Karlsruhe Institute of Technology

Sparsification for Linear Time Multilevel Graph Partitioning

Bachelor Thesis of
Dominik Lukas Rosch

At the Department of Computer Science
Institute of Theoretical Informatics (ITI)

Reviewers:	Prof. Dr. rer. nat. Peter Sanders
	T.T.-Prof. Dr. Thomas Bläsius
Advisors:	Lars Gottesbüren
	Nikolai Maas
	Daniel Seemaier

04.11.2024

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

Karlsruhe, den 4. November 2024

- -

(Dominik Lukas Rosch)

Abstract

An important NP-hard problem in computer science is partitioning a graph into k blocks of roughly equal size such that the weight of edges between those parts is minimized. It has many use cases in which linear time is desirable such as the distribution of work across computers. A practical meta heuristic for this problem is *multilevel graph partitioning (MGP)*. Current implementations of MGP do not have a linear run time, since a lot of levels with many edges are possible.

We show that by using algorithms from the field of sparsification we can control the number of edges and thus achieve MGP in linear time. Additionally, we discuss and implement several such algorithms—including the novel *weighted forest fire*—into the MGP program KaMinPar and thus construct a truly linear time graph partitioner. Our experimental evaluation shows that some of these algorithms only have small impacts on the overall quality of the computed partition and that they can lead to significant speedups on some graphs.

Zusammenfassung

Ein wichtiges NP-schweres Problem in der Informatik ist die Zerteilung eines Graphen in k ungefähr gleich große Blöcke, sodass das Gewicht von Kanten zwischen den Blöcken minimiert wird. Es hat viele Anwendungen in denen Linearzeit erstrebenswert ist, wie beispielsweise das Verteilen von Aufgaben auf Rechner. Eine praxistaugliche Metaheuristik für dieses Problem ist das *Multiebenen-graphpartitionieren (MGP)*. Aktuelle Implementierungen von MGP haben keine lineare Laufzeit, da viele Ebenen mit vielen Kanten auftreten können.

Wir zeigen, dass man mit Algorithmen aus dem Feld der Sparsification die Anzahl an Kanten kontrollieren kann und so MGP in Linearzeit erreicht. Außerdem diskutieren und implementieren wir eine Reihe solcher Algorithmen – auch das neue „weighted forest fire" – in das MGP-Programm KaMinPar und konstruieren damit einen Graphpartitionierer mit echter Linearzeit. Unsere experimentelle Auswertung zeigt, dass manche dieser Algorithmen der Qualität der berechneten Partition nur leicht schaden und dass sich die Geschwindigkeit bei machen Graphen sichtbar erhöht.

Contents

1 Introduction

Graphs are an important data structure with applications across numerous domains. They model various relations (called edges) between entries (called vertices) like physical connections, communication links or social relations.

One important problem when working with graphs is partitioning them into k parts of roughly equal size while minimizing the connections between these parts. This *(balanced) graph partitioning* has many useful applications such as distributing work across k computers in scientific computing. Here the vertices represent pieces of work and edges necessary communications. Thus, a good partition minimizes the time-intensive communication between machines. For such applications it would convenient to have fast—ideally linear time—algorithms. The issue is that graph partitioning is NP-hard [1]. Thus, we turn to heuristics.

A commonly used meta heuristic used for balanced graph partitioning is *multilevel graph partitioning (MGP)*. It first coarsens the input graph into smaller and smaller graphs called levels, then computes an initial partition on the smallest, and finally in a third step projects this partition onto the larger levels refining it along the way. The run time of this heuristic is dependent upon the number of vertices and edges across all these levels. This means that for linear time MGP the total number of vertices and edges needs to be linear in the number of vertices and edges of the input graph.

With just the standard coarsening algorithm of *size-constraint label propagation* the number of vertices does not shrink sufficiently to be linear across all levels. The reason is that it only contracts sets of connected vertices while keeping the contracted vertices within a weight limit. This has been solved by also contracting vertices with hop distance two (*two-hop clustering*).

But an additional problem remains. The coarsening does not always reduce the number of edges sufficiently. For example if pairs of vertices with disjoint neighborhoods are contracted all but the edges along which the contraction happen remain, e.g., in Figure 1. To solve this we turn to the field of *sparsification*, which is used to accelerate numerous numerical and graph algorithms such as solving symmetric, diagonally dominant systems [2], [3], or approximating minimum cuts [4], [5]. Sparsification is about approximating a dense graph with many edges by a graph with fewer edges. Surprisingly, every dense graph can be approximated by a sparse graph where the weight of edges cut by any partition is similar to its cut weight in the dense graph (Lemma 1 and [4], [6], [7]). Thus, when we use a good sparsification algorithm we loose only little partitioning quality.

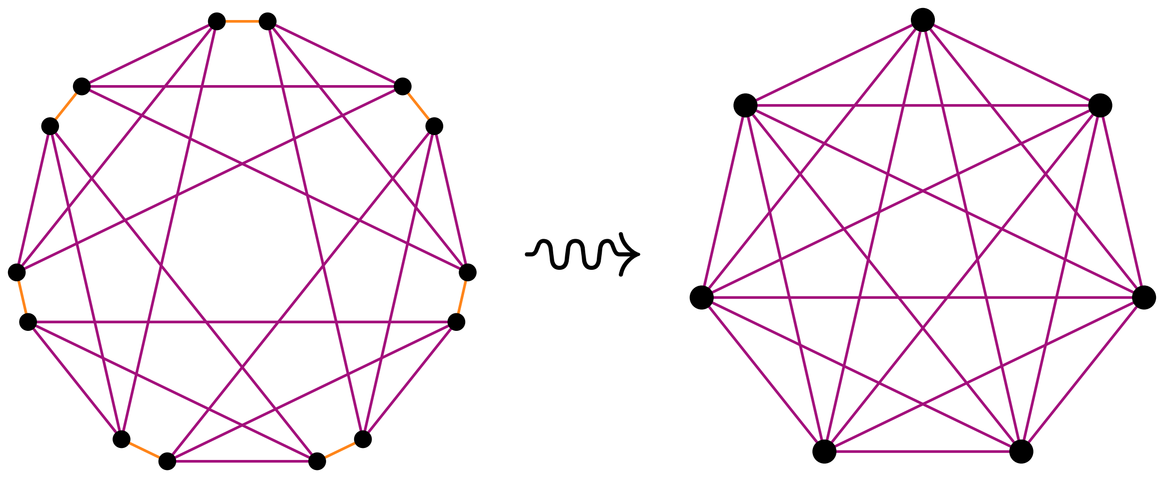

Figure 1: After the vertices are contracted along the orange edges, the $\binom{n/2}{2}$ other edges still remain.

1.1 Problem Statement

In this thesis we analyze whether we can achieve linear run time of multilevel graph partitioning through sparsification. Additionally, several sparsification algorithms should be implemented for the multilevel graph partitioner KaMinPar [8] and their impacts on the partition quality and run time should be examined.

1.2 Contribution

We prove that under certain conditions in MGP the total number of vertices across all levels is linear in the number of vertices of the input graph. If we additionally use a linear time sparsification algorithm to control the number of edges, we achieve MGP in linear time.

We implement several sparsification algorithms for KaMinPar and compare their partition quality. To make them comparable and to control the extent of sparsification we let them sparsify (at least approximately and in expectation) to a certain `target` number of edges. For this purpose this we develop procedures for calculating parameters of each algorithm such that this goal is reached. Additionally, we present the novel sparsification algorithms *weighted forest fire* and *unbiased threshold* as modification of previous ones.

After compiling a benchmark set of graph with a high density increase in over the course of the levels in KaMinPar, we evaluate the impact of these sparsification algorithm on the quality and time of KaMinPar.

1.3 Outline

First we define basic notations, the balanced graph partition problem and then discuss related work including multilevel graph partitioning. Afterwards, in Section 4, we take a look at different theoretical guarantees of sparsification including their relevance for graph partitioning. Then we prove with Theorem 1 that using sparsification MGP can run in linear time. In Section 6 we present several sparsification algorithms including the novel weighted forest fire and analyze how we can use their parameters to control the number of edges. These are evaluated in the next section, where we discuss the benchmark set and their impact on time and quality of KaMinPar. In the conclusion we summarize our results and discuss open questions which could be explored by future work.

2 Preliminaries

Unless stated otherwise, every graph $G = (V, E, c, w)$ is simple, undirected and weighted with positive vertex weights $c : V \to \mathbb{R}_{>0}$ and edge weights: $w : E \to \mathbb{R}_{>0}$. If the vertex weights are irrelevant we write $G = (V, E, w)$.

The number of vertices is denoted with $n := |V|$ and the number of edges with $m := |E|$. An edge between the vertices $u, v \in V$ is denoted as $\{u, v\}$ or uv and the neighbors of a vertex u with $N(u) := \{v \in V : uv \in E\}$. The number of neighbors of a vertex u is called its degree $\deg(u) = |N(u)|$. We fix orderings of the vertices $V = \{v_1, ..., v_n\}$ and the edges $E = \{e_1, ..., e_m\}$.

The weight of a set of edges $F \subseteq E$ is denoted with $w(F) = \sum_{e \in F} w(e)$ and the edges between k disjoint sets of vertices $V_1, ..., V_k \subseteq V$ with $E(V_1, ..., V_k) := \{uv \in E \mid i \neq j, u \in V_i, v \in V_j\}$. Finally, the weight of the edges between sets of vertices is denoted with $w(V_1, ..., V_k) := w(E(V_1, ..., V_k))$.

⬡ **Definition 1** (Cut): A *cut* is a set of vertices $S \subseteq V$. The weight of a cut S is

$$w_{\text{cut}}(S) := w(S, V \setminus S)$$

the weight of all edges between S and $V \setminus S$.

⬡ **Definition 2** (Laplacian): The *Laplacian matrix* $L \in \mathbb{R}^{n \times n}$ of a graph $G = (V, E, w)$ is defined as $L = D - A$ where A is the adjacency matrix and $D = \operatorname{diag}(w_{\text{cut}}(\{v_1\}), ..., w_{\text{cut}}(\{v_n\}))$ is a diagonal matrix with the sum of the weight of incident edges as its entries:

$$L_{ij} = \begin{cases} \sum_{x \in N(v_i)} w(v_i x) & \text{, if } i = j \\ -w(v_i v_j) & \text{, if } v_i v_j \in E \\ 0 & \text{, otherwise.} \end{cases}$$

The quadratic form of the Laplacian is $Q : \mathbb{R}^n \to \mathbb{R}; x \mapsto x^T L x$.

⬡ **Definition 3** (Graph Partitioning): *(Blanced) graph partitioning* is the problem of finding a partition $\Pi = \{V_1, ..., V_k\}$ of the vertex set V into k disjoint sets (blocks) such that the weight of edges between different sets $w(V_1, ..., V_k)$ is minimized while the weight of each block is no more than $\frac{1+\varepsilon}{k}$ times the weight of all vertices, with ε being the balance parameter.

$$\forall U \in \Pi : \sum_{u \in U} c(u) \leq \frac{1+\varepsilon}{k} \sum_{v \in V} c(v)$$

We call $w(V_1, ..., V_k)$ the *cut weight* of Π.

If we allow no imbalance, i.e., $\varepsilon = 0$, approximating balanced k-partitioning with any relative guarantee is NP-hard, which can be shown by reducing the strongly NP-complete problem 3-Partition to the graph partitioning problem [1]. Otherwise, if some imbalance is allowed, i.e., $\varepsilon > 0$, a polynomial time approximation to a factor of $\mathcal{O}(\log^2 n / \varepsilon^4)$ is possible [1].

3 Related Work

In this section, we introduce the multilevel graph partitioning (MGP) meta heuristic and look at a comparison of sparsification algorithms by Chen et al. [9]. This sets the stage for later sections, where we explore the application of sparsification to MGP.

3.1 Multilevel Graph Partitioning

Multilevel graph partitioning (MGP) is a strategy for graph partitioning (Definition 3). MGP consists of three phases: The *coarsening phase* recursively builds a hierarchy of smaller and smaller graphs until some threshold C is reached. In every step, it divides the vertices into clusters and contracts them to vertices of a new graph with edges between two clusters if any of their vertices are neighbors. The weight of a contracted vertex is the sum of the weights of all vertices in the corresponding cluster; the weight of a contracted edge is the sum of all edge weights between the corresponding clusters. The second phase, *initial partitioning*, starts once the number of vertices falls below the threshold C. It computes a partitioning of this coarsest graph. Finally, in the *refinement phase* the coarsening is undone: The partition is projected on the previous graph and gets refined locally until the initial graph is reached. [8]

MPG is advantageous since it computes a global solution with the initial partitioning using an algorithm which would be too expensive on the larger levels. In the third phase the refinement algorithm can escape local minima on the coarser levels while locally improving the solution on finer levels.

A modern scalable MGP Implementation is KaMinPar [8]. It is the one we work with in this thesis. In our case, the coarsening and the refinement phase of KaMinPar consists of *size-constraint label propagation* with the addition of *two-hop clustering* in the coarsening phase. *Size-constraint label propagation* was introduced by Meyerhenke et al. [10] based on the *label propagation* algorithm of Raghavan et al. [11]. It is parameterized with a maximal cluster weight U and round-based. In every round it visits the vertices in a fixed order.

At beginning of the coarsening, size-constraint label propagation assigns every vertex its own cluster. When it visits a vertex u, it moves u to the neighboring Cluster C to which u has the heaviest connection $w(\{u\}, C)$ without violating the size constraint $c(C) + c(u) \leq U$. It terminates once the clusters have converged or after some maximum number of rounds. [8]

The size-constraint label propagation does not always shrink the graphs sufficiently. This happens for example if a lot of vertices have a vertex of weight U as their only neighbor—a star subgraph with a heavy center. Then, all these vertices would just form singleton clusters leading to no contraction in this star subgraph. Thus, the *two-hop matching* algorithm from Metis [12] is adapted to merge vertices with hop distance two. During label propagation, it the algorithm computes *favorite clusters* of clusters; if a vertex cannot be added to any neighboring cluster without violating the constraint U, the highest rated cluster is saved as u's favorite cluster. If, after the size constraint label propagation terminates, the graph has shrunk by less than 50%, two-hop clustering starts: singleton clusters with the same favored cluster are merged without violating the weight constraint until the graph shrinks by at least 50% or no more two-hop clustering is possible. [8]

The size-constraint label propagation during the refinement works similarly. Initially, every vertex is assigned its partition block. Then, the vertices are moved to the block to which they have the heaviest connection without violating the size constraint, just like during the coarsening. This approach improves the solution locally. [8]

Note that the label propagation used during coarsening and refinement has a run time linear in the number of vertices and edges. Every vertex is visited no more than the maximum number of rounds and finding the cluster or block to which a vertex has the heaviest connection is linear in its degree.

In KaMinPar specifically, the maximum cluster weight is set to

$$U := \varepsilon \frac{c(V)}{k'} \qquad \text{with } k' := \min\{k, |V|/C\}$$

and ε being the imbalance parameter. Additionally, it clusters isolated vertices, i.e., vertices without neighbors, with each other [8].

3.2 Graphs Property Preservation

Chen et al. [9] compared sparsification algorithms empirically using a variety of metrics. One of these metrics—the preservation of the minimum weight of any cut—is related to graph partitioning, since the minimum cut is the optimal *unbalanced* partition into $k = 2$ blocks. Note that this is not the same as preserving the optimal *balanced* bipartition, but might still be an indicator.

This comparison highlights three algorithms especially effective at preserving the minimum cut weight: *k-Neighbor* sampling (KN, Section 6.5), *forest fire* (FF, Section 6.3.2) and sampling by *effective resistances* (ER-w/ER-uw for weighted and unweighted variants, Section 6.3.3). In Figure 2 these are compared to sampling the edges randomly with equal probabilities (RN, Section 6.1). [9]

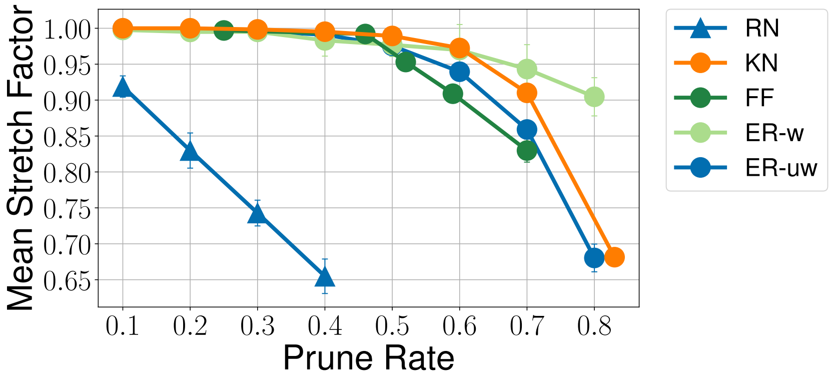

Figure 2: Comparison of how several sparsification algorithms preserve the min-cut at different amounts of sparsification (prune rates). Metric is the mean of by how much the min-cuts between 100, 000 source-destination pairs are distorted after sparsification. A mean stretch factor of 1 means no distortion. By Chen et al. [9].

Another metric is the preservation of the Laplacian quadratic form (Definition 2). As described in Section 4.2, this is sufficient for preserving the weight of every partition. Here sparsification by effective resistances (Section 6.3.3) emerged as the clear winner. [9]

4 Theory of Sparsification

Sparsification aims to approximate an arbitrary graph G with a sparser graph H, which is called a *sparsifier* of G [13]. More precisely we are interested in *edge sparsifiers*—graphs on the same vertex set, but with fewer edges [14].

Depending on context we are interested in the approximation of different properties. For example that the distance between every pair of vertices [13], the weight of cuts [4], [13], the degree distribution [9], [14], or the page ranks [9], [14] are similar.

4.1 Cut Sparsifiers

One formal notion of sparsifiers are *cut sparsifers*. They were first considered by Benczúr & Karger [4].

⬡ **Definition 4** (Cut Sparsifier): An ε-*cut sparsifier* of a graph $G = (V, E, w)$ is a graph $G' = (V, E', w')$ with $E' \subseteq E$ such that

$$\forall S \subseteq V : (1 - \varepsilon) w_{\text{cut}}(S) \leq w'_{\text{cut}}(S) \leq (1 + \varepsilon) w_{\text{cut}}(S).^1$$

For graph partitioning we are interested in preserving the cut weight of partitions. We can show that this is achieved by cut sparsifiers.

⬡ **Lemma 1**: An ε-cut sparsifer preserves the cut weight of any partition up to a factor of $(1 \pm \varepsilon)$

Proof of Lemma 1: The cut weight of a partition $\{V_1, ..., V_k\}$ of $G = (V, E, w)$ can be expressed as a sum of weights of cuts. Every edge $uv \in E(V_1, ..., V_k)$ with $u \in V_i, v \in V_j (i \neq j)$ is in the two cuts V_i and V_j. Therefore,

$$w(V_1, ..., V_k) = \frac{1}{2} \sum_{i=1}^{k} w_{\text{cut}}(V_i).$$

Thus, if $G' = (V, E', w')$ is an ε-cut sparsifier, it holds that

$$w'(V_1, ..., V_k) = \frac{1}{2} \sum_{i=1}^{k} w'_{\text{cut}}(V_i) \leq \frac{1+\varepsilon}{2} \sum_{i=1}^{k} w_{\text{cut}}(V_i) = (1 + \varepsilon) w(V_1, ..., V_k).$$

Analogously $(1 - \varepsilon) w(V_1, ..., V_k) \leq w'(V_1, ..., V_k)$. Therefore, the cut weight of a partition deviates at most by $(1 \pm \varepsilon)$ in an ε-cut-sparsifier compared to the original graph. □

It has been shown by Benczúr & Karger [4] that every graph has an ε-cut sparsifier with $\mathcal{O}\left(\frac{n \log n}{\varepsilon^2}\right)$ edges. This is remarkable since arbitrary graphs can have up to $\binom{n}{2} \in \Theta(n^2)$ edges and thus approximate partitioning on very dense graphs can be reduced to partitioning on much sparser graphs.

4.2 Spectral Sparsifiers

A stronger notion than cut sparsifiers are *spectral sparsifiers*, which are defined by a similar quadratic form of their Laplacians (Definition 2) and were introduced by D. A. Spielman & Teng [15].

⬡ **Definition 5** (Spectral Sparsifier): An ε-*spectral sparsifier* of a graph $G = (V, E, w)$ is a graph $G' = (V, E', w')$ with $E' \subseteq E$, such that the quadratic form of the Laplacian L' of G' approximates the quadratic form of the Laplacian L of G up to a factor of $(1 \pm \varepsilon)$:

$$\forall x \in \mathbb{R}^n : (1 - \varepsilon) x^T L x \leq x^T L' x \leq (1 + \varepsilon) x^T L x$$

[1]Sometimes, e.g. by Batson et al. [13], this condition is defined with $1/(1+\varepsilon)$ replacing $(1-\varepsilon)$. This is a somewhat stronger condition, since $\frac{1}{1+\varepsilon} = \frac{1-\varepsilon}{1-\varepsilon^2} > 1 - \varepsilon$, but makes no difference in our case.

⬡ **Lemma 2** (by D. A. Spielman & Teng [15]): Every ε-spectral sparsifier is an ε-cut sparsifier.

Proof of Lemma 2: Let $G' = (V, E', w')$ be an ε-spectral sparsifer of $G = (V, E, w)$ and let the quadratic forms of their Laplacians be denoted with Q' and $Q : \mathbb{R}^n \to \mathbb{R}$.

We can encode any arbitrary cut $S \subseteq V$ as a binary vector $x = (x_1, ..., x_n) \in \{0, 1\}^n$ with $x_i = 1$ if and only if $v_i \in S$. Then, since the Laplacian is $L = \text{diag}(w_{\text{cut}}(\{v_1\}), ..., w_{\text{cut}}(\{v_n\})) - A$ with A being the adjacency matrix, $Q(x)$ is the weight of the cut S:

$$\begin{aligned} Q(x) &= x^T L x \\ &= x^T \ \text{diag}(w_{\text{cut}}(\{v_1\}), ..., w_{\text{cut}}(\{v_n\}))x - x^T A x \\ &= \sum_{v \in S} w_{\text{cut}}(\{v\}) - \sum_{\substack{uv \in E \\ u,v \in S}} w(uv) \\ &= w_{\text{cut}}(S). \end{aligned}$$

Since G' is a spectral sparsifer of G the quadratic form of its Laplacian Q' approximates Q up to a factor of $(1 \pm \varepsilon)$. This implies that G' approximates the weight of cuts up to the same factor. Therefore, G' is also an ε-cut sparsifier by Definition 4. □

The combination of Lemma 1 and Lemma 2 means that spectral sparsifiers also preserve the cut weight of each partition.

5 Linear Time Multilevel Graph Partitioning through Sparsification

We can use sparsification to achieve MGP with a run time linear in the number of edges and vertices, provided that the coarsening and refinement on each level is in linear time. Or formally, that the time of the i-th level is in $\mathcal{O}(n_i + m_i)$ with n_i and m_i being the number of vertices and edges on the i-th of the L levels. The initial partition is only applied to graphs where the number of vertices falls below a certain threshold C and thus its time is bounded by a constant.

To show the linear time of the MGP we first look at the number of vertices over the course of the levels:

⬡ **Lemma 3**: When size-constraint label propagation with two-hop clustering and maximal cluster size of U is applied to a graph $G = (V, E, c, w)$ the number of computed clusters $\mathcal{C}$ is bounded by

$$|\mathcal{C}| \leq \frac{2}{3}|V| + \frac{4}{3}\frac{c(V)}{U} + 1$$

if isolated vertices are clustered with each other.

Proof of Lemma 3: If the two-hop clustering terminates early because the number of vertices has shrunk by 50%, the lemma holds. Otherwise, the two-hop clustering runs until all possible vertices are clustered.

Recall that the key parameter of the size-constraint label propagation with two-hop clustering described in Section 3.1 is the maximum cluster weight U.

First we define vertices with at least half of the maximum cluster weight U as heavy:

$$V_{\text{heavy}} := \left\{v \in V : c(v) \geq \frac{U}{2}\right\}$$

We can bound the size of V_{heavy} by the total weight of all vertices $c(V)$:

$$c(V) \geq c(V_{\text{heavy}}) \geq \frac{U}{2}|V_{\text{heavy}}| \Longrightarrow 2\frac{c(V)}{U} \geq |V_{\text{heavy}}|. \tag{I}$$

Let $\mathcal{C}$ be the set of clusters computed. We define the set of singleton, light, and non-isolated clusters as

$$\mathcal{C}_1 := \left\{K \in \mathcal{C} : K = \{v\} \wedge c(v) < \frac{U}{2} \wedge \deg(v) \geq 1\right\}.$$

The favorite of a cluster in $K_1 \in \mathcal{C}_1$ cannot be a cluster $K_2 \in \mathcal{C}_1$, since they would have been merged by the size-constraint label propagation because $c(K_1) + c(K_2) < U$. Additionally, no two clusters in $\mathcal{C}_1$ can share the same favorite cluster, since they would have been merged by the two-hop clustering. Therefore, all the favorites of $\mathcal{C}_1$ are distinct clusters in

$$\mathcal{C}_2 := \left\{K \in \mathcal{C} : |K| \geq 2 \vee c(K) \geq \frac{U}{2}\right\}$$

and by the pigeonhole principle this means $|\mathcal{C}_1| \leq |\mathcal{C}_2|$ and the ratio between the sizes

$$\rho := |\mathcal{C}_1| / |\mathcal{C}_2| \in [0, 1] \tag{II}$$

is at most 1.

The set of remaining clusters $\mathcal{C}_3$ only contains singleton, light, and isolated clusters:

$$\mathcal{C}_3 := \mathcal{C} \setminus (\mathcal{C}_1 \cup \mathcal{C}_2) = \left\{ K \in \mathcal{C} : K = \{v\} \wedge c(v) < \frac{U}{2} \wedge \deg(v) = 0 \right\}.$$

Since isolated vertices are clustered together, if there were two clusters in $\mathcal{C}_3$ they would get merged and therefore $|\mathcal{C}_3| \leq 1$.

Next we bound the number of vertices $|\bigcup \mathcal{C}_2|$ in the $\mathcal{C}_2$ clusters. A cluster $K \in \mathcal{C}_2$ contains at least two vertices or one heavy vertex $v \in V_{\text{heavy}}$. Thus, with the bound of Equation I:

$$\begin{aligned} \left|\bigcup \mathcal{C}_2\right| &= \sum_{K \in \mathcal{C}_2} |K| \geq 2|\mathcal{C}_2| - |V_{\text{heavy}}| \\ &\overset{I}{\geq} 2|\mathcal{C}_2| - 2\frac{c(V)}{U} \end{aligned} \tag{III}$$

We can use this to find a relation between n and the size of $\mathcal{C}_2$:

$$\begin{aligned} n &= \left|\bigcup \mathcal{C}\right| = |\mathcal{C}_1| + \left|\bigcup \mathcal{C}_2\right| + |\mathcal{C}_3| \\ &\overset{\text{III}}{\geq} |\mathcal{C}_1| + 2|\mathcal{C}_2| - 2\frac{c(V)}{U} \overset{\text{II}}{=} (2+\rho)|\mathcal{C}_2| - 2\frac{c(V)}{U} \end{aligned}$$

Rearranging the terms yields

$$|\mathcal{C}_2| \leq \frac{1}{2+\rho}\left(n + 2\frac{c(V)}{U}\right)$$

and because $|\mathcal{C}| = |\mathcal{C}_1| + |\mathcal{C}_2| + |\mathcal{C}_3| \leq (1+\rho)|\mathcal{C}_2| + 1$:

$$|\mathcal{C}| \leq \frac{1+\rho}{2+\rho}\left(n + 2\frac{c(V)}{U}\right) + 1 \overset{(\star)}{\leq} \frac{2}{3}\left(n + 2\frac{c(V)}{U}\right) + 1 = \frac{2}{3}n + \frac{4}{3}\frac{c(V)}{U} + 1$$

Inequality $(\star)$ holds because of Equation II and $\max_{\rho \in [0,1]} \frac{1+\rho}{2+\rho} = \frac{1+1}{2+1}$.[2] □

⬡ **Corollary 1**: The total number of vertices across all levels is in $\Theta(n)$, if

1. the levels are computed with size-constraint label propagation with two-hop clustering where isolated vertices are clustered together,
2. and on every level the maximal cluster size U_i is bounded by $U_i \geq \Gamma / n_i$ with $\Gamma > 4c(V)$,

Proof of Corollary 1: On every level $i \in \{1, ..., L-1\}$ from the first to the second-to-last the number of the vertices on the next level n_{i+1} is equal to the number of clusters $\mathcal{C}_i$ computed. Using the bound of Lemma 3 we see

$$n_{i+1} = |\mathcal{C}_i| \leq \frac{2}{3}n_i + \frac{4}{3}\frac{c(V)}{U_i} + 1 \leq \underbrace{\frac{2}{3}\left(1 + 2\frac{c(V)}{\Gamma}\right)}_{=:r} \cdot n_i + 1 = rn_i + 1 \tag{IV}$$

Because $\Gamma > 4c(V)$ holds:

$$r := \frac{2}{3}\left(1 + 2\frac{c(V)}{\Gamma}\right) < \frac{2}{3}\left(1 + 2\frac{c(V)}{4c(V)}\right) = \frac{2}{3}\left(1 + \frac{1}{2}\right) = 1.$$

[2]This can be seen with $\frac{d}{d\rho}\frac{1+\rho}{2+\rho} = \frac{(1+\rho)2 - 1(2+\rho)}{(2+\rho)^2} = \frac{\rho}{(2+\rho)^2} > 0$ for $\rho \in (0,1]$ or intuitively, adding the same amount to a large and a small quantity makes a bigger difference for the small quantity.

Thus, with Equation IV

$$\forall i \in \{1, ..., L\} : n_i \leq r^{i-1}n + \sum_{\ell=0}^{i-2} r^{\ell} \leq r^{i-1}n + \mathcal{O}(1)$$

with $r \in (0, 1)$ since $n_1 = n$. Therefore, the total number of vertices across all levels is

$$\sum_{i=1}^{L} n_i \leq \sum_{i=1}^{L} r^{i-1}n + \mathcal{O}(L) = \Theta(n).$$

(The number of levels L is in $\mathcal{O}(n)$ since every level has fewer vertices than the previous.) □

This result can be combined with sparsification to yield:

⬢ **Theorem 1**: Multi level graph partitioning with sparsification runs in linear time if

1. the sparsification algorithm runs in linear time and can be used for a reduction of the number of edges by at least a factor of $s \in (0, 1)$;
2. the time coarsening and refinement on every level is in linear time $\mathcal{O}(n_i + m_i)$;
3. the coarsening phase uses size-constraint label propagation with two-hop clustering where the maximal cluster weights on each level U_i are bound by $U_i \geq \Gamma / n_i$ with $\Gamma > 4c(V)$ and isolated vertices clustered with each other.

Proof of Theorem 1: The run time of the initial partitioning on the last level is bounded by a constant since it is only applied to graphs at most C vertices. All other levels have linear time in $\mathcal{O}(n_i + m_i)$, since the sparsification, coarsening and refinement all have linear time by the premises.

Let $\mathcal{S}$ be a linear time sparsification algorithm. We can use $\mathcal{S}$ to reduce the number of edges at the beginning of each level by at least a constant factor $s \in (0, 1)$ through sparsification, i.e.,

$$m_{i+1} \leq s m_i \leq s^{i+1} m.$$

Using edge sparsifiers, i.e. just removing edges, this leaves the multilevel graph partitioning algorithm intact: the projection between the levels only depends on the vertices.

By Corollary 1 and the third premise we know that the total number of vertices is in $\Theta(n)$. Thus, the total time is linear since it is dominated by the first level:

$$\sum_{i=0}^{L} \mathcal{O}(n_i + m_i) \subseteq \mathcal{O}(n) + \mathcal{O}\left(m \sum_{i=0}^{L} s^i\right) \overset{s<1}{=} \mathcal{O}(n + m).$$

□

Finally, we can apply this result to KaMinPar:

⬡ **Corollary 2**: Using a linear time sparsification algorithm we achieve linear time multilevel graph partitioning with KaMinPar using label propagation refinement if $\varepsilon C \geq \gamma$ with $\gamma > 4$.

Proof of Corollary 2: Recall from Section 3.1 that KaMinPar uses size-constraint label propagation with two-hop clustering for coarsening and its refinement is also label propagation–based. Thus, coarsening and refinement of KaMinPar take linear time on every level. Additionally, isolated vertices are contracted with each other and cluster size constraint of the i-th level is $U_i := \varepsilon \frac{c(V)}{k'}$ with $k' := \min\left\{k, \frac{n_i}{C}\right\}$. Since $\varepsilon C \geq \gamma$ and $\gamma > 4$ the condition of Theorem 1 holds with $\Gamma := \gamma c(V)$:

$$U_i = \varepsilon \frac{c(V)}{k'} \geq \varepsilon \frac{c(V)}{n_i / C} = \varepsilon C \frac{c(V)}{n_i} \geq \gamma c(V) / n_i = \Gamma / n_i.$$

Therefore, by Theorem 1 KaMinPar has linear run time. □

This is possible with any linear-time sparsification algorithm. But what we have not looked at for now is how the sparsification impacts the quality of the computed partition. Ideally, the same partitions should be good solutions to graph partitioning after the sparsification as before.

Some sparsification algorithms are ε-cut-sparsifiers from Definition 4 which would guarantee us that each partition differs by at most a factor of $(1 \pm \varepsilon)$. But these have non-linear run time and maybe other heuristics have similar or better quality when used in MGP.

6 Sparsification Algorithms

A *sparsification algorithm* computes a sparsifier by *sampling* a subset of edges to include and removing all others. In this sparsifier similar partitions, as in the original graph, should have the minimum weight. This way a graph partitioning solution calculated in the sparsifier would translate well to the original graph. Ideally, it should have linear run time.

To facilitate a fair comparison of their quality, all algorithms—at least roughly or in expectation—should produce sparsifiers where the number of edges is equal to a parameter `target`. Then, we can set as a parameter, that we do not want the density between levels to increase by more than a factor of d. To enforce this, we can let the sparsification algorithm sparsify to the appropriate number of edges by setting

$$\texttt{target} = dm_i \frac{n_{i+1}}{n_i}$$

where n_i and m_i are the number of edges and vertices in the previous level and the n_{i+1} the number of vertices in the next.

6.1 Uniform Random

The simplest sparsification algorithm is sampling the edges independently and randomly with equal probability p [9]. This does not take the weights or the structure of the graph into account, but can still be used as a baseline for comparison.

To produce a sparsifier where the number of edges is equal to `target` in expectation we can choose

$$p = \frac{\texttt{target}}{m}.$$

6.2 Score-Based Sparsification

Most sparsification methods work by associating each edge with a *score* $s : E \to \mathbb{R}_{\geq 0}$ representing roughly its importance for partitioning with higher scores indicating higher importance. There are several options for such scores, e.g., the weight of the edges or interpreting the graph as an electrical network and looking at how much current each edge conducts. Additionally, there are multiple algorithms for using these scores to compute a sparsifier.

6.2.1 Threshold

Threshold sparsification works by deleting every edge e with a score $s(e)$ smaller than a threshold T. It is the basis for all algorithms presented by Lindner et al. [14].

The algorithm can sample exactly `target` edges by choosing T as the `target`-th largest score. This T can be found through linear time quickselect [16], [17]. It then samples every edge with a score greater than T counting how many get sampled. Then, it samples the number of edges remaining from edges with score at T.

6.2.2 Random with Replacement

When sampling *randomly with replacement* in each of q draws an edge is chosen randomly with a probability proportional to its score, i.e., edge $e \in E$ is drawn with probability

$$p_e = \frac{s(e)}{\sum_{e' \in E} s(e')}$$

and added to the sparsifier with weight $\frac{w(E)}{q}$. If an edge is sampled again the weights are added up.

This approach is used in a variety of sparsification algorithms [6], [7]. Notably, it is used for spectral sparsification with effective resistances by Spielman & Srivastava [6].

It can be efficiently implemented in linear time using the *alias method* [18], [19]. However, the number of edges is tricky to control.

6.2.3 Independent Random

The *Independent Random* (*IR*) sparsification algorithm samples every edge e independently with probability $p_e = \min\{1, C \cdot s(e)\}$, where $s(e)$ is the edges score and C is a normalization factor such that number of edges in the sparsifier m' is equal to `target` in expectation, i.e.,

$$\begin{gathered}\texttt{target} \overset{!}{=} M(C) := \mathbb{E}[m'] = \sum_{e \in E} p_e = |L_C| + C \sum_{e \in S_C} s(e) \\ \text{with } L_C := \{e \in E : 1 \leq Cs(e)\} \text{ and } S_C := E \setminus L_C.\end{gathered} \tag{V}$$

To calculate C we first approximate it with an interval and then we can easily calculate it directly in said interval. In the first step we binary search over the reciprocals of the scores

$$\Omega = \left\{\frac{1}{x} : x \in \operatorname{Image}(s)\right\}$$

to find the smallest $\omega_R \in \Omega$ such that $\texttt{target} \leq M(\omega_R)$. Let $\omega_L := \max\{\omega \in \Omega : \omega < \omega_R\}$ be the largest element in Ω smaller than ω_R. By definition of ω_R the normalization factor C is in the interval $(\omega_L, \omega_R]$. Since ω_L and ω_R are reciprocals of consecutive scores, for all $C \in (\omega_L, \omega_R]$

$$L_C = \left\{e \in E : \frac{1}{s(e)} \leq C\right\} = \left\{e \in E : \frac{1}{s(e)} \leq \omega_R\right\} = L_{\omega_R}$$

and $S_C = E \setminus L_{\omega_R}$. Thus, we can calculate C as

$$C = \frac{\texttt{target} - \left|L_{\omega_R}\right|}{\sum_{e \in S_{\omega_R}} s(e)}$$

which solves Equation V.

The parallelization and linear time implementation of the sampling itself—once the normalization factor C has been calculated—is rather simple since it is independent of every edge. Thus, the main difficulty lies in the calculation of the normalization factor.

To find ω_R we need a fast way to calculate the size of L_ω and the weight of all edges in S_ω. This can be achieved through some precalculations. We sort the scores $\operatorname{Image}(s)$ and calculate the prefix sum of these sorted scores. Thus, if the last smaller than $1/\omega$ is the i-th in the sorted scores, the size of L_ω is $n - i$ and the weight of S_ω is the i-th prefix sum. Because of the sorting necessary this takes $\Theta(n \log n)$ time.

In special cases, e.g., if the scores are 32-bit integers, we can only approximately sort the score into a small number of *exponential buckets* where the i-th bucket contains all scores in $[2^{i-1}, 2^i)$. Then, if $1/\omega$ fits into the i-th bucket, we can approximate the size of L_ω with the number of elements of the higher order buckets and the weight of S_ω with the weight of all edges in all buckets up to the i-th. This accelerates the calculation of the normalization factor by replacing the sorting with the linear time sorting into buckets, which can also be easily parallelized.

Both approaches do not guarantee that the number of edges in the sparsifier are equal to `target`. The goal is only to choose the normalization factor such that it is equal to `target` in expectation. This is achieved by the first while exponential bucket algorithm is only an approximation. Elements in the i-th buckets might belong in L_ω or S_ω and thus are not always treated correctly by this approximation, but if not a lot of the scores fall into the same bucket as $1/\omega$, the approximation is close to the optimal normalization factor.

6.3 Score Functions

A *score function* $s : E \to \mathbb{R}_{\geq 0}$ should roughly approximate each edge's importance for the partitioning.

The importance of an edge depends both on its weight, as well as on its position within the structure of the graph. It is trivial that heavier edges are more important. The importance of structure can be seen in the simple example of two cliques connected by a bridge b were all edges have equal weight as in Figure 3. An edge e in one of the cliques is less imported than b since the many other edges in the clique share the same role as e.

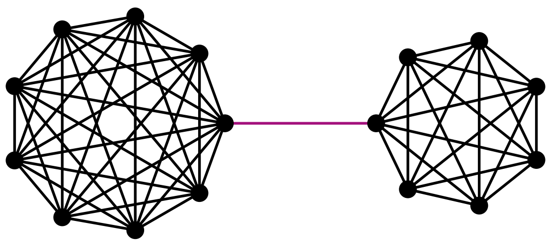

Figure 3: Two cliques connected by a bridge.

6.3.1 Weight

The simplest option is to just use the edge's weight as its score $s(e) := w(e)$. This ignores the structure of the graph, but might be sufficient in praxis.

6.3.2 (Weighted) Forest Fire

Forest Fire (FF) was introduced by Leskovec et al. [20] and adapted into an edge score by Lindner et al. [14]. It is based on an analogy of forest fires burning across edges to ignite new vertices. The idea is that the more often an edge burn the more important it is. Thus, we use how often an edge was burned as its score.

Forest Fire ignores the edge weights. Additionally, we present a novel variant *Weighted Forest Fire (WFF)* in which the fire is more likely to spread across heavier edges. Both algorithms have, besides the graph itself, two parameters `targetBurntRatio` and `p`. As long as the `targetBurntRatio` is not reached, a new fire is started at a random vertex, which becomes the first active node. The `targetBurntRatio` is reached once the number of (not necessarily distinct) edges burned is at least `targetBurntRatio` times the total number of edges. The algorithm can be seen in Pseudocode 1.

While there is an active node u, we consider its neighbors N which have not been visited in the current fire. If N is non-empty, with probability `p` we pick a random neighbor v from N. Then burn the edge uv, make v an active node, and remove v from N. Otherwise, we go to the next active node, until there are none left.

The standard and weighted variants of Forest Fire differ in how this random neighbor v is drawn from N. In the standard variant every element of N has equal probability to be picked, while in the weighted variant the probabilities are proportional to the edge weights, i.e., the probability that $v \in N$ is drawn is $\frac{w(uv)}{W_{u,N}}$ with $W_{u,N} = \sum_{x \in N} w(ux)$.

```
(WEIGHTED)FORESTFIRE(G: Graph, targetBurntRatio, p):
1  while targetBurntRatio is not reached do          // start new fire
2      let $u$ := random vertex
3      activeNodes = {$u$}
4      while activeNodes $\neq \emptyset$ do
5          $u$ := activeNodes.takeElement()
6          N := not visited neighbors of $u$
7          while N $\neq \emptyset$ and WEIGHTEDCOIN(p) do
8              $v$ := choose randomly* from N          // * here WFF differs from FF
9              burn $uv$
10             mark $v$ as visited
11         end
12     end
13 end
14 return how often each edge was burned
```

Pseudocode 1: The (weighted) forest fire algorithm. WEIGHTEDCOIN (p) is a function which is TRUE with probability p and FALSE with probability $(1 - p)$. In line 8, weighted forest selects a $v \in N$ with probabilities proportional to $w(uv)$ while the standard forest fire selects elements of N with equal probabilities disregarding the edge weights.

Assuming `targetBurntRatio` and `p` are constant, the standard variant of Forest Fire runs in $\Theta(n + m)$. In the weighted variant, the selection of a random neighbor, with probabilities proportional to edge weight and without replacement is the main challenge for a linear time implementation. For this we can use the *exponential clock method* [19], [21]. Let Z be a (pseudo-)random variable with the same distribution as the random variable denoting the number of edges drawn from N. In the exponential clock method we assign every $v \in N$ a key

$$k_v = -\frac{\ln(U_v)}{w(uv)} \qquad \text{with} \quad U_v \sim U(0,1)$$

and draw the Z neighbors with the smallest keys k_v [19], [21]. We can use quickselect [16], [17] to find the Z-smallest key $\hat{k}$ in linear time and then draw all neighbors $v \in N$ with $k_v \leq \hat{k}$.

Now we need a way to simulate Z. Notice that Z is the minimum of $|N|$ and a random variable Y with a shifted geometric distribution.

$$X_i \sim \mathrm{Ber}(\mathtt{p})$$
$$Y + 1 = \min\{i \in \mathbb{N}_+ : X_i = 0\} \sim \mathrm{Geo}(1 - \mathtt{p})$$

To simulate a random variable $G \sim \mathrm{Geo}(p)$, one can use a pseudo random uniform variable $U \in (0,1)$ and the formula

$$G := \lceil \ln U / \ln(1 - p) \rceil [18].$$

Thus, a random variable Z with the same distribution as the number of neighbors drawn from N can be calculated as

$$Z = \min\{|N|, Y\} = \min\{|N|, \lceil \ln U / \ln \mathtt{p} \rceil - 1\}.$$

Using these techniques we can sample the neighbors in linear time, and thus weighted forest fire can be implemented in $\Theta(n + m)$.

In our case we have set `targetBurntRatio` $= 5$ and `p` $= 95\%$ for forest fire and also for its weighted variant. These have shown promising results,

6.3.3 Effective Resistance

Spielman & Srivastava [6] present a sparsification algorithm based on an analogy to effective conductance in electrical networks. It does not have linear time but satisfies the guarantees of spectral sparsification from Definition 5. The effective resistances are defined by identifying the graph with an electrical network, with n nodes in which an edge e is a resistor with resistance $1/w(e)$. The effective resistance R_e across an edge $e \in E$ is the potential difference between its endpoints when an unit current is applied to one endpoint of e and extracted at the other [6].

The algorithm has as inputs a graph G and a parameter q. It samples q edges e with replacement and with probabilities p_e proportional to $w(e) \cdot R_e$. It then adds every sampled edge e to the sparsifier with weight $\frac{w(e)}{qp_e}$, summing up the weights if it is selected multiple times [6]. We can also use $s(e) := R_e$ as a score function.

Spielman & Srivastava [6] also present a data structure to calculate the effective resistances efficiently. Let $W \in \mathbb{R}^{m \times m}$ be a diagonal matrix with $W_{ii} = w(e_i)$ and let $B \in \mathbb{R}^{m \times n}$ be defined by orienting the edges arbitrarily and setting:

$$B_{ik} = \begin{cases} 1 & \text{, if } v_k \text{ is head of } e_i \\ -1 & \text{, if } v_k \text{ is tail of } e_i \\ 0 & \text{, otherwise} \end{cases}.[6]$$

The Laplacian L can be written as $L = B^T W B$ and is a symmetric positive semi-definite matrix. If L^+ is the *Moore-Penrose Pseudoinverse* of L, the effective resistance R_e of an edge $e = v_i v_j$ can be calculated by subtracting the i-th and j-th columns of $W^{\frac{1}{2}} B L^+$:

$$R_e = \| W^{\frac{1}{2}} B L^+ (b_i - b_j) \|_2^2,$$

where $\{b_1, ..., b_n\}$ is the standard basis.

To create a data structure for querying the effective resistances in $\mathcal{O}(\log n)$ time Spielman & Srivastava [6] use a lemma from Achlioptas [22] and a solver form D. A. Spielman & Teng [23]. This data structure is a matrix $Z = Q W^{\frac{1}{2}} B L^+ \in \mathbb{R}^{k \times n}$, where $Q \in \mathbb{R}^{k \times n}$ is a random $\pm \frac{1}{\sqrt{k}}$-matrix with $k = \frac{24 \log n}{\varepsilon^2}$ [6]. After Z has been calculated, the effective resistance of an edge $e = uv$ can be approximated in $\mathcal{O}(\log n)$ time by subtracting two columns of Z: $R_{v_i v_j} \approx \| Z(b_i - b_j) \|^2$ [6].

6.4 Unbiased Threshold

Unbiased Threshold is a variant of Threshold sparsification (Section 6.2.1) which uses edge weights as scores ($s = w$). It preserves partition weights in expectation and has a threshold T as its central parameter.

For every edge e this variant samples e with its original weight if $T \leq w(e)$. Otherwise, with probability $p_e = \frac{w(e)}{T}$, the edge e is sampled and assigned the weight T. Then the weight of every edge e is preserved in the sparsifier $G' = (V, E', w')$ in expectation:

$$\mathbb{E}[w'(e)] = \begin{cases} T p_e = w(e) & \text{, if } w(e) < T \\ w(e) & \text{, otherwise.} \end{cases}$$

Additionally, the Laplacian matrix (Definition 2) of G' and the weight of every cut (Definition 1) and partition (Definition 3) in G' are in expectation identical to their counterparts in G since they are sums of edge weights.

With this variant, the number of edges cannot be controlled precisely. The number of edges is in expectation

$$M(T) := \mathbb{E}[m'] = |\{e \in E : T \leq w(e)\}| + \frac{w(\{e \in E : T > w(e)\})}{T}.$$

We want to solve

$$M(T) = \texttt{target} \tag{VI}$$

for T.

First we notice that M is a continuous, monotonically decreasing function, since it is the sum of the expected value of indicator random variables $X_e \in \{0, 1\}$ with $X_e = 1$ if and only if e is sampled

$$M(T) = \mathbb{E}[m'] = \sum_{e \in E} \mathbb{E}[X_e]$$

and

$$\mathbb{E}[X_e] = \begin{cases} 1 & \text{, if } \ T < w(e) \\ \frac{w(e)}{T} & \text{, otherwise} \end{cases}$$

is a continuous, monotonically decreasing function in T. Therefore, we can solve Equation VI for T by employing a technique similar to the calculation of the normalization factor in Section 6.2.3.

First we search for an interval of consecutive weights $w_L, w_R \in \text{Image}(w)$ by finding

$$w_R = \min\{x \in \text{Image}(w) : M(w_R) \leq \texttt{target}\}.$$

Then the T solving Equation VI must be in the interval $(w_L, w_R]$. And since w_L and w_R are consecutive weights this reduces Equation VI to

$$|\{e \in E : w_R \leq w(e)\}| + \frac{w(\{e \in E : w_R > w(e)\})}{T} = \texttt{target} \ .$$

Thus, we can calculate the threshold T as[3]

$$T = \frac{w(\{e \in E : w(e) < w_R\})}{\texttt{target} - |\{e \in E : w_R \leq w(e)\}|}.$$

6.5 K-Neighbor

The *k-Neighbor* (*kN*) is a local sparsification algorithm by Sadhanala et al. [24]. At every vertex $u \in V$ with $\deg(u) \geq k$ it samples k incident edges with replacement and probabilities proportional to the edge weights. The selected edges are added to the sparsifier with weight $\frac{W_u}{2k}$ with

$$W_u := \sum_{v \in N(u)} w(uv).$$

[3]To hit the desired number of edges in expectation a non-integer threshold can be necessary. If we need integer edge weights we can employ randomized rounding where $x = n + f$ with $n \in \mathbb{N}, f \in [0, 1]$ is rounded to $\lceil x \rceil = n + 1$ with probability f and to $\lfloor x \rfloor = n$ with probability $1 - f$. Then $\mathbb{E}(\text{round}(x)) = (n + 1)f + n(1 - f) = n + f = x$.

If an edge is selected multiple times its final weight is the sum. At vertices u with $\deg(u) < k$, all edges are selected with half their original weights. We use only half of the weight, since every edge can be selected from two endpoints. [24]

The k-Neighbor sampler can be implemented in $\mathcal{O}(m)$ time and according to its authors "is practically robust across various settings and performs just as well as spectral sparsification" [24].

We want to select k such that the `target` number of edges in the sparsifer is reached in expectation. If an edge $e = uv \in E$ is incident to at least one vertex of degree $\leq k$ it is always sampled. Otherwise, if both its endpoints are of degree $> k$, the probability that e is selected at its endpoint u at one of the k draws is $\frac{w(e)}{W_u}$. Thus, the probability of e being selected at none of the k draws at u and none of the k draws at v is

$$\left(1 - \frac{w(e)}{W_u}\right)^k \left(1 - \frac{w(e)}{W_v}\right)^k.$$

Let S_{uv} be an indicator random variable, which is 1 if and only if an edge $uv \in E$ is sampled, then:

$$\mathbb{E}(S_{uv}) = \begin{cases} 1 & \text{, if } \deg(u) \leq k \lor \deg(v) \leq k \\ 1 - \left(1 - \frac{w(uv)}{W_u}\right)^k \left(1 - \frac{w(uv)}{W_v}\right)^k & \text{, otherwise} \end{cases}.$$

Therefore,

$$\begin{aligned} \mathbb{E}[m'] &= \sum_{e \in E} \mathbb{E}[S_e] \\ &= |\{uv \in E : \deg(u) \leq k \lor \deg(v) \leq k\}| \\ &\quad + \frac{1}{2} \sum_{\substack{u \in V \\ \deg(u) > k}} \sum_{\substack{v \in N(u) \\ \deg(v) > k}} 1 - \left(1 - \frac{w(uv)}{W_u}\right)^k \left(1 - \frac{w(uv)}{W_v}\right)^k. \end{aligned}$$

Since $\mathbb{E}[m']$ is a monotonous function in k, we can use this formula to binary search for the k such that $\mathbb{E}[m']$ approximately equals `target`.

7 Experimental Evaluation

Looking at all the sparsification algorithms from Section 6, we want to test which are suitable for multilevel graph partitioning with KaMinPar and how they impact the run time of KaMinPar.

7.1 Benchmark

To compare the different sparsification algorithms we need a benchmark set of graphs where sparsification could be potentially useful for KaMinPar. We used the increase of density (number of edges divided by the number of vertices) across the levels as an indicator, only selecting graphs where this density increased at least by a factor of four.

Our first source of graphs was the benchmark set of Maas et al. [25]. Additionally, we downloaded 464 large matrices from the *SuiteSparse Matrix Collection* [26]. These were converted to graphs by interpreting them as adjacency matrices. Only 25 of these over 500 large graphs had a sufficient density increase and were included in the benchmark. This indicates that on a lot of graphs sparsification is unnecessary. To increase the benchmark size we generated 12 random graphs generated with the *Communication-free Graph Generators KaGen* [27]. These graphs were generated using the Erdős-Rényi model and the R-Mat model [27], [28].

In Figure 4 we see that the graphs in the benchmark are fairly large with the average number of vertices being about 34 million and the average number of edges being about 233 million. We also see that the density can increase a lot over the course of the levels—in one instance even by more than 400. In Table 1 we list the graphs with an increase above 100. There are natural graphs such as the protein k-mer graphs, or the Chinese microblogging site Sina Weibo as well as a synthetic graph (ba_2_22_s0_md_2) generated with the Barabási-Albert model [29].

To speed up the experiments we also selected 12 random graphs from our benchmark of 37 graphs for a *small benchmark set.*

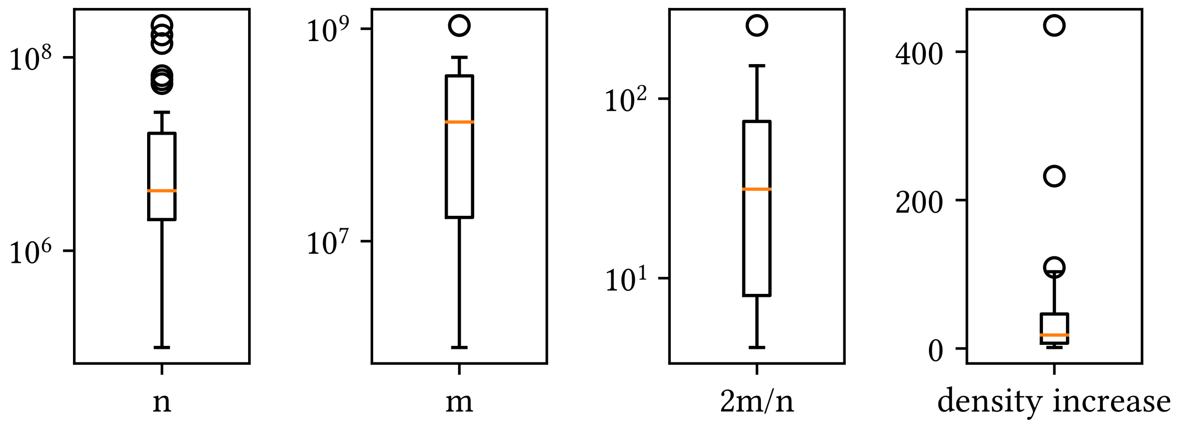

Figure 4: Basic statistics about the benchmark. From left to right the number of nodes, the number of edges, the average degree, and the maximal density increase over the course of the levels without sparsification and $k = 32$ as boxplots.

Graph	Density Increase
kmer_V1r	435.5
ba_2_22_s0_md_2	232.2
kmer_P1a	109.1
soc-sinaweibo	103.4
kmer_A2a	101.1

Table 1: Graphs with the very hight (> 100) density increase over the course of the levels without sparsification with $k = 32$.

7.2 Partition Quality

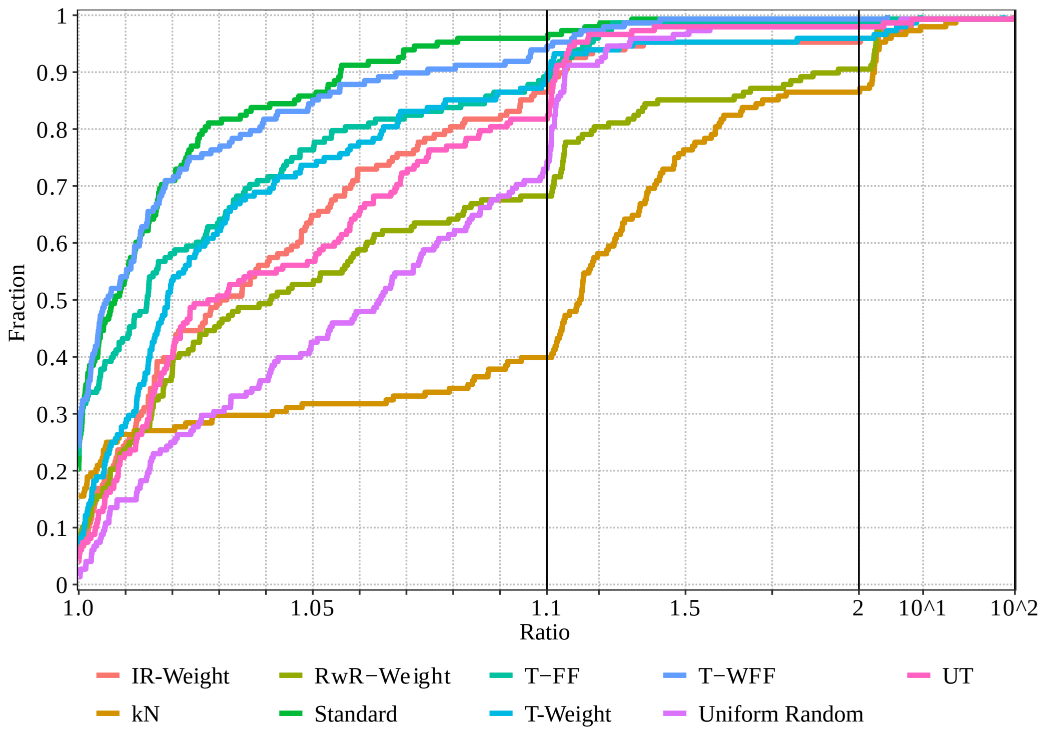

Figure 5: Performance profile of the algorithms with sparsification set such that the density does not increase by a factor of more than 2 between levels. It shows in which *fraction* of the instances the quality of the algorithm was no more than *ratio* times the best solution found by any algorithm, i.e., the point $(1.05, 0.7)$ means that in 70% of instances the algorithm deviated from the best solution of any of the algorithms by at most 5%.

Important is of course the quality, i.e. the cut weight of the computed partition. Sparsification reduces the amount of information available for partitioning. Thus, we expect the standard variant of KaMinPar without sparsification to perform the best. As an additional baseline we look at the quality impacts of uniform random sampling, i.e., just sampling every edge with equal probability (Uniform Random) and thus ignoring weight and position of edges. Since graph partitioning is an NP-hard problem it is not feasible to calculate the optimal solution as a reference point. Thus, we just compare the results of the algorithms to each other.

In Figure 5 we see that many of the sparsification algorithms perform significantly better than the baseline of uniform random sampling. The best sparsification algorithm in the tests was threshold with the weighted forest fire score (T-WFF) with a mean cut weight of $6.805 \cdot 10^7$, which is almost as good as no sparsification with $6.801 \cdot 10^7$. Notably, also the threshold with weight (T-Weight) is also quite good ($6.868 \cdot 10^7$), despite being very simple. The sparsification algorithms k-Neighbors and Random with Replacement with weights perform even worse than Uniform Random.

7.3 Run Time

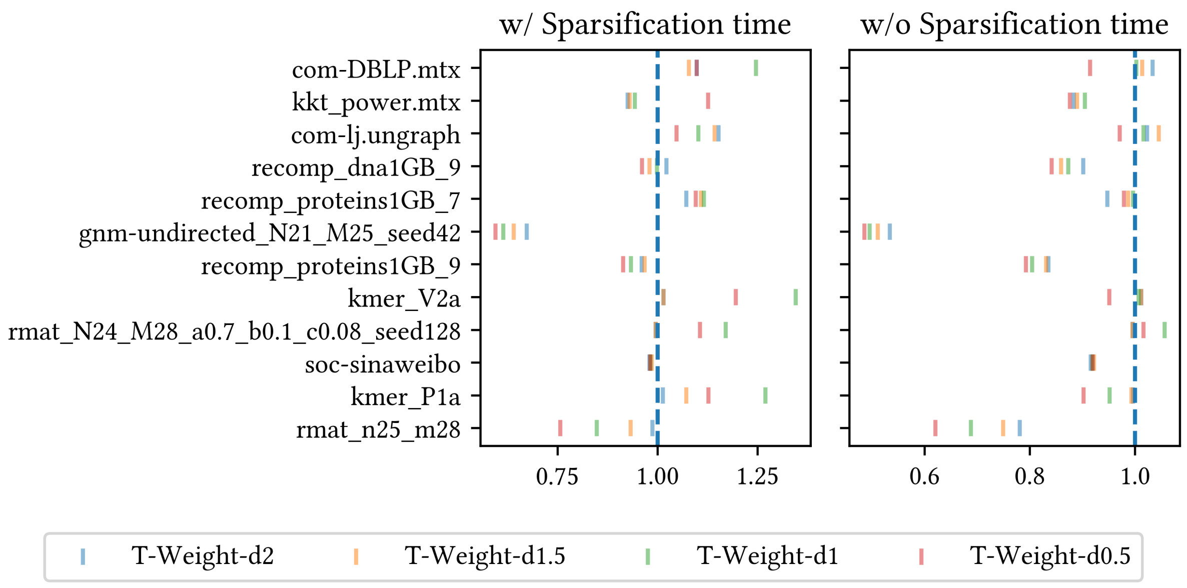

Figure 6: Time across the graphs of the small benchmark relative to KaMinPar without sparsification of the weight threshold algorithms with sparsification set such that the density does not increase by more than $\{2, 1.5, 1, 0.5\}$. The left plot includes sparsification time and the right one does not.

While sparsification leads to provable linear time, most of our implementations of sparsification algorithms made KaMinPar slower. Looking for speedups we thus focus on weight threshold (T-Weight) —the simplest and fastest of the high-quality algorithms from Section 7.2. In Figure 6 we see that this leads to speedups on some instances. For example in the recomp_proteins1GB_9 graph or in the gnm-undirected_N21_M25_seed42 Erdős-Rényi graph. Thus, sparsification does not only improve the asymtotic time but also leads to practical accalations.

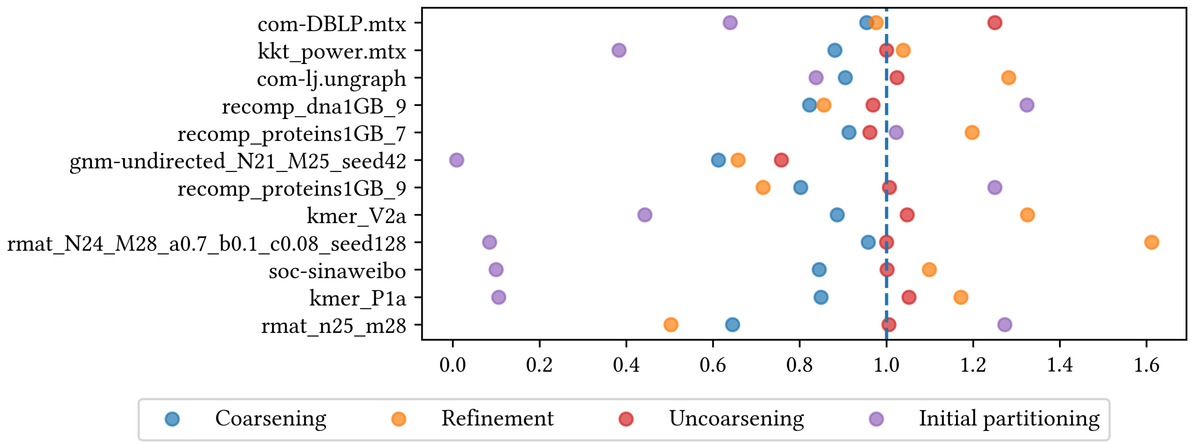

Figure 7: Times of the central MGP steps in weight threshold such that the maximal density increase is no more than 0.5 (T-Weight-d0.5) relative to their time without sparsification in instances of the small benchmark set.

However, even when ignoring sparsification time the KaMinPar sometimes gets slower with sparsification. To examine this closer, we look at the speedups and slowdowns by category of the most aggressive of these algorithms in Figure 7. There the third phase of MGP is boken down into uncoarsening, which is the projection of the partitions onto the larger levels, and refinement, which is the local improvement by moving vertices between blocks. We see that refinement, uncoarsening, and initial partitioning all take longer on some graphs with sparsification than without.

In Figure 8 we can see that the number of levels with sparsification is often higher than without and the levels—especially the last one—can have more vertices. This is to be expected, since sparsification leaves a graph with fewer edges and thus fewer opportunities to merge clusters during coarsening. The increased number of levels and increased number of vertices on levels counteracts the speedup of sparsification and could explain why parts of the MGP gets slower sometimes. For example, refinement gets more new information, which had been reduced by sparsication and uncoarsening as well as refinement are done more often as the number of levels increases.

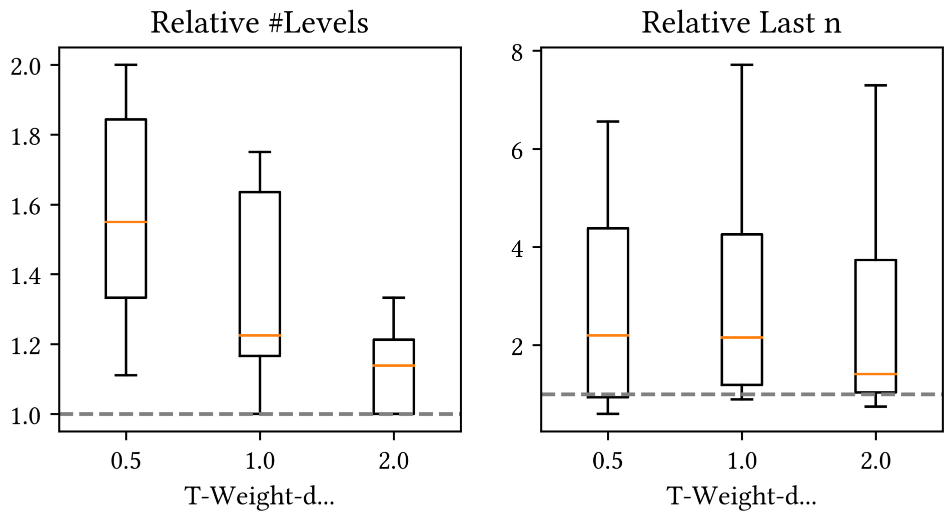

Figure 8: boxplot of the number of levels and the number of vertices in the last level of weight threshold sparisfying such that density increases by no more than 0.5 relative to no sparsification

There are couple of areas where further engineering could lead to time improvements. A part shared by all sparsification algorithms is the building of the sparsifier graph after the edges to include have been selected. KaMinPar uses the CSR format to represent graphs internally, which does not allow the deletion of edges without changing the whole structure. Because of this we need to rebuild the whole graph in the CSR format for the sparsifier. This could be avoided by either integrating the sparsification into the construction of the coarse graph such that a CSR format only has to be constructed once not twice. Or through some lazy evaluation, which leaves the underling CSR graph the same but marks edges as to be ignored.

The threshold sampling has three parts: calculating the scores, finding the threshold with quickselect, and the sampling where all edges with scores above the threshold are selected as well as the right number of edges at the threshold. Some speedups might be achieved in all these areas by increasing the parallelization, optimizing quickselect or tuning the parameters of weighted forest fire.

Most important is tuning when and how aggressively we sparsify. We have seen in Section 7.3 that even on our benchmark set of instances with high density increase the speedup or slowdown with sparsification heavily depends on the particular graph. Thus, on some instances no sparsification would be appropriate while others might benefit from heavy sparsification.

8 Conclusion

In this thesis we proved that through sparsification linear time multilevel graph partitioning is possible. We presented several sparsification algorithms and analyzed how their parameters have to be set to approximately sparsify to a target number of edges. Finally, we evaluated them experimentally.

We have shown that under certain conditions in multilevel graph partitioning the total number of vertices is linear in the number of vertices of the input. These conditions are that size-constraint label propagation and two-hop clustering is used for the coarsening, isolated vertices are clustered with each other, and that the maximum cluster size is larger than four times the average vertex weight. If we additionally use a linear time sparsification algorithm to control the number of edges across the levels, then MGP runs in linear time. By implementing a linear time sparsification algorithm into KaMinPar we achieve linear time MGP.

We discussed a variety of sparsification algorithms. Some of them are based on weights such as Unbiased Threshold and K-Neighbor, while others use different kind of scores to evaluate the importance of edges such as effective resistance or the forest fire score. We modified the forest fire scores which were originally desingned for unweighted graphs by Lindner et al. [14] into weighted forest fire score which take edge weights into account. These scores can be used by different score-based sparsification algorithms. To make all these different sparsification approaches comparable we analyzed how many edges these algorithms sample (in expectation) as a function of their parameters and developed routines to set these such that a given target is reached.

The experimental evaluation shows that sparsification can be employed with only little reduction in quality. Especially, our weighted forest fire performed almost as well as no sparsification. But also the simple weight thrshold stood out by preserving the quality well while being very simple. While we have improved the asymtotic time to linear, the practical time remains a more mixed picture. On some graphs sparsification leads to significant speedups while on others it can lead to KaMinPar taking more time.

8.1 Future Work

Sparsification has very promising impacts on the theoretical asymtotic run time as well as the practical quality. We have also seen practical speedups in our experimental evaluation. But we have also seen that sparsification has adverse effects on the run time of some instances. To practically and reliably use sparsification to accelerate MGP more optimization and research is necessary. Especially, the question when and how much to sparsify remains open. Additionally, the sparsification itself could be sped up by integrating it more tightly into KaMinPar and by optimizing and tuning the algorithms.

There are also other interesting research directions. Given a graph, can we predict the effects of sparsification during MGP on it? Can we counteract adverse effects such as an increased number of levels? Maybe it is even possible to use the theoretical guarantees of ε-cut sparsifers to infer statements about the impact of sparsification on the quality of MGP without sacrificing the fast run time.

Bibliography

[1] K. Andreev and H. Räcke, “Balanced graph partitioning,” in *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, in SPAA '04. Barcelona, Spain: Association for Computing Machinery, 2004, pp. 120–124. doi: 10.1145/1007912.1007931.

[2] I. Koutis, G. L. Miller, and R. Peng, “Approaching Optimality for Solving SDD Linear Systems,” *SIAM Journal on Computing*, vol. 43, no. 1, pp. 337–354, 2014, doi: 10.1137/110845914.

[3] R. Peng and D. A. Spielman, “An efficient parallel solver for SDD linear systems,” in *Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing*, in STOC '14. New York, New York: Association for Computing Machinery, 2014, pp. 333–342. doi: 10.1145/2591796.2591832.

[4] A. A. Benczúr and D. R. Karger, “Approximating s-t minimum cuts in Õ(n2) time,” in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, in STOC '96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 47–55. doi: 10.1145/237814.237827.

[5] C. Chekuri and C. Xu, “Minimum Cuts and Sparsification in Hypergraphs,” *SIAM Journal on Computing*, vol. 47, no. 6, pp. 2118–2156, 2018, doi: 10.1137/18M1163865.

[6] D. A. Spielman and N. Srivastava, “Graph sparsification by effective resistances,” in *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, in STOC '08. Victoria, British Columbia, Canada: Association for Computing Machinery, 2008, pp. 563–568. doi: 10.1145/1374376.1374456.

[7] W. S. Fung, R. Hariharan, N. J. Harvey, and D. Panigrahi, “A general framework for graph sparsification,” in *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing*, in STOC '11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 71–80. doi: 10.1145/1993636.1993647.

[8] L. Gottesbüren, T. Heuer, P. Sanders, C. Schulz, and D. Seemaier, “Deep Multilevel Graph Partitioning,” in *29th Annual European Symposium on Algorithms (ESA 2021)*, P. Mutzel, R. Pagh, and G. Herman, Eds., in Leibniz International Proceedings in Informatics (LIPIcs), vol. 204. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 1–17. doi: 10.4230/LIPIcs.ESA.2021.48.

[9] Y. Chen *et al.*, “Demystifying Graph Sparsification Algorithms in Graph Properties Preservation,” *Proc. VLDB Endow.*, vol. 17, no. 3, pp. 427–440, Nov. 2023, doi: 10.14778/3632093.3632106.

[10] H. Meyerhenke, P. Sanders, and C. Schulz, “Partitioning Complex Networks via Size-Constrained Clustering,” in *Experimental Algorithms*, J. Gudmundsson and J. Katajainen, Eds., Cham: Springer International Publishing, 2014, pp. 351–363.

[11] U. N. Raghavan, R. Albert, and S. Kumara, “Near linear time algorithm to detect community structures in large-scale networks,” *Phys. Rev. E*, vol. 76, no. 3, p. 36106, Sep. 2007, doi: 10.1103/PhysRevE.76.036106.

[12] D. LaSalle, M. M. A. Patwary, N. Satish, N. Sundaram, P. Dubey, and G. Karypis, “Improving graph partitioning for modern graphs and architectures,” in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, in IA3 '15. Austin, Texas: Association for Computing Machinery, 2015. doi: 10.1145/2833179.2833188.

[13] J. Batson, D. A. Spielman, N. Srivastava, and S.-H. Teng, “Spectral sparsification of graphs: theory and algorithms,” *Commun. ACM*, vol. 56, no. 8, pp. 87–94, Aug. 2013, doi: 10.1145/2492007.2492029.

[14] G. Lindner, C. L. Staudt, M. Hamann, H. Meyerhenke, and D. Wagner, “Structure-preserving sparsification of social networks,” in *2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, 2015, pp. 448–454. doi: 10.1145/2808797.2809313.

[15] D. A. Spielman and Teng, “Spectral Sparsification of Graphs,” *SIAM Journal on Computing*, vol. 40, no. 4, pp. 981–1025, 2011, doi: 10.1137/08074489X.

[16] C. A. R. Hoare, “Algorithm 65: find,” *Commun. ACM*, vol. 4, no. 7, pp. 321–322, Jul. 1961, doi: 10.1145/366622.366647.

[17] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, “Time bounds for selection,” *Journal of Computer and System Sciences*, vol. 7, no. 4, pp. 448–461, 1973, doi: https://doi.org/10.1016/S0022-0000(73)80033-9.

[18] D. E. Knuth, *The Art of Computer Programming, Volume 2 / Seminumerical Algorithms*, 3rd ed. Addison-Wesley, 1973.

[19] L. Hübschle-Schneider and P. Sanders, “Parallel Weighted Random Sampling,” *ACM Trans. Math. Softw.*, vol. 48, no. 3, Sep. 2022, doi: 10.1145/3549934.

[20] J. Leskovec, J. Kleinberg, and C. Faloutsos, “Graph evolution: Densification and shrinking diameters,” *ACM Trans. Knowl. Discov. Data*, vol. 1, no. 1, p. 2–es, Mar. 2007, doi: 10.1145/1217299.1217301.

[21] L. Hübschle-Schneider, “Communication-Efficient Probabilistic Algorithms: Selection, Sampling, and Checking,” Karlsruher Institut für Technologie (KIT), 2020. doi: 10.5445/IR/1000127719.

[22] D. Achlioptas, “Database-friendly random projections: Johnson-Lindenstrauss with binary coins,” *Journal of Computer and System Sciences*, vol. 66, no. 4, pp. 671–687, 2003, doi: https://doi.org/10.1016/S0022-0000(03)00025-4.

[23] D. A. Spielman and Teng, “Nearly Linear Time Algorithms for Preconditioning and Solving Symmetric, Diagonally Dominant Linear Systems,” *SIAM Journal on Matrix Analysis and Applications*, vol. 35, no. 3, pp. 835–885, 2014, doi: 10.1137/090771430.

[24] V. Sadhanala, Y.-X. Wang, and R. Tibshirani, “Graph Sparsification Approaches for Laplacian Smoothing,” in *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, A. Gretton and C. C. Robert, Eds., in Proceedings of Machine Learning Research, vol. 51. Cadiz, Spain: PMLR, 2016, pp. 1250–1259. [Online]. Available: https://proceedings.mlr.press/v51/sadhanala16.html

[25] N. Maas, L. Gottesbüren, and D. Seemaier, “Parallel Unconstrained Local Search for Partitioning Irregular Graphs,” in *2024 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pp. 32–45. doi: 10.1137/1.9781611977929.3.

[26] T. A. Davis and Y. Hu, “The university of Florida sparse matrix collection,” *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011, doi: 10.1145/2049662.2049663.

[27] D. Funke, S. Lamm, P. Sanders, C. Schulz, D. Strash, and M. von Looz, “Communication-free Massively Distributed Graph Generation,” in *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21 – May 25, 2018*, 2018.

[28] L. Hübschle-Schneider and P. Sanders, “Linear Work Generation of R-MAT Graphs,” *Network Science*, vol. 8, no. 4, pp. 543–550, 2020.

[29] A.-L. Barabási and R. Albert, “Emergence of Scaling in Random Networks,” *Science*, vol. 286, no. 5439, pp. 509–512, 1999, doi: 10.1126/science.286.5439.509.

Acknowledgments

This thesis is written in the beautiful typesetting language typst[4] using the packages certz[5], ctheorems[6], and algo[7]. Graphics were created with cetz, matplotlib[8], and mkexp[9].

Small parts of scripts, like cosmetic improvements of the plots, where created with the help of the generative A.I. GitHub Copilot[10].

Some proof reading has been done by Linda Staerke, Karina Schalldach, Achim Rosch, and my advisors.

Appendix

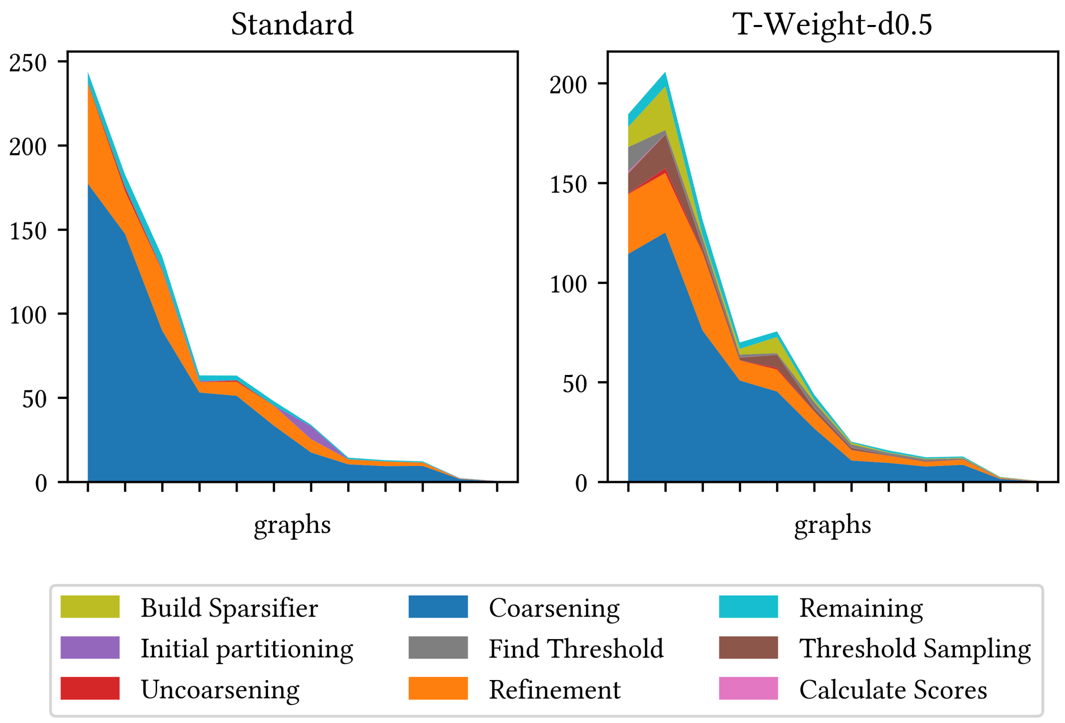

Figure 9: Run time breakdown with one thread by different categories of the KaMinPar without sparsification (left) and KaMinPar with threshold weight sparsication such that the density does not increase by more than 0.5 (right). On the x-axis are the graphs from the small benchmark set.

[4]typst.app
[5]https://typst.app/universe/package/cetz/
[6]https://typst.app/universe/package/ctheorems/
[7]https://typst.app/universe/package/algo/
[8]https://matplotlib.org/
[9]https://github.com/DanielSeemaier/mkexp
[10]https://github.com/features/copilot

T-Weight-d	**0.5**	**1**	**1.5**	**2**
com-DBLP.mtx	2.14	N/A	N/A	N/A
com-lj.ungraph	2.64	N/A	N/A	N/A
gnm-undirected_N21_M25_seed42	0.21	0.23	0.26	0.3
kkt_power.mtx	2.02	0.41	0.36	0.36
kmer_P1a	2.3	6.56	10.83	4.36
kmer_V2a	5.0	N/A	N/A	N/A
recomp_dna1GB_9	0.76	0.99	0.85	1.23
recomp_proteins1GB_7	5.47	30.11	9.1	2.35
recomp_proteins1GB_9	0.58	0.66	0.81	0.76
rmat_N24_M28_a0.7_b0.1_c0.08_seed128	N/A	N/A	N/A	N/A
rmat_n25_m28	0.36	0.51	0.73	0.94
soc-sinaweibo	0.77	0.78	0.81	0.76
Min	0.21	0.23	0.26	0.3
Median	2.02	0.72	0.81	0.85
Max	5.47	30.11	10.83	4.36

Table 2: Needed speedups to reach the same time as without sparsification. N/A means that even when ignoring sparsification there were no speedups, i.e., sparsification made the rest slower or no sparsification was activated.