–13:19. DOI: 10.4230/LIPIcs.SEA.2022.13.

[Blä+22b] Thomas Bläsius, Tobias Friedrich, Maximilian Katzmann, Ulrich Meyer, Manuel Penschuck, and Christopher Weyand. "Efficiently generating geometric inhomogeneous and hyperbolic random graphs." In: *Network Science* (Nov. 2022), pp. 1–20. DOI: 10.1017/nws.2022.32.

[Blä+22c] Thomas Bläsius, Tobias Friedrich, David Stangl, and Christopher Weyand. "An Efficient Branch-and-Bound Solver for Hitting Set." In: *Symposium on Algorithm Engineering and Experiments (ALENEX)*. 2022, pp. 209–220. DOI: 10.1137/1.9781611977042.17.

[BN16] Michele Borassi and Emanuele Natale. "KADABRA is an ADaptive Algorithm for Betweenness via Random Approximation." In: *24th Annual European Symposium on Algorithms (ESA 2016)*. Vol. 57. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 20:1–20:18. DOI: 10.4230/LIPIcs.ESA.2016.20.

[Boa18] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 5.0.* 2018. URL: `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf`.

[Böc+07] Sebastian Böcker, Sebastian Briesemeister, Quang Bao Anh Bui, and Anke Trub. "A fixed-parameter approach for weighted cluster editing." In: *Proceedings of the 6th Asia-Pacific Bioinformatics Conference.* 2007. DOI: `10.1142/9781848161092_0023`.

[Böc+09] Sebastian Böcker, Sebastian Briesemeister, Quang Bao Anh Bui, and Anke Truß. "Going weighted: Parameterized algorithms for cluster editing." In: *Theoretical Computer Science* 410.52 (2009), pp. 5467–5480. DOI: `10.1016/j.tcs.2009.05.006`.

[Böc12] Sebastian Böcker. "A golden ratio parameterized algorithm for cluster editing." In: *Journal of Discrete Algorithms* 16 (2012), pp. 79–89. DOI: `10.1016/j.jda.2012.04.005`.

[Boc71] F. C. Bock. "An algorithm to construct a minimum directed spanning tree in a directed network." In: *Developments in Operations Research* (1971).

[Boj14] Ćendić-Lazović Bojana. "A Genetic Algorithm for the Minimum Hitting Set." In: *Scientific Publications of the State University of Novi Pazar Series A: Applied Mathematics, Informatics and mechanics* 6.2 (2014), pp. 107–117.

[BPK10] Marián Boguná, Fragkiskos Papadopoulos, and Dmitri Krioukov. "Sustaining the internet with hyperbolic mapping." In: *Nature communications* 1 (2010), p. 62.

[BS20] René van Bevern and Pavel V. Smirnov. "Optimal-size problem kernels for d-Hitting Set in linear time and space." In: *Information Processing Letters* 163 (2020), p. 105998. DOI: `10.1016/j.ipl.2020.105998`.

[Car+15] Danilo Carastan-Santos, Raphael Yokoingawa De Camargo, David Correa Martins, Siang Wun Song, Luiz Carlos Silva Rozante, and Fabrizio Ferreira Borelli. "A Multi-GPU Hitting Set Algorithm for GRNs Inference." In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing.* 2015, pp. 313–322. DOI: `10.1109/CCGrid.2015.29`.

[Car+17] Danilo Carastan-Santos, Raphael Y. de Camargo, David C. Martins, Siang W. Song, and Luiz C.S. Rozante. "Finding Exact Hitting Set Solutions for Systems Biology Applications Using Heterogeneous GPU Clusters." In: *Future Generation Computer Systems* 67 (2017), pp. 418–429. DOI: `10.1016/j.future.2016.02.009`.

[CC12] Yixin Cao and Jianer Chen. "Cluster editing: Kernelization based on edge cuts." In: *Algorithmica* 64.1 (2012), pp. 152–169. DOI: `10.1007/s00453-011-9595-1`.

[CF06] Deepayan Chakrabarti and Christos Faloutsos. “Graph Mining: Laws, Generators, and Algorithms.” In: *ACM Comput. Surv.* 38.1 (2006). DOI: `10.1145/1132952.1132954`.

[CFM79] Paolo M. Camerini, Luigi Fratta, and Francesco Maffioli. “A note on finding optimum branchings.” In: *Networks* 9.4 (1979), pp. 309–312. DOI: `10.1002/net.3230090403`.

[CG97] B. V. Cherkassky and A. V. Goldberg. “On Implementing the Push—Relabel Method for the Maximum Flow Problem.” In: *Algorithmica* 19.4 (1997), pp. 390–410. DOI: `10.1007/pl00009180`.

[CH03] Reuven Cohen and Shlomo Havlin. “Scale-free networks are ultrasmall.” In: *Physical review letters* 90.5 (2003), p. 058701. DOI: `https://doi.org/10.1103/PhysRevLett.90.058701`.

[CH09] Bala G. Chandran and Dorit S. Hochbaum. “A Computational Study of the Pseudoflow and Push-Relabel Algorithms for the Maximum Flow Problem.” In: *Operations Research* 57.2 (2009), pp. 358–376. ISSN: 0030364X, 15265463.

[Cha+11] Karthekeyan Chandrasekaran, Richard Karp, Erick Moreno-Centeno, and Santosh Vempala. “Algorithms for Implicit Hitting Set Problems.” In: *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2011).* 2011, pp. 614–629. DOI: `10.1137/1.9781611973082.48`.

[Chu65] Yoeng-Jin Chu. “On the shortest arborescence of a directed graph.” In: *Scientia Sinica* 14 (1965).

[CKW10] Graham Cormode, Howard Karloff, and Anthony Wirth. “Set Cover Algorithms for Very Large Datasets.” In: *Proceedings of the 19th ACM International Conference on Information and Knowledge Management (CIKM 2010).* 2010, pp. 479–488. DOI: `10.1145/1871437.1871501`.

[CL02a] Fan Chung and Linyuan Lu. “Connected Components in Random Graphs with Given Expected Degree Sequences.” In: *Annals of Combinatorics* 6.2 (2002), pp. 125–145. DOI: `10.1007/PL00012580`.

[CL02b] Fan Chung and Linyuan Lu. “The Average Distances in Random Graphs with Given Expected Degrees.” In: *Proceedings of the National Academy of Sciences* 99.25 (2002), pp. 15879–15882. DOI: `10.1073/pnas.252631999`.

[CM12] Jianer Chen and Jie Meng. “A 2k kernel for the cluster editing problem.” In: *Journal of Computer and System Sciences* 78.1 (2012), pp. 211–220. DOI: `10.1016/j.jcss.2011.04.001`.

[Cor+09] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* MIT press, 2009.

[CTF00] Alberto Caprara, Paolo Toth, and Matteo Fischetti. “Algorithms for the Set Covering Problem.” In: *Annals of Operations Research* 98.1/4 (2000), pp. 353–371. DOI: `10.1023/a:1019225027893`.

[CV07] Kenneth L. Clarkson and Kasturi Varadarajan. “Improved Approximation Algorithms for Geometric Set Cover.” In: *Discrete & Computational Geometry* 37 (2007), pp. 43–58.

[Din70] Yefim Dinitz. “Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation.” In: *Soviet Mathematics Doklady* 11 (1970), pp. 1277–1280.

[DM89] U. Derigs and W. Meier. “Implementing Goldberg’s max-flow-algorithm — A computational investigation.” en. In: *Zeitschrift für Operations Research* 33.6 (1989), pp. 383–403. ISSN: 1432-5217. DOI: 10.1007/BF01415937. URL: https://doi.org/10.1007/BF01415937.

[DS14] Irit Dinur and David Steurer. “Analytical Approach to Parallel Repetition.” In: *Forty-Sixth Annual ACM Symposium on Theory of Computing (STOC 2014)*. 2014, pp. 624–633. DOI: 10.1145/2591796.2591884.

[Edm67] Jack Edmonds. “Optimum branchings.” In: *Journal of Research of the National Bureau of Standards* 71B.4 (1967), p. 233. DOI: 10.6028/jres.071b.032.

[EG17] Nicole Eikmeier and David F Gleich. “Revisiting Power-law Distributions in Spectra of Real World Networks.” In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM. 2017, pp. 817–826.

[ER59] Paul Erdős and Alfréd Rényi. “On random graphs, I.” In: *Publicationes Mathematicae (Debrecen)* 6 (1959), pp. 290–297.

[ET75] Shimon Even and R. Endre Tarjan. “Network Flow and Testing Graph Connectivity.” In: *SIAM Journal on Computing* 4.4 (1975), pp. 507–518. DOI: 10.1137/0204043.

[Fer06] Henning Fernau. “Parameterized Algorithms for Hitting Set: The Weighted Case.” In: *Algorithms and Complexity (CIAC 2006)*. 2006, pp. 332–343.

[FF56] L. R. Ford and D. R. Fulkerson. “Maximal Flow Through a Network.” In: *Canadian Journal of Mathematics* 8 (1956), pp. 399–404. DOI: 10.4153/CJM-1956-045-5.

[FK18] Tobias Friedrich and Anton Krohmer. “On the Diameter of Hyperbolic Random Graphs.” In: *SIAM Journal on Discrete Mathematics* 32.2 (2018), pp. 1314–1334. DOI: 10.1137/17M1123961.

[FLG00] Gary William Flake, Steve Lawrence, and C. Lee Giles. “Efficient identification of Web communities.” In: *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD ’00*. ACM Press, 2000. DOI: 10.1145/347090.347121.

[FO19] Orr Fischer and Rotem Oshman. “A Distributed Algorithm for Directed Minimum-Weight Spanning Tree.” In: *33rd International Symposium on Distributed Computing (DISC 2019).* Vol. 146. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 16:1–16:16. DOI: `10.4230/LIPIcs.DISC.2019.16`.

[FO21] Orr Fischer and Rotem Oshman. “A distributed algorithm for directed minimum-weight spanning tree.” In: *Distributed Computing* (June 2021). DOI: `10.1007/s00446-021-00398-3`.

[Fre+86] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. “The pairing heap: A new form of self-adjusting heap.” In: *Algorithmica* 1.1-4 (1986). DOI: `10.1007/bf01840439`.

[FT87] Michael L. Fredman and Robert E. Tarjan. “Fibonacci heaps and their uses in improved network optimization algorithms.” In: *Journal of the ACM* 34.3 (1987). DOI: `10.1145/28869.28874`.

[FTT04] Gary William Flake, Robert E. Tarjan, and Kostas Tsioutsiouliklis. “Graph Clustering and Minimum Cut Trees.” In: *Internet Mathematics* 1.4 (2004), pp. 385–408. DOI: `10.1080/15427951.2004.10129093`.

[Fun+18] Daniel Funke, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. “Communication-Free Massively Distributed Graph Generation.” In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS).* 2018, pp. 336–347. DOI: `10.1109/IPDPS.2018.00043`.

[Fun+19] Daniel Funke, Sebastian Lamm, Ulrich Meyer, Manuel Penschuck, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. “Communication-free massively distributed graph generation.” In: *Journal of Parallel and Distributed Computing* 131 (2019), pp. 200–217. DOI: `10.1016/j.jpdc.2019.03.011`.

[Gab+86] Harold N. Gabow, Zvi Galil, Thomas H. Spencer, and Robert E. Tarjan. “Efficient algorithms for finding minimum spanning trees in undirected and directed graphs.” In: *Combinatorica* 6.2 (1986). DOI: `10.1007/bf02579168`.

[Geo03] Leonidas Georgiadis. “Arborescence optimization problems solvable by Edmonds’ algorithm.” In: *Theoretical Computer Science* 301.1-3 (2003). DOI: `10.1016/s0304-3975(02)00888-5`.

[GF64] Bernard A. Galler and Michael J. Fisher. “An improved equivalence algorithm.” In: *Communications of the ACM* 7.5 (1964). DOI: `10.1145/364099.364331`.

[GH61] R. E. Gomory and T. C. Hu. “Multi-Terminal Network Flows.” In: *Journal of the Society for Industrial and Applied Mathematics* 9.4 (1961), pp. 551–570. ISSN: 0368-4245.

[GH85] Ronald L. Graham and Pavol Hell. “On the History of the Minimum Spanning Tree Problem.” In: *IEEE Annals of the History of Computing* 7.1 (1985). DOI: 10.1109/mahc.1985.10011.

[Gil59] E. N. Gilbert. “Random Graphs.” In: *The Annals of Mathematical Statistics* 30.4 (Dec. 1959), pp. 1141–1144.

[Gil61] Edgar N. Gilbert. “Random Plane Networks.” In: *Journal of the Society for Industrial and Applied Mathematics* 9.4 (1961), pp. 533–543. DOI: 10.1137/0109045.

[GL04] Diego Garlaschelli and Maria I. Loffredo. “Fitness-Dependent Topological Properties of the World Trade Web.” In: *Physical Review Letters* 93.18 (Oct. 2004), p. 188701. DOI: 10.1103/physrevlett.93.188701.

[GL08] Diego Garlaschelli and Maria I. Loffredo. “Maximum likelihood: Extracting unbiased information from complex networks.” In: *Physical Review E* 78.1 (July 2008), p. 015101. DOI: 10.1103/physreve.78.015101.

[Gol+11] Andrew V. Goldberg, Sagi Hed, Haim Kaplan, Robert E. Tarjan, and Renato F. Werneck. “Maximum Flows by Incremental Breadth-First Search.” en. In: *19th Annual European Symposium on Algorithms (ESA 2011).* Lecture Notes in Computer Science. Springer, 2011, pp. 457–468. ISBN: 9783642237195. DOI: 10.1007/978-3-642-23719-5_39.

[Got+20a] Lars Gottesbüren, Michael Hamann, Philipp Schoch, Ben Strasser, Dorothea Wagner, and Sven Zühlsdorf. “Engineering Exact Quasi-Threshold Editing.” In: *18th International Symposium on Experimental Algorithms (SEA 2020).* Vol. 160. Leibniz International Proceedings in Informatics (LIPIcs). 2020, 10:1–10:14. ISBN: 978-3-95977-148-1. DOI: 10.4230/LIPIcs.SEA.2020.10.

[Got+20b] Lars Gottesbüren, Michael Hamann, Philipp Schoch, Ben Strasser, Dorothea Wagner, and Sven Zühlsdorf. “Engineering Exact Quasi-Threshold Editing.” In: *18th International Symposium on Experimental Algorithms (SEA 2020).* Vol. 160. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 10:1–10:14. DOI: 10.4230/LIPIcs.SEA.2020.10.

[GPP12] Luca Gugelmann, Konstantinos Panagiotou, and Ueli Peter. “Random Hyperbolic Graphs: Degree Sequence and Clustering.” In: *International Colloquium on Automata, Languages, and Programming (ICALP).* 2012, pp. 573–585. DOI: 10.1007/978-3-642-31585-5_51.

[Gra73] Mark S. Granovetter. “The Strength of Weak Ties.” In: *American Journal of Sociology* 78.6 (1973), pp. 1360–1380.

[GS78] Leo J. Guibas and Robert Sedgewick. “A dichromatic framework for balanced trees.” In: *Proceedings of the Annual Symposium on Foundations of Computer Science (SFCS).* 1978. DOI: 10.1109/sfcs.1978.3.

[GT14] Andrew V. Goldberg and Robert E. Tarjan. “Efficient Maximum Flow Algorithms.” In: *Commun. ACM* 57.8 (2014), pp. 82–89. DOI: 10.1145/2628036.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. "A new approach to the maximum-flow problem." In: *Journal of the ACM* 35.4 (1988), pp. 921–940. ISSN: 0004-5411. DOI: `10.1145/48014.61051`.

[Guo09] Jiong Guo. "A more effective linear kernelization for cluster editing." In: *Theoretical Computer Science* 410.8-10 (2009), pp. 718–726. DOI: `10.1016/j.tcs.2008.10.021`.

[Gur21] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual.* 2021. URL: `https://www.gurobi.com`.

[Gus90] Dan Gusfield. "Very Simple Methods for All Pairs Network Flow Analysis." In: *SIAM Journal on Computing* 19.1 (1990), pp. 143–155. DOI: `10.1137/0219009`.

[GV17] Andrew Gainer-Dewar and Paola Vera-Licona. "The Minimal Hitting Set Generation Problem: Algorithms and Computation." In: *SIAM Journal on Discrete Mathematics* 31.1 (2017), pp. 63–100. DOI: `10.1137/15m1055024`.

[GW97] Tal Grossman and Avishai Wool. "Computational experience with approximation algorithms for the set covering problem." In: *European Journal of Operational Research* 101.1 (1997), pp. 81–92. DOI: `10.1016/s0377-2217(96)00161-0`.

[Han+17] Thomas Dueholm Hansen, Haim Kaplan, Robert E. Tarjan, and Uri Zwick. "Hollow Heaps." In: *Transactions on Algorithms* 13.3 (2017). DOI: `10.1145/3093240`.

[HH15] Sepp Hartung and Holger H. Hoos. "Programming by Optimisation Meets Parameterised Algorithmics: A Case Study for Cluster Editing." In: *Learning and Intelligent Optimization.* Springer, 2015, pp. 43–58. DOI: `10.1007/978-3-319-19084-6_5`.

[HLK18] Pim van der Hoorn, Gabor Lippner, and Dmitri Krioukov. "Sparse Maximum-Entropy Random Graphs with a Given Power-Law Degree Distribution." In: *Journal of Statistical Physics* (2018). DOI: `https://doi.org/10.1007/s10955-017-1887-7`.

[Hoc08] Dorit S. Hochbaum. "The Pseudoflow Algorithm: A New Algorithm for the Maximum-Flow Problem." In: *Operations Research* 56.4 (2008), pp. 992–1009. DOI: `10.1287/opre.1080.0524`.

[HYW11] Felix Halim, Roland H.C. Yap, and Yongzheng Wu. "A MapReduce-Based Maximum-Flow Algorithm for Large Small-World Network Graphs." In: *2011 31st International Conference on Distributed Computing Systems.* ISSN: 1063-6927. 2011, pp. 192–202. DOI: `10.1109/ICDCS.2011.62`.

[IK04] Noriko Imafuji and Masaru Kitsuregawa. "Finding Web Communities by Maximum Flow Algorithm Using Well-Assigned Edge Capacities." In: *IEICE Transactions* (2004).

[Int19] Intel. *Intel 64 and IA-32 Architectures Developer's Manual.* Intel Corporation. 2019.

[ISO20] ISO. *ISO/IEC 14882:2020 Programming languages — C++.* Sixth. International Organization for Standardization, 2020. URL: `https://www.iso.org/standard/79358.html`.

[ITK00] Trey E. Ideker, Vesteinn Thorsson, and Richard M. Karp. "Discovery of Regulatory Interactions Through Perturbation: Inference and Experimental Design." In: *Pacific Symposium on Biocomputing.* 2000, pp. 302–313.

[Jar30] Vojtěch Jarník. "O jistém problému minimálním [About a certain minimal problem]." In: *Práce moravské přírodovědecké společnosti* 4.6 (1930).

[Jom+10] Thibaut Jombart, Rosalind M. Eggo, Peter J. Dodd, and Francois Balloux. "Reconstructing disease outbreaks from genetic data: a graph approach." In: *Heredity* 106.2 (2010). DOI: `10.1038/hdy.2010.78`.

[Kam14] Naoyuki Kamiyama. "Arborescence Problems in Directed Graphs: Theorems and Algorithms." In: *Interdisciplinary Information Sciences* 20.1 (2014). DOI: `10.4036/iis.2014.51`.

[Kar71] Richard M. Karp. "A simple derivation of edmonds' algorithm for optimum branchings." In: *Networks* 1.3 (1971). DOI: `10.1002/net.3230010305`.

[Kar72] Richard M. Karp. "Reducibility among Combinatorial Problems." In: *Complexity of Computer Computations.* Springer US, 1972, pp. 85–103. DOI: `10.1007/978-1-4684-2001-2_9`.

[Kar73] Alexander V. Karzanov. "On finding a maximum flow in a network with special structure and some applications." In: *Matematicheskie Voprosy Upravleniya Proizvodstvom* 5 (1973), pp. 81–94.

[Kel+21] Leon Kellerhals, Tomohiro Koana, André Nichterlein, and Philipp Zschoche. "The PACE 2021 Parameterized Algorithms and Computational Experiments Challenge: Cluster Editing." In: *International Symposium on Parameterized and Exact Computation (IPEC).* 2021, 26:1–26:18. DOI: `10.4230/LIPIcs.IPEC.2021.26`.

[KKT09] Naoyuki Kamiyama, Naoki Katoh, and Atsushi Takizawa. "Arc-disjoint intrees in directed graphs." In: *Combinatorica* 29.2 (2009). DOI: `10.1007/s00493-009-2428-z`.

[Kle11] Johan de Kleer. "Hitting set algorithms for model-based diagnosis." In: *22nd International Workshop on Principles of Diagnosis.* 2011.

[KM86] Mirko Křivánek and Jaroslav Morávek. "NP-hard problems in hierarchical-tree clustering." In: *Acta informatica* 23.3 (1986), pp. 311–323. DOI: `10.1007/BF00289116`.

[Kri+10] Dmitri Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguñá. "Hyperbolic Geometry of Complex Networks." In: *Physical Review E* 82 (3 2010). DOI: `10.1103/physreve.82.036106`.

[Kru56] Joseph Bernard Kruskal. “On the shortest spanning subtree of a graph and the traveling salesman problem.” In: *Proceedings of the American Mathematical Society* 7.1 (1956). DOI: `10.1090/s0002-9939-1956-0078686-7`.

[Kun13] Jérôme Kunegis. “KONECT – The Koblenz Network Collection.” In: *Proc. Int. Conf. on World Wide Web Companion.* 2013, pp. 1343–1350. URL: `http://konect.cc/`.

[Lan04] Kevin Lang. *Finding Good Nearly Balanced Cuts in Power Law Graphs.* Tech. rep. YRL-2004-036. Yahoo! Research Labs, 2004.

[Les+09] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. “Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters.” In: *Internet Mathematics* 6.1 (2009), pp. 29–123. DOI: `10.1080/15427951.2009.10129177`.

[LF16] Matthias Lischke and Benjamin Fabian. “Analyzing the bitcoin network: The first four years.” In: *Future Internet* 8.1 (2016), p. 7.

[LK14] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection.* `http://snap.stanford.edu/data`. 2014.

[LM16] Moritz von Looz and Henning Meyerhenke. “Querying Probabilistic Neighborhoods in Spatial Data Sets Efficiently.” In: *International Workshop on Combinatorial Algorithms (IWOCA).* 2016, pp. 449–460. DOI: `10.1007/978-3-319-44543-4_35`.

[LMP15] Moritz von Looz, Henning Meyerhenke, and Roman Prutkin. “Generating Random Hyperbolic Graphs in Subquadratic Time.” In: *International Symposium on Algorithms and Computation (ISAAC).* 2015, pp. 467–478. DOI: `10.1007/978-3-662-48971-0_40`.

[Loo+16] Moritz von Looz, Mustafa Safa Özdayi, Sören Laue, and Henning Meyerhenke. “Generating Massive Complex Networks with Hyperbolic Geometry Faster in Practice.” In: *IEEE High Performance Extreme Computing Conference (HPEC).* 2016, pp. 1–6. DOI: `10.1109/HPEC.2016.7761644`.

[Loo19] Moritz von Looz. “High-Performance Graph Algorithms.” PhD thesis. Karlsruhe Institute of Technology (KIT), 2019. DOI: `10.5445/IR/1000095908`.

[Lov85] L. Lovasz. “Computing ears and branchings in parallel.” In: *26th Annual Symposium on Foundations of Computer Science (sfcs 1985).* 1985, pp. 464–467. DOI: `10.1109/SFCS.1985.16`.

[LR04] Kevin Lang and Satish Rao. “A Flow-Based Method for Improving the Expansion or Conductance of Graph Cuts.” In: *Integer Programming and Combinatorial Optimization.* Springer, 2004, pp. 325–337. DOI: `10.1007/978-3-540-25960-2_25`.

[Men+06] Ran Mendelson, Robert E. Tarjan, Mikkel Thorup, and Uri Zwick. “Melding priority queues.” In: *Transactions on Algorithms* 2.4 (2006). DOI: `10.1145/1198513.1198517`.

[Mil+07] Gerald A Miller, Yi Y Shi, Hong Qian, and Karol Bomsztyk. “Clustering coefficients of protein-protein interaction networks.” In: *Physical Review E* 75.5 (2007), p. 051910.

[Mil67] Stanley Milgram. “The small world problem.” In: *Psychology today* 2.1 (1967), pp. 60–67.

[MK13] Erick Moreno-Centeno and Richard M. Karp. “The Implicit Hitting Set Approach to Solve Combinatorial Optimization Problems with an Application to Multigenome Alignment.” In: *Operations Research* 61.2 (2013), pp. 453–468. DOI: `10.1287/opre.1120.1139`.

[MMO14] Eli A Meirom, Shie Mannor, and Ariel Orda. “Network formation games with heterogeneous players and the internet structure.” In: *Proceedings of the fifteenth ACM conference on Economics and computation*. ACM. 2014, pp. 735–752.

[Mor66] Guy M Morton. *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. Tech. rep. International Business Machines Company New York, 1966. URL: `https://domino.research.ibm.com/library/cyberdig.nsf/0/0dabf9473b9c86d48525779800566a39?OpenDocument`.

[MS02a] Sergei Maslov and Kim Sneppen. “Specificity and stability in topology of protein networks.” In: *Science* 296.5569 (2002), pp. 910–913.

[MS02b] Jose M Montoya and Ricard V Solé. “Small world patterns in food webs.” In: *Journal of theoretical biology* 214.3 (2002), pp. 405–412.

[MS17] Tobias Müller and Merlijn Staps. “The Diameter of KPKVB Random Graphs.” In: *CoRR* abs/1707.09555 (2017). arXiv: `1707.09555`. URL: `http://arxiv.org/abs/1707.09555`.

[MU14] Keisuke Murakami and Takeaki Uno. “Efficient algorithms for dualizing large-scale hypergraphs.” In: *Discrete Applied Mathematics* 170 (2014), pp. 83–94. DOI: `10.1016/j.dam.2014.01.012`.

[OM84] J. A. Orenstein and T. H. Merrett. “A Class of Data Structures for Associative Searching.” In: *ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*. 1984, pp. 181–190. DOI: `10.1145/588011.588037`.

[OZ14] Lorenzo Orecchia and Zeyuan Allen Zhu. “Flow-Based Algorithms for Local Graph Clustering.” In: *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 2014. DOI: `10.1137/1.9781611973402.94`.

[Pen03] Mathew Penrose. *Random geometric graphs*. Vol. 5. Oxford University Press, 2003.

[Pen17] Manuel Penschuck. “Generating Practical Random Hyperbolic Graphs in Near-Linear Time and with Sub-Linear Memory.” In: *International Symposium on Experimental Algorithms (SEA).* Vol. 75. 2017, 26:1–26:21. DOI: 10.4230/LIPIcs.SEA.2017.26.

[PN04] Juyong Park and M. E. J. Newman. “Statistical mechanics of networks.” In: *Physical Review E* 70.6 (Dec. 2004), p. 066117. DOI: 10.1103/physreve.70.066117.

[Pri57] Robert C. Prim. “Shortest Connection Networks And Some Generalizations.” In: *Bell System Technical Journal* 36.6 (1957). DOI: 10.1002/j.1538-7305.1957.tb01515.x.

[Pri76] Derek De Solla Price. “A general theory of bibliometric and other cumulative advantage processes.” In: *Journal of the American Society for Information Science* 27.5 (1976), pp. 292–306. DOI: https://doi.org/10.1002/asi.4630270505.

[RA15] Ryan A. Rossi and Nesreen K. Ahmed. “The Network Data Repository with Interactive Graph Analytics and Visualization.” In: *AAAI.* 2015. URL: http://networkrepository.com.

[Rah+07] Sven Rahmann, Tobias Wittkop, Jan Baumbach, Marcel Martin, Anke Truß, and Sebastian Böcker. “Exact and Heuristic Algorithms for Weighted Cluster Editing.” In: *Computational Systems Bioinformatics.* 2007. DOI: 10.1142/9781860948732_0040.

[Rei87] Raymond Reiter. “A theory of diagnosis from first principles.” In: *Artificial Intelligence* 32.1 (1987), pp. 57–95. DOI: 10.1016/0004-3702(87)90062-2.

[SA96] Raimund Seidel and Cecilia R. Aragon. “Randomized search trees.” In: *Algorithmica* 16.4-5 (1996). DOI: 10.1007/bf01940876.

[Sav13] Iztok Savnik. “Index Data Structure for Fast Subset and Superset Queries.” In: *Availability, Reliability, and Security in Information Systems and HCI.* Springer Berlin Heidelberg, 2013, pp. 134–148. DOI: 10.1007/978-3-642-40511-2_10.

[SBT04] Jeffrey R Sharom, David S Bellows, and Mike Tyers. “From large networks to small molecules.” In: *Current opinion in chemical biology* 8.1 (2004), pp. 81–90.

[SC10] Lei Shi and Xuan Cai. “An Exact Fast Algorithm for Minimum Hitting Set.” In: *2010 Third International Joint Conference on Computational Science and Optimization.* IEEE, 2010. DOI: 10.1109/cso.2010.240.

[Sch+22] Hjalmar Schulz, André Nichterlein, Rolf Niedermeier, and Christopher Weyand. “Applying a Cut-Based Data Reduction Rule for Weighted Cluster Editing in Polynomial Time.” In: *International Symposium on Parameterized and Exact Computation (IPEC).* 2022. DOI: 10.4230/LIPIcs.IPEC.2022.25.

[Sch07] Satu Elisa Schaeffer. “Graph clustering.” en. In: *Computer Science Review* 1.1 (2007), pp. 27–64. ISSN: 1574-0137. DOI: `10.1016/j.cosrev.2007.05.001`.

[Sch11] Boris Schäling. *The Boost C++ libraries.* 2011. URL: `https://theboostcpplibraries.com/`.

[Sco17] John Scott. *Social network analysis.* Sage, 2017.

[SG11] Tiziano Squartini and Diego Garlaschelli. “Analytical maximum-likelihood method to detect patterns in real networks.” In: *New Journal of Physics* 13.8 (Aug. 2011), p. 083001. DOI: `10.1088/1367-2630/13/8/083001`.

[SJN06] S.-W. Son, H. Jeong, and J. D. Noh. “Random field Ising model and community structure in complex networks.” In: *The European Physical Journal B* 50.3 (2006), pp. 431–437. DOI: `10.1140/epjb/e2006-00155-4`.

[Ski20] Steven S. Skiena. *The Algorithm Design Manual.* Springer International Publishing, 2020. DOI: `10.1007/978-3-030-54256-6`.

[Sla97] Petr Slavík. “A Tight Analysis of the Greedy Algorithm for Set Cover.” In: *Journal of Algorithms* 25.2 (1997), pp. 237–254. DOI: `10.1006/jagm.1997.0887`.

[Spo+04] Olaf Sporns, Dante R Chialvo, Marcus Kaiser, and Claus C Hilgetag. “Organization, development and function of complex brain networks.” In: *Trends in cognitive sciences* 8.9 (2004), pp. 418–425.

[SS20] Yaroslav V. Salii and Andrey S. Sheka. “Improving dynamic programming for travelling salesman with precedence constraints: parallel Morin–Marsten bounding.” In: *Optimization Methods and Software* (2020), pp. 1–27. DOI: `10.1080/10556788.2020.1817447`.

[SS84] Minsoo Suk and Ohyoung Song. “Curvilinear feature extraction using minimum spanning trees.” In: *Computer Vision, Graphics, and Image Processing* 26.3 (1984). DOI: `10.1016/0734-189x(84)90221-4`.

[ST86] Daniel Dominic Sleator and Robert E. Tarjan. “Self-Adjusting Heaps.” In: *Journal on Computing* 15.1 (1986). DOI: `10.1137/0215004`.

[Sta20] David Stangl. “Learning by Doing: Adapting Branching for Better Pruning in Search Trees.” MA thesis. Hasso Plattner Institute, 2020.

[Tar75] Robert E. Tarjan. “Efficiency of a Good But Not Linear Set Union Algorithm.” In: *Journal of the ACM* 22.2 (1975). DOI: `10.1145/321879.321884`.

[Tar77] Robert E. Tarjan. “Finding optimum branchings.” In: *Networks* 7.1 (1977). DOI: `10.1002/net.3230070103`.

[Teo17] Dusan Teodorovic. *Airline operations research.* Routledge, 2017.

[VB12] Tanmay Verma and Dhruv Batra. “MaxFlow Revisited: An Empirical Comparison of Maxflow Algorithms for Dense Vision Problems.” In: *Procedings of the British Machine Vision Conference 2012.* British Machine Vision Association, 2012. DOI: `10.5244/c.26.61`.

[VGM16] Nate Veldt, David F. Gleich, and Michael W. Mahoney. “A simple and strongly-local flow-based method for cut improvement.” In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. ICML’16. JMLR.org, 2016, pp. 1938–1947. URL: `http://proceedings.mlr.press/v48/veldt16.html` (visited on 03/16/2020).

[VKG19] Nate Veldt, Christine Klymko, and David F. Gleich. “Flow-Based Local Graph Clustering with Better Seed Set Inclusion.” In: *Proceedings of the 2019 SIAM International Conference on Data Mining*. Society for Industrial and Applied Mathematics, 2019, pp. 378–386. DOI: `10.1137/1.9781611975673.43`.

[VSH04] Vera Van Noort, Berend Snel, and Martijn A Huynen. “The yeast coexpression network has a small-world, scale-free architecture and can be explained by a simple model.” In: *EMBO reports* 5.3 (2004), pp. 280–284.

[Wei98] Karsten Weihe. “Covering trains by stations or the power of data reduction.” In: *Algorithms and Experiments, ALEX* (1998), pp. 1–8.

[WF01] Andreas Wagner and David A Fell. “The small world inside large metabolic networks.” In: *Proceedings of the Royal Society of London B: Biological Sciences* 268.1478 (2001), pp. 1803–1810.

[WF94] Stanley Wasserman and Katherine Faust. *Social network analysis: Methods and applications*. Vol. 8. Cambridge university press, 1994.

[WGS03] Ryan Williams, Carla P Gomes, and Bart Selman. “Backdoors to typical case complexity.” In: *IJCAI*. Vol. 3. Citeseer. 2003, pp. 1173–1178.

[WS98a] Duncan J Watts and Steven H Strogatz. “Collective dynamics of ‘small-world’networks.” In: *nature* 393.6684 (1998), p. 440.

[WS98b] Duncan J. Watts and Steven H. Strogatz. “Collective Dynamics of “Small-World” Networks.” In: *Nature* 393 (6684 1998), pp. 440–442. DOI: `10.1038/30918`.

[Yu+08] Shan Yu, Debin Huang, Wolf Singer, and Danko Nikolić. “A small world of neuronal synchrony.” In: *Cerebral cortex* 18.12 (2008), pp. 2891–2901.