Engineering Sampling-Based Streaming Graph Partitioning Algorithms

Loris Wilwert

August 17, 2023

4165279

Bachelor Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:
Univ.-Prof. PD. Dr. rer. nat. Christian Schulz

Co-Supervisor:
Prof. Dr. Bora Uçar

Co-Supervisor:
M. Sc. Marcelo Fonseca Faraj

Acknowledgments

I would like to thank Prof. Dr. Christian Schulz for giving me the opportunity to work on this interesting project under his supervision. With his open nature, he created one of the most pleasant working environments I experienced during my three years of Bachelor's studies. The same goes for Prof. Dr. Bora Uçar, whom I had the pleasure of meeting personally during my stay in Lyon, where I defended my thesis. I would also like to express my gratitude to Marcelo Fonseca Faraj, who I could turn to at any time and who always made an effort to answer any of my questions as quickly as possible. Furthermore, I would like to thank Henrik Reinstädtler for his help during the writing of my thesis.
My greatest thanks go to my parents, who have dedicated parts of their lives to helping me become the person that I am today. This work would not have been possible without their constant support and love. Likewise, I cannot thank my sister enough, who accompanied me through my entire life and who, together with her partner, facilitate my studies in Heidelberg. Finally, I am grateful for the help of all my various friends, who have been on my side in the different stages of my life up to this point.

Hiermit versichere ich, dass ich die Arbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und wörtlich oder inhaltlich aus fremden Werken Übernommenes als fremd kenntlich gemacht habe. Ferner versichere ich, dass die übermittelte elektronische Version in Inhalt und Wortlaut mit der gedruckten Version meiner Arbeit vollständig übereinstimmt. Ich bin einverstanden, dass diese elektronische Fassung universitätsintern anhand einer Plagiatssoftware auf Plagiate überprüft wird.

Heidelberg, August 17, 2023

Wilwert

Loris Wilwert

Abstract

A common feature of many scalable algorithms that work on graphs for various applications is that they require the graphs to be partitioned. This means that the nodes of a graph should be divided into k blocks of roughly equal size while minimizing the sum of the weights of the edges running between these blocks. However, the applicable algorithm to solve this partitioning problem highly depends on the available memory of the machine. Due to the growing size of many real-world graphs, in-memory partitioners often struggle to partition large networks on machines with low computational resources. For this reason, there has been an interest recently in using streaming algorithms for graph partitioning, which have a low memory usage but typically yield only low-quality solutions. In this work, we tackle this quality issue by proposing a new algorithm that uses an extended streaming model, with which we are allowed to perform all node assignments after having streamed the entire graph. To achieve this, we sample only a subset of the edges of every streamed node, enabling us to represent a sampled version of the input graph using $O(n)$ memory. We further simplify the graph by contracting twins, i.e. nodes that share the same neighbourhood. The resulting contracted graph is then partitioned by an underlying in-memory multilevel algorithm. During the last step, we undo the twin contractions and make the final node assignments. Our proposed streaming algorithm overall outperforms current state-of-the-art competitors on graphs from scientific applications in terms of edge-cut quality and running time.

Contents

CHAPTER 1

Introduction

1.1 Motivation

In our everyday life, we use many tools and technologies that rely on large complex systems such as social networks, biological systems or road networks. To make the analysis on these systems feasible, the usual way is to model them as graphs. A common feature of many scalable algorithms, that work on these graphs for various applications, is that they require the graphs to be partitioned. One generally differentiates between edge-based and vertex-based partitioning [36], but we only focus on the latter formulation, which is also simply referred to as *balanced graph partitioning* [1] or *k-way partitioning* [8]. More specifically, for a given (strictly) positive integer k, the goal is to divide the nodes of the graph into k disjoint blocks of roughly equal size. In addition, the partition must most often minimize the *edge-cut*, which is the sum of the weights of all edges running between different blocks. There exists a large number of applications where the partition of the graph is required as a preprocessing step [8]. Scientific simulations for example use graph partitioning to balance the load over the processing elements (PEs) of a supercomputer while minimizing the communication between these PEs [15]. Another application is the use of partitioned graphs in the precomputation phase of route planning algorithms [10].
Since the problem for $k = 2$, which is also called the *minimum bisection problem*, is already NP-complete [17] and since there does not exist a polynomial-time approximation algorithm with a constant factor for any k [1], the usual approach is to construct algorithms that are based on heuristics. The most commonly used heuristic is the *multilevel scheme*, which serves as a basis for the currently most well-known in-memory tools like Metis [21], Scotch [27] or KaHIP [31, 32, 35]. Those algorithms manage to compute high quality partitions, but they usually require the input graph to fit entirely into the main memory of a single machine. Thus, they often struggle to partition large networks on machines with low computational resources. Besides, in-memory partitioners are not applicable in *online* scenarios where the graph is not directly available but arrives piece-by-piece.

For the reasons above, there has been an interest recently in using *streaming* algorithms for graph partitioning [8], which are the focus of this thesis. The most common streaming model is the *one-pass model*, where the nodes of the input graph are streamed together with their edges one after the other and directly and permanently assigned to a block. The advantage of streaming algorithms is that they use little memory, because the set of edges of the graph does not have to fit entirely into the main memory, but they typically yield only low-quality solutions since they do not have a global view on the graph.
Our interest lies in combining the best of both worlds. We strive to construct a new streaming algorithm that manages to yield a partition quality that comes close to existing in-memory multilevel algorithms. At the same time, our algorithm should still be as fast and memory-efficient as current state-of-the-art streaming algorithms.

1.2 Our Contribution

In this thesis, we propose a new streaming algorithm for the balanced graph partitioning problem, together with an extension of the well-known one-pass model. More specifically, we introduce the notion of budget edge sampling to the streaming case, meaning that we sample for every streamed node v a budget b_v of its edges. This allows us to represent a sampled version of the input graph using $O(n)$ memory, which gives us the possibility to make all node assignments only after having streamed the whole graph. With the help of tuning experiments, we observe that the best sampling approach is to use the same low constant budget b for every node. Tuning this parameter b allows us to perform a feasible time-quality trade-off, which enables us to adjust our algorithm to specific use cases.
We also introduce twin contractions to the balanced graph partitioning problem. In detail, we argue that contracting *true* twins, i.e. nodes that have the same closed neighbourhood, in a preprocessing step to further simplify the graph is an effective heuristic in most cases. This is confirmed by our tuning experiments, where we also observe that contracting *false* twins, i.e. nodes that have the same open neighbourhood, turns out to be mostly ineffective. The resulting contracted graph is partitioned by an underlying in-memory multilevel algorithm, which is in our case KaFFPa. After that, we uncontract the twins and make the permanent assignments of the nodes to the blocks.
In comparison to current state-of-the-art competitors, our algorithm yields promising results on graphs from scientific applications, where it manages to achieve our goal of further closing the current gap between high-quality in-memory multilevel algorithms and fast streaming approaches. Regarding social networks, our algorithm still has room for improvement in terms of solution quality, because it even falls short of matching the edge-cut quality of the one-pass algorithm Fennel. Since our proposed algorithm introduces several new techniques to the streaming graph partitioning problem, we outline multiple approaches to further refine these techniques in future work.

1.3 Structure

The remainder of this thesis is organized as follows. In Chapter 2, we first state general definitions and notations that we use throughout the rest of this thesis. Moreover, we shortly go over the different phases of the multilevel scheme, which is the most common heuristic for the graph partitioning problem. Afterwards, we describe in Chapter 3 previous work that is related to this thesis. In detail, we start with the in-memory partitioners, where we outline the program KaFFPa, which builds the basis for our proposed streaming algorithm. We then move on to describe the field of research for the streaming graph partitioning problem, where we especially highlight Fennel and HeiStream. We close the chapter by briefly describing whether there exists previous work using budget edge sampling for graph partitioning. Next, we propose our new streaming algorithm in Chapter 4. This includes explaining our extended streaming model as well as outlining the two main parts of our algorithm, which are budget edge sampling and twin contraction. At the end of the chapter, we give a description of the implementation details. In Chapter 5, we determine the best values for the parameters of our algorithm in the tuning experiments before comparing the final configuration to the state-of-the-art competitors in the test experiments. Chapter 6 contains the conclusion from our experiments as well as an outlook to possible approaches and improvements in future work.

CHAPTER 2

Fundamentals

In this chapter, we first give general definitions and notations used throughout this thesis. Afterwards, we briefly explain the different phases of the multilevel scheme, which is the most commonly used heuristic for the graph partitioning problem.

2.1 General Definitions

Let $G = (V, E)$ be an *undirected* graph with $V = \{0, 1, ..., n - 1\}$ being the set of nodes and $E \subseteq V \times V$ being the set of edges. The graph G is required to be *simple*, meaning that it does not contain any multiple or self edges. We denote $n = |V|$ as the number of nodes and $m = |E|$ as the number of edges. Let $c : V \rightarrow \mathbb{R}_{>0}$ be a node-weight function and let $\omega : E \rightarrow \mathbb{R}_{>0}$ be an edge-weight function. The functions c and ω are generalized to sets, such that $c(V') = \sum_{v \in V'} c(v)$ for $V' \subseteq V$ and $w(E') = \sum_{e \in E'} \omega(e)$ for $E' \subseteq E$. The input graphs in this thesis are always *unweighted*, which is the same as setting unit node and edge weights, i.e. $\forall v \in V : c(v) = 1$ and $\forall e \in E : \omega(e) = 1$.
We define $N(v) = \{u : \{u, v\} \in E\}$ as the *open* neighbourhood and $N[v] = N(v) \cup \{v\}$ as the *closed* neighbourhood of a node $v \in V$. Let $d(v) = |N(v)|$ be the degree of v. Two nodes $u, v \in V$ are called *true* twins if they share the same closed neighbourhood, i.e. $N[u] = N[v]$, and they are called *false* twins if they share the same open neighbourhood, i.e. $N(u) = N(v)$. Figure 2.1 visualizes true and false twins.
A *path* in G is a (finite) sequence of pairwise distinct edges. Two nodes $u, v \in V$ are called *connected* if there exists a path between u and v. In a connected graph, every pair of nodes in the graph is connected. A graph $S = (V', E')$ is an *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. We call S an *induced* subgraph of G if $E' = E \cap (V' \times V')$. A *matching* $M \subseteq E$ is a (sub)set of edges that do not share any common node, meaning that $\forall e, e' \in M : e \cap e' = \emptyset$.
The *balanced graph partitioning* or *k-way partitioning* problem consists of dividing, for a given (strictly) positive integer k, the nodes of G into k disjoint blocks $V_1 \cup ... \cup V_k = V$

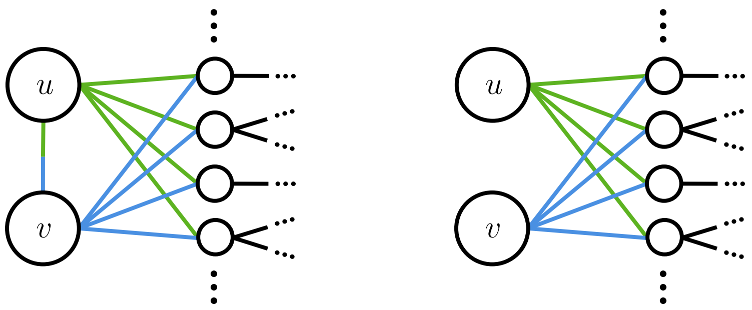


Figure 2.1: A visualization of two twins u and v. On the left side, both nodes are true twins because they share the same closed neighbourhood. On the right side, both nodes are false twins because they share the same open neighbourhood. The main difference is that true twins share a common edge in contrast to false twins.

with $\forall\, 1 \leq i < j \leq k : V_i \cap V_j = \emptyset$. Note that for $k = 2$ we speak of a *bipartition*. In addition, the partition must optimize a given objective function. Most often, the so-called *edge-cut* must be minimized, which is the sum of the weights of all edges running between different blocks:

$$\sum_{1 \leq i < j \leq k} \omega(E_{ij}) \quad \text{with} \quad E_{ij} = \{\{u,v\} \in E : u \in V_i \land v \in V_j\} \tag{2.1}$$

There exist also other objective functions that go beyond the scope of this thesis. The so-called *balance constraint* requires the weights of the blocks to be below a certain threshold L_{max} for a given *imbalance* ϵ:

$$\forall\, 1 \leq i \leq k : c(V_i) \leq L_{max} = (1+\epsilon) \cdot \left\lceil \frac{c(V)}{k} \right\rceil \tag{2.2}$$

There exist instances for which Equation 2.2 can never be fulfilled, i.e. graphs where only one node has an extremely large weight. To guarantee solvability, one must add $\max_{v \in V} c(v)$ to the equation. Increasing the second parameter ϵ generally allows to find smaller edge-cuts. For the extreme case $\epsilon = 0$, the graph is said to be *perfectly* balanced. Note that the perfect balance may sometimes not be achievable for the weighted case. A cluster is similar to a partition with the difference that there does not exist an overall balance constraint and that the number k of clusters is not given in advance.
The *quotient graph* Q of a partitioned graph G considers every block V_i as a node i of Q with a weight of $c(V_i)$. Two nodes i and j of the quotient graph are connected by an edge if there exist two nodes $u \in V_i$ and $v \in V_j$ so that $\{u, v\} \in E$. Two blocks are said to be *neighbouring blocks* if they are connected by at least one edge in the quotient graph. We call a node $v \in V_i$ a *boundary node* if it has a neighbour $u \in V_j$ with $i \neq j$.
The general idea of streaming models is to use little memory. The most common streaming model is the *one-pass model*, where the nodes of the input graph G are streamed together with their edges one after the other. In this model, a streamed node must be directly and permanently assigned to a block before streaming the next node.

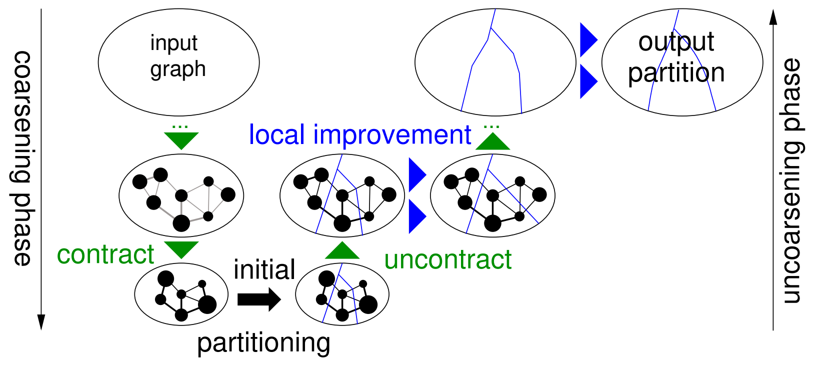


Figure 2.2: A visualization of the three phases that are used in the multilevel scheme. Figure taken from [13] (modified).

2.2 Multilevel Scheme

The most common heuristic for the graph partitioning problem is the *multilevel scheme*. The idea behind the multilevel scheme can be split up into three phases, which are depicted in Figure 2.2. First, the size of the input graph G is recursively reduced in the *coarsening* or *contraction* phase until the graph is small enough. Then, a partition of the coarsest graph is obtained in the *initial partitioning* phase. Finally, the contractions are undone in the *uncoarsening* or *refinement* phase while applying local search algorithms or even more advanced methods at every level to improve the existing partition obtained from the coarser level. We now go over these three phases to outline their most important steps.

Coarsening or contraction. The goal of this phase is to reduce the size of the original graph by constructing a series of coarser graphs, so that the initial partitioning can be applied on a simplified graph [43]. Typically, such a coarser graph is constructed by finding and contracting a matching of the graph, i.e. a subset of edges that do not share a common endpoint. Contracting a matching means to merge the two endpoints u and v of every edge $\{u, v\}$ in the matching to a new node w so that $c(w) = c(u) + c(v)$ and $N(w) = (N(u)\backslash\{v\}) \cup (N(v)\backslash\{u\})$. In the case where merging the endpoints produces parallel edges, they are then unified to a new edge whose weight is the sum of the weights of these parallel edges. This ensures that a partition of the coarsest graph yields the same balance and edge-cut when being applied directly on the original graph [8]. Finding and contracting a matching is done repeatedly until the size of the graph is small enough, meaning that the number of nodes in the graph goes below a certain threshold. Ideally, the most important features of the original graph are preserved during the coarsening phase. For social network graphs, it is more suitable to use a label propagation algorithm to directly form and contract clusters of nodes instead of contracting a matching, because it is more aggressive. The contraction of matchings fails to reduce the size of social network graphs efficiently due to its irregular structure [8].

Initial partitioning. This phase applies one (or multiple) initial partitioning algorithms on the coarsest graph. In the case where multiple algorithms are applied, the partition with the best edge-cut quality is chosen [8]. Since the coarsest graph has considerably fewer nodes than the original graph, even slow algorithms are still relatively quick.

Uncoarsening or refinement. After the initial partition has been computed, the coarsest graph is reverted back to the original graph by repeatedly undoing the contractions in the reversed order as they were applied during coarsening phase. At every level, local improvements are performed to further refine the quality of the partition [31]. The idea behind this approach is that, when moving a node on a coarser level to a different block, this translates to moving an entire set of nodes of the original graph to this block. Thus, the local improvement algorithms benefit from a global view on the graph, since they are applied on coarser versions of the original graph [35]. Note that, in general, the applied local improvement algorithms guarantee to not worsen the quality of the existing partition.

CHAPTER 3

Related Work

The field of research for the graph partitioning problem is very active and mainly focuses on constructing algorithms that are based on heuristics such as local search, spectral partitioning or flow computations. We refer the reader to several sources that give either a general overview of the used heuristics [5, 34] or that outline recent advances in the field [7, 8], which include many techniques and approaches that are not topic of this thesis. Some of these techniques are for example evolutionary algorithms, which invest a lot of resources to obtain even better quality, or the use of GPUs to accelerate the computations.
We focus on the well-known multilevel scheme heuristic, which we defined in Section 2.2. It was proposed by Hendrickson and Leland [18] in 1995. The multilevel scheme serves as a basis for the currently most well-known in-memory tools like Metis [21], Scotch [27] or KaHIP [31, 32, 35]. Since the work in this thesis is based on the latter tool and more specifically on one of its programs named KaFFPa, we use Section 3.1 to explain KaFFPa in detail. Note that we stay in the sequential context of the multilevel scheme, even though it can also be used in the context of parallelization [43, 19, 8], where it typically performs worse, since parallel algorithms do not have a global view on the graph [8].
In Section 3.2, we look at *streaming* algorithms, which are the focus of this thesis and which usually stream the nodes one by one together with their edges, instead of loading the entire graph at once. Streaming algorithms have the advantage of using little memory, but they typically yield a lower quality compared to in-memory approaches. Section 3.3 describes whether there exists previous work using budget edge sampling for graph partitioning.

3.1 KaFFPa

KaFFPa (Karlsruhe Fast Flow Partitioner) is one of the programs of KaHIP, which comes with several improvements to the existing multilevel graph partitioning approach [35, 31, 33]. A particularity of KaFFPa is that before searching for a matching in the graph, the edges are rated based on local information to estimate how suitable an edge is for being contracted [31]. The matching algorithm then tries to maximize the sum of the edge ratings. KaFFPa implements its own initial partitioning algorithm that uses the multilevel recursive bisection scheme [35]. Regarding the uncoarsening phase, we outline below the two most important types of local improvement schemes that are implemented by KaFFPa.

Quotient graph style refinement. For this local improvement scheme, instead of looking directly at the graph of the current level, KaFFPa considers its quotient graph. The idea is that *two-way local search* can be applied between two connected nodes i and j of the quotient graph to improve the bipartition between the blocks V_i and V_j sharing a non-empty boundary. For this, KaFFPa uses a variant of the Fiduccia-Mattheyses (FM) algorithm [14] that maintains for every pass two priority queues based on the so-called *gain* of a node, which is the reduction in the edge-cut when moving the node to the other block. The node with the highest gain is moved to the other block, whilst respecting the balance constraint. The FM algorithm runs in linear time and typically needs only a small number of passes [14]. KaFFPa also applies a different, more advanced technique called *adaptive flow iterations* [31, 35], which again looks repeatedly at bipartitions between two blocks V_i and V_j and improves its quality by constructing a max-flow min-cut problem. Since the max-flow min-cut algorithm finds the *minimum* edge-cut between the two blocks, the bipartition can never get worse [33]. This method is extremely effective when trying to produce high-quality partitions [8].

Global k-way local search. This second improvement scheme is also based on the FM algorithm [14]. A first method is the *classic* k-way search, which in contrast to the FM algorithm keeps a more global view. A node can be moved to any other block and not only to one fix neighbouring block as before. The gain of a node v is now given by $g(v) = max_P\{g_P(v)\}$, meaning that for every node v, KaFFPa looks for the block P that causes the highest gain when moving v to P [31]. KaFFPa further extends this notion with the so-called *multi-try* k-way local search [35, 31], which is a more localized variant of the classic k-way search. One of the advantages of this extension is that it has a higher chance to escape local minima.

As previously described, KaFFPa introduces several improvements to the existing multi-level graph partitioning approach, which usually results in a higher partition quality. We outline these extensions briefly in the following.

Label propagation with size constraints. For social network graphs, it is more suitable to use a label propagation algorithm in the coarsening phase to directly form and contract clusters of nodes. The reason is that the usual approach of contracting matchings cannot efficiently reduce the size of a social network graph due to its irregular structure. Contracting the clusters however reduces the size of a graph far more aggressively [8]. The label propagation clustering algorithm was originally proposed by Raghavan et al. [28].

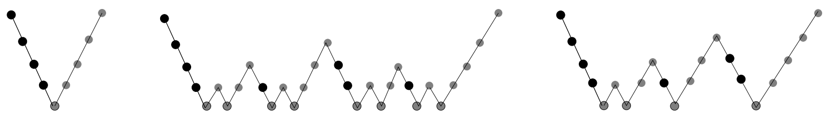


Figure 3.1: A V-cycle, a W-cycle and a F-cycle from left to right. Figure taken from [31].

KaFFPa extends this notion by applying a size-constraint to the clusters [24]. The main reason for this is that it would be impossible to find a feasible initial partition of the graph that still respects the balance constraint if there existed a cluster of size greater than L_{max}. Note that label propagation can also be used as a local improvement algorithm during the uncoarsening phase [33].

Iterated multilevel scheme. The principle of the multilevel scheme can be extended by repeating it with different random seeds for the coarsening and the uncoarsening phase. This is the so-called *iterated* multilevel scheme, which was introduced by Walshaw [42] in 2004. When repeating the multilevel scheme, there exists already a partition from the previous iteration that can directly be used as the initial partition of the next iteration. Since the improvement algorithms in the uncoarsening phase usually do not worsen the initial partition, the iterated multilevel scheme is helpful for finding high-quality solutions [33]. A single iteration of the iterated multilevel scheme is called *V-cycle*, since its behaviour of going down from the original graph to the coarsest graph and then coming back up to the original graph resembles the shape of the letter V. Sanders and Schulz [31, 35] define for KaFFPa two different types of cycles, namely *W-cycles* and *F-cycles*. Figure 3.1 visualizes all three types of cycles.

KaFFPa comes in three different variants, which use other parameters and techniques. Choosing between these three variants roughly relates to performing a time-quality trade-off. We list these variants below.

Strong. This variant tries to achieve the best partition quality. The running time plays a less important role. It performs $\frac{64}{log(k)}$ tries of initial partitioning and uses k-way local search as well as quotient graph style refinement. The multilevel scheme is repeated in the form of two F-cycles.

Eco. The eco variant fills the gap between the variants strong and fast, meaning that it tries to give good partitions in a reasonable amount of time. It still tries multiple initial partitioning algorithms but uses different parameters than the strong variant. Besides, the multilevel scheme is not repeated iteratively, i.e. a single V-cycle is used.

Fast. The goal of this variant is to compute the partition as fast as possible while still trying to beat other fast competitors in terms of quality. It only uses a single initial partitioning algorithm and does not use flow-based algorithms during the uncoarsening phase.

Note that for all three variants one can take the first letter of the name and append the suffix *-social* (i.e. *fsocial*, *esocial* and *ssocial*) to use the label propagation algorithm with the size-constraint cluster contraction in the coarsening phase. This is more effective for social network graphs than the usual matching contraction.

3.2 Streaming Graph Partitioning

Apart from the classical in-memory algorithms, there has been an interest recently in using *streaming* algorithms for graph partitioning [8], which are the focus of this thesis. They were first proposed by Stanton and Kliot [38] in 2012. More specifically, the authors use the *one-pass model*, where the nodes of the input graph G are streamed together with their edges one after the other and directly and permanently assigned to a block. The advantage of streaming algorithms is that they use little memory. However, they typically yield only low-quality solutions, since they do not have a global view on the graph.
Stanton and Kliot [38] introduce many one-pass heuristics for the graph partitioning problem such as *Chunking*, *Hashing* or *Linear Deterministic Greedy (LDG)*. The latter one is generally being preferred over the other heuristics, even though its time complexity of $O(m + nk)$ is worse than e.g. the time complexity $O(n)$ of Hashing. The reason is that LDG produces a far better edge-cut quality than the other proposed heuristics [12]. Nonetheless, LDG still produces low-quality partitions in comparison to in-memory algorithms like KaFFPa. In detail, LDG greedily and permanently assigns a streamed node v to the block V_i that holds the most of its already streamed neighbours while penalizing larger blocks with a linear penalty function p, meaning that:

$$i = \underset{1\leq j\leq k}{\arg\max}\{|N(v) \cap V_j| \cdot p(j)\} \quad \text{with} \quad p(j) = \left(1 - \frac{|V_j|)}{\frac{n}{k}}\right) \tag{3.1}$$

In case of ties, LDG assigns the node to the block with the smallest size. Note that we here only describe the unweighted case, since we only deal with unweighted graphs.
Tsourakakis et al. [41] propose another one-pass heuristic named *Fennel*, which in contrast to LDG uses a penalty function that is additive and non-linear instead of multiplicative and linear. The objective function of Fennel interpolates between the attraction to blocks with more neighbours on the one hand and the repulsion from blocks with more non-neighbours on the other hand. We outline this heuristic in detail in Section 3.2.1.
Nishimura and Ugander [25] introduce *re*streaming algorithms to the graph partitioning problem. These algorithms perform multiple passes through the entire input graph G and hence allow an iterative improvement of the partition. They successfully expand LDG and Fennel respectively to their restreaming variants *ReLDG* and *ReFennel*.
Eyubov et al. [11] propose *FREIGHT*, which is an adaptation of Fennel for the *hyper*graph partitioning problem. In fact, the mathematical definitions of FREIGHT and Fennel are the

same, but the implementation of FREIGHT is significantly faster theoretically and empirically. The authors manage to reduce the time complexity for finding the best block for a streamed node by using a data structure that keeps all blocks sorted by cardinality.
In the context of parallelization, Faraj and Schulz [12] perform recursive multi-sections on the fly. More specifically, their shared-memory streaming algorithm uses multiple passes to partition the graph based on a given hierarchy sequence. Their algorithm is mainly used for process mapping, but it can also be applied on the graph partitioning problem.
Likewise, Battaglino et al. [4] also extend the restreaming notion for the context of parallelization but with the difference that they used distributed memory for their proposed streaming algorithm *GraSP*. After every pass, the partition information of the just finished pass is communicated between all processors.
Awadelkarim and Ugander [2] analyse how the order of (re)streaming the nodes affects the quality of the computed partition in both the sequential and the parallel context. They introduce so-called *prioritized* (re)streaming algorithms which either statically or dynamically (re)stream the nodes. Static (re)streaming means that the order of the nodes is based only on the properties of the input graph and does not need to be updated between iterations. Examples would be BFS, DFS or random ordering. Dynamic (re)streaming reorders the nodes after every iteration based on some priority.

In this thesis, we introduce a new streaming model which is different from the described one-pass model. In fact, other streaming models have been used in previous work besides the one-pass model to solve the graph partitioning problem. For example, Patwary et al. [26] propose *WStream*, which is a greedy streaming algorithm that uses a sliding window of the most recently streamed nodes. The size of the sliding window can be adjusted, but it generally is in the order of a few hundred nodes.
Jafari et al. [20] propose another streaming model in the context of parallelization by introducing a shared-memory streaming algorithm that combines the multilevel paradigm with the buffered approach, meaning that, instead of streaming every node after the other, they load a batch of nodes into a buffer.
Similarly, Faraj and Schulz [13] come up with *HeiStream*, which is a streaming algorithm that also combines the multilevel paradigm with buffering. In contrast to the algorithm proposed by by Jafari et al. [20], they stay in the sequential context, they construct a model instead of processing the nodes of the buffer directly and they base their multilevel scheme on Fennel and not on LDG. We outline HeiStream more in detail in Section 3.2.2.

3.2.1 Fennel

Fennel is a heuristic that tries to bridge the gap between algorithms that are based on precise mathematical work but do not scale well in practice and heuristics that are often used in practice but have no mathematical base [41]. We here only describe the unweighted case, since we deal only with unweighted graphs.

The main idea of Fennel is to formulate a single objective function that simultaneously accounts for the cost of the cutting edges on the one hand and for the cost of imbalanced block sizes on the other hand. Similar to LDG, Fennel assigns a streamed node v directly and permanently to the block V_i that holds the most of its already streamed neighbours while penalizing larger blocks with a penalty function p. The difference to LDG is that the penalty function of Fennel is additive and non-linear instead of multiplicative and linear, meaning that:

$$i = \underset{1 \leq j \leq k}{\arg\max}\{|N(v) \cap V_j| - p(j)\} \quad \text{with} \quad p(j) = \alpha \cdot \gamma \cdot |V_j|^{\gamma-1} \tag{3.2}$$

Note that the assignment is only performed if the overall balance constraint can be respected, otherwise the next best block is considered. The penalty function $p(j)$ can also be interpreted as the marginal cost of increasing the block V_j by one additional node. The authors of Fennel choose $\alpha = m \cdot \frac{k^{\gamma-1}}{n^{\gamma}}$ while stating that this is a good and natural choice but it may not be the most optimal [41].
The parameter γ can be chosen freely in the range $[1, 2]$ and hence the objective function interpolates between the attraction to blocks with more neighbours on the one hand and the repulsion from blocks with more non-neighbours on the other hand. More specifically, γ controls how much a large block will be penalized. Choosing the extreme case $\gamma = 1$ causes the penalty function to be constant, meaning that large blocks will not be penalized at all. Therefore, the objective function can be simplified to $|N(v) \cap V_j|$, meaning that nodes are attracted to blocks with more neighbours. Choosing the other extreme case $\gamma = 2$, large blocks will be penalized linearly. In this case, nodes are repelled from blocks with more non-neighbours. This becomes clearer if one assumes $\alpha = \frac{1}{2}$, because then the objective function is simplified to $|N(v) \cap V_j| - |V_j|$. This is exactly the negative number of non-neighbours in the block V_j, which can be maximized if the block contains no non-neighbours. However, this repulsion is valid for every value of α, since the penalty function gets only weighted differently for other values of α. After analysing the parameter γ, the authors fix $\gamma = \frac{3}{2}$, since it yields the best performance pointwise.
The authors claim that Fennel has a time complexity of $O(m+n)$, since they assume that k is constant. More generally however, Fennel has a time complexity of $O(m+nk)$, because Fennel iterates over all k blocks for every streamed node.

3.2.2 HeiStream

HeiStream tries to fill a different gap than Fennel, which lies also more in the focus of this thesis. More specifically, HeiStream is a compromise between existing sampling algorithms that are fast and use little memory but have a low partition quality on the one hand and offline multilevel algorithms that yield a good partition quality but need a lot of time and memory on the other hand [13]. To achieve this, HeiStream uses the buffered streaming model.

HeiStream first streams a batch of δ nodes together with their edges. Then a model graph $\mathcal{B}$ is built, which represents the nodes of the current batch and the already assigned nodes from previous batches. In fact, HeiStream can build two different models to allow a time-quality trade-off. The *basic* model favours time over quality and is the subgraph of the input graph G induced by the nodes in the current batch. For every iteration except the very first one, the model contains k artificial nodes representing the k different blocks holding the already assigned nodes from previous batches. In our context of unweighted graphs, we have $c(a_i) = |V_i|$ for an artificial node a_i. A node v of the current batch is connected in the model $\mathcal{B}$ to the artificial node a_i if v has a neighbour that was part of the *previous* batch and that got assigned to V_i. Parallel edges will be combined to a single new edge whose weight is the number of parallel edges. The *extended* model considers so-called *ghost edges* to *ghost nodes*, i.e. edges to nodes that will be streamed in future batches. To avoid overloading the memory, these ghost nodes are randomly contracted with one of their neighbours in the current batch. Since the extended model contains more information about G, it usually yields a better partition quality. In this thesis, we only use the extended model of HeiStream.
HeiStream applies the multilevel scheme onto the constructed model graph $\mathcal{B}$. The coarsening phase uses a label propagation algorithm to contract size-constraint clusters. Artificial nodes are ignored and not contracted further, since they each represent an entire block. HeiStream uses the objective function of Fennel for the initial partitioning. Since the algorithm builds a weighted graph as a preprocessing step to the multilevel scheme, Faraj and Schulz had to adjust the objective function of Fennel to the weighted case [13]. In the uncoarsening phase, the same label propagation algorithm is used. The difference is that a node v is moved to the neighbouring block that maximises the Fennel objective and not to the neighbouring block that holds the most neighbours, which would be the usual case. As in the coarsening phase, artificial nodes will be ignored and hence not moved, but they will be considered when computing the Fennel function for other nodes.
After partitioning the model, the nodes from the current batch are permanently assigned to the blocks that were determined by the multilevel scheme. Afterwards, if there are still nodes left that were not streamed yet, the algorithm loads the next batch of nodes and continues the process.
Overall, the running time of HeiStream is $O(n + m)$. Generally speaking, HeiStream outperforms all competitors (Fennel, LDG, Hashing) with regards to partition quality for the majority of instances and especially for huge graphs [13].

3.3 Sampling-Based Graph Partitioning

The algorithm that we propose in this thesis is an *edge sampling* algorithm, meaning that it only uses a selected subset or sample of the edges of the initial graph G. Generally speaking, there exist also *node sampling* algorithms [44] that are beyond the focus of this thesis.

The idea of using edge sampling in the streaming context has previously been used for other problems. For example, Lim and Kang [23] apply edge sampling in their one-pass algorithm *MASCOT* to count local triangles in graphs by sampling the edges at random with a fixed probability p. In contrast, our algorithm samples the edges of the graph by using a budget per node, meaning that every node v samples $min\{b_v, d(v)\}$ of its edges. This budget sampling approach is not totally new and has already been analysed in a similar form under the name of *kN sampling* by Sadhanala et al. [30] in the context of Laplacian Smoothing. The main differences are that they consider weighted and not unweighted graphs and that their algorithm is not designed for the streaming context. In addition, they handle the case of sampling the same edge multiple times by summing up the edge weights appropriately, while we simply keep one of these samples and discard the rest. Sadhanala et al. conclude that kN sampling shows strong empirical performance for their problem. To the best of our knowledge, budget edge sampling has never been used before in the context of streaming graph partitioning.

CHAPTER 4

Streaming Sampling Partitioning

In this chapter we propose a new streaming algorithm for the balanced graph partitioning problem. Our goal is to close the current gap between high-quality in-memory multilevel algorithms and fast streaming approaches. This means that in the best case we manage to obtain a balance and a partition quality that comes close to existing in-memory multi-level algorithms while still being as fast and memory-efficient as current state-of-the-art streaming algorithms.
Our approach does not use the well-known one-pass model, but comes along with a new streaming model that bases on budget edge sampling. Hence, we refer to our algorithm as *SSP* (Streaming Sampling Partitioner). First, we use Section 4.1 to explain our extended streaming model. Then, we outline the overall structure of our algorithm in Section 4.2 by briefly describing the different phases. In Section 4.3, we describe the overall notion of budget edge sampling and define two different sampling methods. Section 4.4 introduces the idea of contracting twins to the (streaming) graph partitioning problem. More specifically, we describe why applying twin contractions as a preprocessing step seems to be good heuristic. Finally, we give a detailed description about the implementation of our algorithm in Section 4.5.

4.1 Extended Streaming Model

In our streaming model, as in the one-pass model, we iterate over the nodes of the input graph G once in sequential order by streaming the nodes one by one together with their edges. We are only allowed to keep $O(n)$ memory and thus cannot load the whole graph into the main memory. However, contrary to the one-pass model, we are not required to make decisions on the fly, i.e. we do not have to assign a node permanently to a block before streaming the next node. We can make all assignment decisions after having streamed the whole graph. To guarantee that we still use only $O(n)$ memory, we perform budget edge sampling while streaming the nodes, which we describe in Section 4.3.

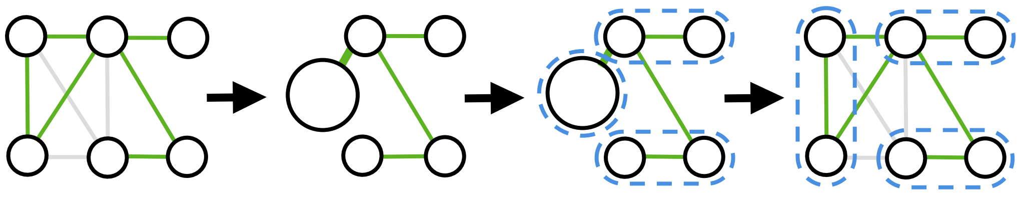


Figure 4.1: A visualization of the four phases of our algorithm. From left to right, we have the following phases: budget edge sampling, twin contraction, partitioning and finally uncontraction with permanent assignment. The sampled edges are marked in green and the blocks in blue.

4.2 Overall Structure

The overall structure of our algorithm can be divided in four different phases, which are visualized in Figure 4.1. First, we apply budget edge sampling to the streamed nodes in order to obtain a simplified sampled graph G_S that only uses memory $O(n)$ and that still preserves the most important features of the initial graph G. In the second phase, we can further simplify the graph by identifying twins in the original graph G and contracting them *after* having sampled the edges. This means that we perform the twin contractions on the sampled graph G_S to obtain the contracted graph G_{SC}. The idea of this novel approach is described in Section 4.4, but in short we assume that twins should be put into the same block, since they share the same neighbourhood and are located very closely in G. Afterwards, the contracted graph G_{SC} is partitioned by an offline in-memory multilevel algorithm. After having undone the twin contractions in the last phase, the partition of G_{SC} serves as a partition of the initial graph G.
In this thesis, we use the state-of-the-art partitioner KaFFPa as the underlying in-memory multilevel algorithm, because it produces partitions that are superior in terms of quality compared to competitors like Metis or Scotch. However, this choice is non-binding and one could replace KaFFPa with any other graph partitioning algorithm.

4.3 Sampling

As usual, our algorithm streams the nodes of the input graph G one after the other along with their edges. For every streamed node, we sample only a subset of its edges. As mentioned, the goal of this approach is to obtain a simplified subgraph G_S with size $O(n)$ that still preserves the most important features of the initial graph G. More specifically, every streamed node v has a budget b_v that indicates the number of edges that will be sampled for v. Note that if v has fewer than b_v edges, we simply sample all of them. So, the total number of edges sampled $s(v)$ is:

$$s(v) = \min\{d(v), b_v\} \tag{4.1}$$

For every streamed node v, SSP performs edge sampling by repeatedly picking among the edges of v until the number of edges sampled equals $s(v)$. The edges are selected randomly and independently. For the randomisation, we use a uniform distribution, meaning that every edge of v is equally likely to get picked. Already picked edges are not considered for the next picks. After having sampled the edges for every node, we construct the subgraph G_S, which is simply the initial graph G but with only the sampled edges. Note that the overall number of edges sampled is always upper bounded by the overall budget $B = \sum_{v \in V} b_v$. Thus the size of the sampled graph G_S lies in $O(n + B)$. Due to practical reasons, we introduce unit node and edge weights to G_S, meaning that every node and every edge in G_S obtains a weight of one. This simplifies the optional twin contraction that we describe in Section 4.4.

Note that every edge may be sampled twice, because it can happen that an edge is sampled independently by both of its endpoints. We handle this case by keeping only one of these two samples, i.e. we sample every edge at most once. This avoids that we obtain multiedges. As a consequence, the real number of edges sampled may be smaller than the overall budget B, because a doubly sampled edge decreases the budgets of both endpoints by one, but we keep only one of the two samples.

SSP can be used with two different sampling methods which differ in the way that they define the budget b_v for a given node v. The default sampling method treats every node identically by using the same constant budget for every node. As an alternative, instead of using the same one-size-fits-all budget for all nodes, we can weigh the budget of every node by its degree. We outline both sampling methods in the following.

4.3.1 Constant Budget

For the default sampling method of our algorithm we define the budget as a single constant value, meaning that every streamed node has the same constant budget b. Thus, for every node v, we can simplify Equation 4.1 by setting $b_v = b$. This means that most of the nodes sample the same constant number b of edges with the exception that low-degree nodes might sample fewer edges. In fact, a constant budget implies that the algorithm always samples all of the edges of nodes with a degree equal or below the constant threshold b. When summing up over all nodes, the overall budget B of the entire graph can be denoted as $B = bn$. Thus regarding the memory usage of the sampled graph G_S, we achieve our previously stated goal of $O(n)$ because b is a constant and therefore $O(n + B) = O(n + bn) = O(n)$.

Tweaking the constant budget b allows us to perform a time-quality trade-off, because increasing b results in sampling more and more edges of the initial graph G. A higher number of edges sampled causes G_S to be a better representation of G and therefore the partition of G_S should still maintain a high quality when being applied directly to G. In the most extreme case, the constant budget b is higher than the highest node degree of the input graph G, which then means that $G_S = G$. In this case, SSP would be identical to directly using the underlying in-memory partitioner on G (if no twin contractions are performed). Increasing the constant budget b most probably comes not only at the cost of higher time usage but also at the cost of higher memory usage.

4.3.2 Weighted Budget

The second sampling method can be considered as a *weighted* case of the first sampling approach, because it also aims for $B = bn$ where B is the overall budget of the graph and b is a (constant) parameter. Note that we can therefore also guarantee a memory usage of $O(n + B) = O(n + bn) = O(n)$ for the second sampling method. However, the overall budget B is not distributed evenly over the nodes as in the first approach. Instead, B is divided among the nodes in proportion to their degrees, meaning that every node v obtains its own budget b_v weighted by its degree. The intuition behind this approach is that every node samples the same fix fraction of its edges, as we explain at the very end of this section. This should generally maintain the degree distribution of the original graph better. More specifically, we assign to a node v a budget b_v that is weighted with respect to its degree $d(v)$:

$$b_v = \left\lceil \frac{bn \cdot d(v)}{2m} \right\rceil \tag{4.2}$$

Note that b_v is defined as the ceiling of a weighted division. The reason for rounding up is that the budget must be an integer value. Rounding down would imply that we assign a budget of zero to nodes that have a very low degree relative to the total number of edges in the graph, which would mean that we would sample none of their edges. Note that without rounding, the overall budget B exactly matches our desired value bn, because:

$$B = \sum_{v \in V} b_v = \sum_{v \in V} \frac{bn \cdot d(v)}{2m} = \frac{bn \cdot \sum_{v \in V} d(v)}{2m} = \frac{bn \cdot 2m}{2m} = bn \tag{4.3}$$

Thus, we would have the exact same overall budget for both sampling methods. However, since the budget of every node is rounded up, the overall budget B of the weighted budget method is in practice slightly *larger* than bn:

Theorem 4.1
When using the weighted budget method, we get the following estimation for the overall budget B:

$$bn \leq B \leq (b + 1)n \tag{4.4}$$

Proof of Theorem 4.1
The first inequation $bn \leq B$ directly follows from Equation 4.3:

$$B = \sum_{v \in V} b_v = \sum_{v \in V} \left\lceil \frac{bn \cdot d(v)}{2m} \right\rceil \geq \sum_{v \in V} \frac{bn \cdot d(v)}{2m} = bn$$

The second inequation $B \leq (b+1)n$ comes from the following observation:

$$B = \sum_{v \in V} b_v = \sum_{v \in V} \left\lceil \frac{bn \cdot d(v)}{2m} \right\rceil \leq \sum_{v \in V} \left(\frac{bn \cdot d(v)}{2m} + 1 \right) = bn + n = (b+1)n$$

Thus we get overall $bn \leq B \leq (b+1)n$. □

This means that, when we use the same value of b for both sampling methods, the weighted case has a slightly *larger* overall budget than the constant case. Since b can be chosen freely and independently for both sampling methods, this can easily be adapted if necessary. For example, we can simply decrease b by one for the weighted case, because then we have a slightly *smaller* overall budget compared to the constant case. For a mathematical comparison of both sampling methods, we refer to Section A.2 of the appendix.
Another way of interpreting Equation 4.2 is that we sample for every node v the same fixed fraction $\frac{bn}{2m}$ of its edges. Generally speaking, the fraction lies in $[0, 1]$ for most graphs and thus the weighted case samples a fixed percentage of the edges of every streamed node. Note that the minimum number of edges sampled for a node with a non-zero degree is always one.

4.4 Twin Contraction

SSP can optionally perform twin contractions to further decrease the size of the graph before passing it down to the underlying in-memory multilevel algorithm. One typically differentiates between true and false twins (see Section 2.1). The intuition behind performing twin contractions is that we expect that twins should be put into the same block in a good partition of a graph. A key factor of this assumption is their *locality* or *closeness*, i.e. twins are distant by at most two edges and thus located very closely in G. In combination with the fact that twins share the same open/closed neighbourhood, putting them into the same block seems to be a reasonable heuristic. This may violate the balance constraint, but we tackle this by introducing a size constraint to the twin clusters, which we explain later in detail. Our algorithm contracts either only true twins or only false twins or both. Note that we identify the twins based on their neighbourhood in the original input graph G. The reason is that we want to perform the sampling as well as the twin identification on the original graph, because we try to preserve the most important features of G as good as possible. The contraction is applied *after* having sampled the edges, meaning that we perform the twin contractions on the sampled graph G_S, even if the nodes might not be twins anymore in G_S. Note that we denote the resulting contracted graph by G_{SC}, which can be understood as the quotient-graph of G_S with respect to the constructed twin clusters.
In detail, we contract every set of twins into a new node. To ensure that the underlying in-memory multilevel algorithm still manages to maintain a good balance for the partition, the weight of the contracted node is set to the sum of the weights of the contracted twins.

Since every node in G_S has a weight of one, the weight of the cluster node is exactly the number of contracted twins. In addition, the neighbourhood of the cluster node is set to $N(t_1)\backslash\{t : t \in T \wedge t \neq t_1\}$ where $T \subseteq V$ is the set of contracted twins and $t_1 \in T$ is any one of the contracted nodes. If the contraction forms parallel edges, then we replace them with a new edge whose weight is the sum of the weights of the parallel edges. Again, since every edge has a weight of one, the weight of the new edge is equal to the number of contracted parallel edges. This is necessary because it guarantees that the computed partition for the contracted graph G_{SC} has the same edge-cut when being applied directly to G_S. Note that if we contract false twins (or both twin types), every isolated node has the same empty open neighbourhood and should per definition be put in the same cluster. However, since this may be counterproductive, we treat isolated nodes separately by making sure that they stay isolated and are not part of any twin contraction. After partitioning the contracted graph G_{SC} with the underlying in-memory multilevel algorithm, we must undo the twin contractions before we can apply the computed partition to the input graph G.

An important detail is that we must introduce a size constraint to the number of twins contracted together, meaning that we define an upper bound to the weight of a contracted node. If a contracted node exceeds this upper bound, it is split up into multiple smaller nodes whose weights are exactly equal to the upper bound except for the last, which simply obtains the remaining weight. The reason for applying a size constraint to the contracted nodes is the same as described in Section 3.1 for the label propagation algorithm of KaFFPa. In detail, the underlying in-memory multilevel algorithm cannot find a feasible partition that still respects the balance constraint if the size of a cluster is greater than $(1+\epsilon) \cdot \lceil \frac{c(V)}{k} \rceil$. To ensure that there is still enough freedom to assign the clusters to the different blocks, we define the upper bound as $\lfloor 5\% \cdot (1+\epsilon) \cdot \frac{c(V)}{k} \rfloor$.

When opting for the third mode (which is the contraction of true *and* false twins), the order in which true and false twins are contracted does not matter and we can even contract both twin types at the same time. This is a consequence of the observation that a node can never simultaneously have true and false twins (see Claim 4.1). This means that the true twin clusters and the false twin clusters never intersect and hence can be contracted independently at the same time.

Claim 4.1

There exists no node v in any undirected graph G that has true and false twins at the same time.

Proof of Claim 4.1

First assume that the node v has a true twin u and a false twin w, meaning that $N[v] = N[u]$ and $N(v) = N(w)$. The latter equation means that v and w share the exact same (open) neighbourhood. In particular, w is also connected to u. However, from the definition of false twins we get that w is not connected to v. This contradicts our assumption that v and u are true twins, since $w \notin N[v] \wedge w \in N[u] \Rightarrow N[v] \neq N[u]$. $\square$

4.5 Implementation Details

4.5.1 Metis File Format

Our algorithm accepts the *Metis File Format*, which is the same format used by KaFFPa [33] and Metis [21]. It has also been used during the 10th DIMACS Implementation Challenge on Graph Clustering and Partitioning [3]. In detail, the format stores a graph with n nodes and m edges in $n + 1$ non-comment lines. The very first line contains two integers n and m, which respectively indicate the number of nodes and the number of (undirected) edges. There exists a third optional integer f for the first line, which indicates whether the graph has node and/or edge weights. We specifically only focus on undirected and unweighted graphs without multiple or self edges. Thus, in our case, the optional integer f is always either omitted or set to zero, which means that no weights are assigned. The remaining n lines store the structure of the graph. When excluding the very first line as well as comments, the ith line contains the list of the neighbouring nodes of the ith node. For undirected graphs, the common approach is to represent every undirected edge between two nodes u and w as two opposite directed edges (u, w) and (w, u). Thus w is included in the list of u and vice versa. However, the integer m in the first line of the format always indicates the number of *undirected* edges, meaning that (u, w) and (w, u) are counted as one single edge and not separately.

4.5.2 Sampling

SSP comes with two streaming modes. The first streaming mode directly streams the nodes of the input graph one after the other from the hard disk, meaning that it follows the default behaviour of common streaming algorithms. The second streaming mode allows reading the entire input graph first into the main memory and then streaming the nodes one after the other from there. This is especially useful when testing the algorithm or when running experiments, because the required IO time is usually lower.
When we stream a node v from the input graph G along with its edges, then we actually only stream the outgoing edges of v and their endpoints, but not their respective opposed incoming edges. This means that the budget edge sampling is only performed on the outgoing edges of v. However, when picking such an outgoing edge, we also sample directly its respective opposite incoming edge. Note that the incoming edge can be quickly constructed by flipping the outgoing edge. This means that we sample *pairs* of opposite directed edges. The reason is that the sampled graph G_S should be a subgraph of the input G and therefore in particular it should be also undirected. Figure 4.2 gives an example on how we sample the edge pairs for a streamed node.
The sampled edge pairs are first stored in an adjacency list, i.e. every node stores a list of its sampled outgoing edges. The idea behind this approach is that for every streamed node v, we can first *append* to its list all of its outgoing edges. If $d(v) \leq b_v$, we are already done and we only have to sample all of the respective incoming edges. Otherwise, we

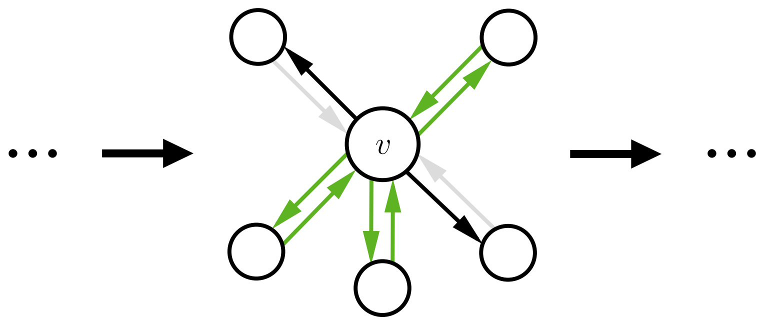


Figure 4.2: A visualization of streaming a node v along with its outgoing edges and their endpoints (in black), but not with the respective opposed incoming edges (in grey). The sampled pairs of outgoing and incoming edges are marked in green. Here we assume that $b_v = 3 \wedge d(v) = 5$.

generate a random index of the last $d(v) - j$ elements using a uniform distribution, where j is the number of edges already sampled. The edge at the generated index i is considered as sampled and swapped with the edge at the last $(d(v) - j)$-th position. After the swap, we increment j by one. We repeat this process until the budget b_v is full. At the end, the non-picked edges are eliminated from the list, which are exactly the last $d(v) - b_v$ elements. This approach can also be seen in the lines 9 to 15 of Algorithm 1. After having streamed all nodes, we use the adjacency list to build the internal graph structure.

4.5.3 Twin Contraction

To identify the twins in the input graph G, we assign to every streamed node v a true twin id $T_t(v)$ and/or a false twin id $T_f(v)$. In fact, we only need both twin ids when we choose the third contraction mode, which simultaneously contracts true *and* false twins. Otherwise, we only need the appropriate twin id. In practice, we compute and sum up the squares of the ids of the nodes in the open/closed neighbourhood of v:

$$T_t(v) = \sum_{u \in N[v]} h(u) \qquad T_f(v) = \sum_{u \in N(v)} h(u) \qquad \text{with} \qquad h(u) = u^2 \tag{4.5}$$

The nodes that have the same twin ids are contracted together *after* having sampled the edges, meaning that we contract the nodes in the sampled graph G_S. This is done by first remapping the discrete twin ids of the nodes to continuous cluster ids starting at zero. In detail, we iterate through the nodes and check for every node v if we have already seen a node u with the same discrete twin id. If this is the case, then we assign v to the existing cluster of its twin u, otherwise we assign v to a new cluster that obtains the next continuous cluster id. For this purpose, we use a `dense_hash_map` from Google[1]. More specifically, every time we cannot find a twin for a node, we create an entry in the

[1]https://github.com/sparsehash/sparsehash.

hash map. The key of the hash map is the discrete twin id and as a value we store three different properties, namely the assigned continuous cluster id, the size of the cluster and the parent node that created the cluster. The size of the cluster is necessary for applying the size-constraint as explained in Section 4.4. Thus, if the addition of a node v to an existing twin cluster exceeds the size constraint, we create a new cluster and replace the values stored in the hash map. This means that for the given hash key we replace the cluster id, we set the size back to one and we set the parent of the cluster to v.
When contracting true *and* false twins simultaneously, we use two different hash maps, one for the true twins and one for the false twins, to avoid unnecessary collisions between both twin types. This means that if we find neither a true twin nor a false twin for a node, we create an entry in both hash maps with the same values. As a consequence of Claim 4.1, where we proved that there does not exist a node that has true and false twins at the same time, we can make a subtle optimization by removing the entry of the cluster parent from the true twin hash map when we find its first false twin and vice versa. Again, this avoids unnecessary hash collisions between both twin types. The lines 23 to 36 of Algorithm 1 give a schematic overview of this approach.
After having remapped every node to its respective continuous cluster id, we perform the actual contraction by computing the quotient graph of G_S with respect to the computed cluster ids. For this, we use an existing internal contraction method of KaFFPa that is capable of computing the quotient graph.

4.5.4 Pseudocode

Algorithm 1 describes the different preprocessing steps of SSP before passing the graph down to KaFFPa. For simplicity, we only focus on the third contraction mode, meaning that we contract true *and* false twins simultaneously. Assuming that the underlying partitioning algorithm needs $O(bn)$ time and assuming that the parameter b is constant, the overall time complexity of our algorithm lies in:

$$O\left(\sum_{v\in V}(d(v)+b_v)+bn\right)=O(2m+bn+bn)=O(m+n) \tag{4.6}$$

When streaming the graph directly from the hard disk, the memory usage lies in $O(n)$, because we stream every node after the other and we keep only $O(n)$ overall sampled edges. When streaming from the main memory, we use $O(n+m)$, because the entire graph is loaded into the main memory.

Algorithm 1: Preprocessing steps of SSP

```
Data: graph $G = (V, E)$, $b > 0$, $k > 0$, imbalance $\epsilon > 0$
Result: sampled and contracted graph $G_{SC}$
foreach $v \in V$ do                                        // stream nodes $\rightarrow O(n)$
    $h \leftarrow 0$
    foreach $u \in N(v)$ do                                 // iterate over edges $\rightarrow O(d(v))$
        append $u$ to adjList[$v$] and set $h \leftarrow h + u^2$
    $T_f(v) \leftarrow h \ \wedge\ T_t(v) \leftarrow h + v^2$
    if $d(v) \leq b_v$ then                                 // sample all edges $\rightarrow O(d(v))$
        foreach $u \in N(v)$ do append $v$ to adjList[$u$]
    else
        $j \leftarrow 0$
        while $j < b_v$ do                                  // randomly sample edges $\rightarrow O(b_v)$
            $i \leftarrow$ random index of one of the last $d(v) - j$ elements of adjList[$v$]
            $t \leftarrow$ adjList[$v$].size $- (d(v) - j)$
            swap adjList[$v$][$i$] with adjList[$v$][$t$]
            increase $j$ by one and append $v$ to adjList[adjList[$v$][$t$]]
        remove the last $d(v) - b_v$ elements of adjList[$v$]     // $\rightarrow O(d(v))$
build $G_S$ from adjList by filtering out doubly sampled edges       // $\rightarrow O(bn)$
$c \leftarrow \lfloor 5\% * (1 + \epsilon) \cdot \frac{c(V)}{k} \rfloor$                      // size constraint of clusters
falseMap $\leftarrow \emptyset \wedge$ trueMap $\leftarrow \emptyset$
foreach $v \in V$ do                                        // remap to continuous cluster ids $\rightarrow O(n)$
    if $d(v) = 0$ then                                      // treat isolated nodes separately
        assign $v$ to a new cluster
    else
        if $T_t(v)$ in trueMap then                         // found true twin $\rightarrow O(1)$
            if trueMap[$T_t(v)$].size $< c$ then
                assign $v$ to the cluster with id trueMap[$T_t(v)$].cluster_id
                increment trueMap[$T_t(v)$].size by one
                remove $T_f$(trueMap[$T_t(v)$].parent) from falseMap
            else
                assign $v$ to a new cluster with id $l$
                trueMap[$T_t(v)$] $\leftarrow$ {cluster_id: $l$, size: 1, parent: $v$}
        else if $T_f(v)$ in falseMap then                   // found false twin $\rightarrow O(1)$
            ...    // analogous to the lines 24 - 30 but for false twins
        else                                                // no twins found $\rightarrow O(1)$
            assign $v$ to a new cluster with id $l$
            trueMap[$T_t(v)$] $\leftarrow$ {cluster_id: $l$, size: 1, parent: $v$}
            falseMap[$T_f(v)$] $\leftarrow$ {cluster_id: $l$, size: 1, parent: $v$}
build $G_{SC}$ by contracting $G_S$ with respect to the remapped cluster ids   // $\rightarrow O(bn)$
```

CHAPTER 5

Experimental Evaluation

In this chapter, we tune our proposed algorithm and compare it to existing state-of-the-art algorithms based on some empirical experiments. We start by outlining the hardware of the machine on which we are running our experiments. Afterwards, we describe our methodology. Then, we indicate the data sets that we use. Finally, we perform the experiments and analyse their results.

5.1 Hardware

We run each instance of our experiments on a single core of a machine that contains an `Intel(R) Xeon(R) Silver 4216` CPU and that has 93 GB of available RAM. The CPU contains 16 cores that each can handle two threads. The clock speed has a base of 2.1 GHz and can reach a minimum of 0.8 GHz as well as a maximum of 3.2 GHz. The machine contains an L2-Cache of 16 MiB and is based on the `x86_64` architecture. It runs `Ubuntu 20.04.1 LTS` with the Linux kernel version `5.4.0-152-generic`.

5.2 Methodology

We divide our experiments into two different phases. First, we perform a series of tuning experiments, to find appropriate configurations of our algorithm. In the second phase, we then compare our algorithm to the current state-of-the-art competitors. For the tuning phase, we start with a base configuration of SSP that uses the constant budget, performs no twin contraction and calls the fast/fsocial variant of KaFFPa. Note that the parameter b is not fixed for our base configuration, as it is determined in the very first tuning experiment. Afterwards, we try to find an appropriate configuration for the remaining parameters. More specifically, we analyse how switching to the eco/esocial variant of KaFFPa influences the behaviour of our algorithm. Then, we determine whether a constant or a weighted budget yields better results. Finally, we compare the effects of the different twin contraction

modes. In the second phase, we compare SSP to the following competitors: KaFFPa [32], HeiStream [13] and Fennel [41]. For the first two algorithms, we take the original implementation from the authors. KaFFPa is preconfigured with the fast/fsocial variant. For HeiStream, we use the extended model with a buffer size of $128 \cdot 1024 = 131072$. Since there does not exist any publicly available version of Fennel, we use an implementation from Faraj and Schulz [13]. They mention that they reproduce the results presented in the original paper of Fennel [41] and that their implementation is optimized for running time as much as possible. We describe the used command-line arguments for all competitors in Section A.1 of the appendix.
All algorithms are written in C++. More specifically, all competitors use C++11, whereas our proposed algorithm uses C++14. Furthermore, all algorithms got compiled with g++ version `9.4.0` using full optimization (`-O3`). Every instance of an experiment is repeated three times using different seeds. The streaming algorithms load the entire graph first into the main memory and then stream the graph from there, because this speeds up the experiments. We exclude the IO time and only focus on the partition time. In total, we measure for every instance the edge-cut of the partition, the balance, the running time and the peak memory usage. For the latter, we use `/usr/bin/time` and extract the maximum resident set size, which is given in kilobytes. Note that the peak memory usage should only be interpreted as a rough indication, because the actual memory usage of streaming algorithms can only be measured when we stream the graphs directly from the hard disk. For all experiments, we allow an imbalance ϵ of 3% and we use $k \in \{4, 16, 64, 256, 1024\}$. With *GNU Parallel* [39], we independently run seven instances in parallel on our machine. The only exception is the huge graph `arabic-2005`, for which we only run three instances in parallel.
Regarding the evaluation, we plot the results most often in the form of a *performance profile*. This type of plot is widely used to compare different objectives such as the edge-cut, the running time or the peak memory usage. For every algorithm A, we plot a line on a 2D graph where the y-axis indicates a fraction of all the instances of this algorithm A, most often expressed in percentage, while the x-axis represents an increasing variable τ starting from one. Every point (f, τ) on the plotted line of algorithm A indicates the fraction f of instances of A whose objective is smaller or equal to τ times the objective of the best algorithm for the same instance. Note that the instances may have different best algorithms. Thus, the point (f, τ) indicates that algorithm A is for a fraction f of its instances never more than τ times worse than the best algorithm on a per-instance level. Figure 5.1 visualizes an example plot. Performance profiles give a broader view on the results and may sometimes lead to other conclusions compared to only looking for example at the geometric mean of the values.
Nonetheless, we use the geometric mean for the generation of so-called *k-improvement plots*, which allow us to compare an algorithm A to a base algorithm B for increasing k with respect to a specific objective. In detail, we first compute the mean of the objective for every algorithm respectively over the three seeds and then we compute the geometric mean g_k over these mean values for every k. Then, we create a plot that contains the

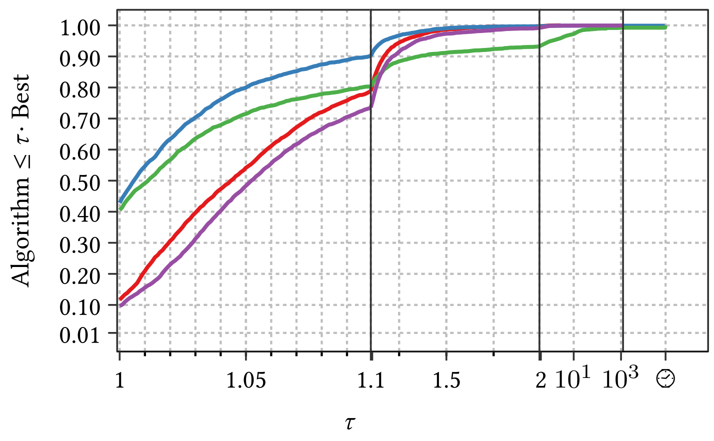


Figure 5.1: A visualization of an example of a performance profile, which was taken from [8]. Four algorithms are plotted in different colours. The algorithm plotted in blue is the best for 40% of the instances and 90% of its instances are no more than 1.1 times worse than the best algorithm on a per-instance level.

improvement $100 \cdot (\frac{g_k \text{ of } B}{g_k \text{ of } A} - 1)$ on the y-axis and the respective k on the x-axis. Again, the objective can be the edge-cut or the peak memory. For the running time, we rather use the k-*speed-up plot*, which has the exact same structure except for the y-axis, where we use the speed-up $\frac{g_k \text{ of } B}{g_k \text{ of } A}$. Whenever we speak of improvement or speedup in the following, we use the just described approach. Note that the geometric mean has the advantage of minimizing the influence of extreme outliers.
For some results, we include the *balance value plots*, which visualize for every algorithm the balance of the partition of every instance in increasing order. In detail, the x-axis indicates the balance for every instance. The instances are sorted with respect to their balance and the y-axis indicates the index of every instance within this sorted order. More specifically, the balance b_i of an instance i is given as:

$$b_i = \frac{\max\limits_{1 \leq j \leq k} \{c(V_j)\}}{\left\lceil \frac{c(V)}{k} \right\rceil} \tag{5.1}$$

Regarding the naming scheme in our plots, we include the chosen configuration in the name of our algorithm. For example, the name `SSP_fast_B3_Tnone_Sdefault` indicates that our proposed algorithm uses the fast variant of KaFFPa with $b = 3$ and no twin contraction. The suffix `_Sdefault` indicates that we use the default sampling mode, which is the constant budget approach. Another possible configuration would be `SSP_fsocial_B2_Ttrue_Sweighted`, which uses the fsocial variant of KaFFPa with $b = 2$, true twin contraction and the weighted budget approach.

5.3 Data Sets

We focus on undirected and unweighted graphs with no multiple or self edges. In addition, we only use graphs with a certain minimum density, i.e. $m \geq 20n$. The idea behind this choice is that for sparser graphs, we would sample a fairly large portion of the edges even for low values of b, meaning that our algorithm would behave very similar to the direct application of our underlying partitioner. In other words, we would not benefit from the budget sampling performed in the preprocessing step of our algorithm.

We use two different data sets. We dedicate one set to the tuning experiments and the other set to the final test experiments. The tuning set contains five non-social graphs and five social graphs. The test set contains ten non-social and ten social graphs. Most of the graphs were already used in previous work on graph partitioning and originate from various sources [29, 16, 37, 3, 9, 22, 6, 40]. If necessary, we converted the graphs in the Metis File Format, we eliminated existing multiple or self edges and we dropped the edge weights. A complete list of all graphs can be found in Table 5.1. Note that we exclude the huge graph `arabic-2005` for the KaFFPa variant tuning, because otherwise the experiments for the esocial variant would take too long.

Regarding the social graphs, we use KaGen [16] to create one artificial 2D random geometric graph (RGG2D). A RGG2D is generated by randomly placing nodes into the unit square of the Euclidean space. Two nodes are connected by an edge if their distance is smaller or equal to a previously defined radius r. For our graph `rgg2d_25`, we choose 1 000 000 nodes and a desired total number of 2^{25} edges, meaning that the radius r is automatically approximated by using a naïve approach of Newton's method. In addition, we generated one artificial 2D random hyperbolic graph (RHG2D) with KaGen. A RHG2D is a generalization of a RGG2D, because its nodes are placed into a hyperbolic space instead of an Euclidean space. For our graph `rhg2d_128`, we choose 1 000 000 nodes and a desired average node degree of 128. We use the default value for the power-law exponent $\gamma = 2.6$, meaning that we get $\alpha = 0.8$.

Since the fsocial variant of KaFFPa is specifically designed for social networks, we use this variant for the social graphs of both data sets. For the remaining graphs, we use the fast variant. The same goes for SSP. HeiStream and Fennel have no particular configuration for social graphs, thus we use the same version across all graphs.

5.4 Tuning Experiments

Starting from a base configuration of SSP, where we use the constant budget, perform no twin contraction and call the fast/fsocial variant of KaFFPa, we perform several experiments to tune the parameters of our algorithm. After having determined the best choice for a specific parameter, we continue with this choice for the subsequent experiments.

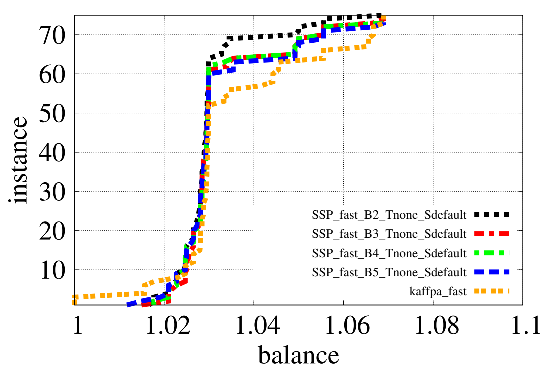
Graph	n	m	Kind	Source(s)
		Non-Social Tuning Graphs		
bcsstk30	28 924	1 007 284	Stiffness	[37]
pdb1HYS	36 417	2 154 174	Protein	[9]
GaAsH6	61 349	1 660 230	Chemistry	[9]
cant	62 208	1 972 466	Meshes	[9]
shipsec5	179 860	4 966 618	Meshes	[9]
		Social Tuning Graphs		
Texas80	31 560	1 219 650	Social Network	[40]
kron_g500-simple-logn20	1 048 576	44 619 402	Artificial	[3]
hollywood-2011	2 180 759	114 492 816	Collaboration	[6]
enwiki-2013	4 206 785	91 939 728	Wikipedia	[6]
arabic-2005[1]	22 744 080	553 903 073	Web	[6]
		Non-Social Test Graphs		
smt	25 710	1 863 737	Meshes	[9]
bcsstk32	44 609	985 046	Stiffness	[37]
Ga3As3H12	61 349	2 954 799	Chemistry	[9]
crankseg_2	63 838	7 042 510	Meshes	[9]
boneS01	127 224	3 293 964	Misc	[9]
bmwcra_1	148 770	5 247 616	Meshes	[9]
audikw1	943 695	38 354 076	Meshes	[9, 3]
ldoor	952 203	22 785 136	Misc	[9, 3]
Flan_1565	1 564 794	57 920 625	Meshes	[9]
Bump_2911	2 852 430	62 409 240	Meshes	[9]
		Social Test Graphs		
Texas84	36 371	1 590 655	Social Network	[40]
Penn94	41 554	1 362 229	Social Network	[40]
livemocha	104 103	2 193 083	Social Network	[22]
libimseti-sorted	220 970	17 233 144	Social Network	[22]
actor-collaboration	382 219	15 038 083	Collaboration	[22]
rhg2d_128	1 000 000	58 906 226	Artificial	[16]
rgg2d_25	1 000 000	33 563 554	Artificial	[16]
dewiki-2013	1 532 354	33 093 029	Wikipedia	[6]
kron_g500-simple-logn21	2 097 152	91 040 932	Artificial	[3]
orkut	3 072 441	117 185 082	Social Network	[29]

Table 5.1: Graphs of the tuning set and the test set

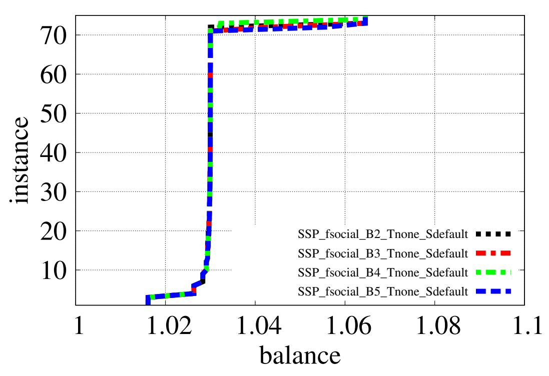
[1] We exclude this graph for the KaFFPa variant tuning, because otherwise the running time for the esocial variant would be too long.

5.4.1 Budget Value

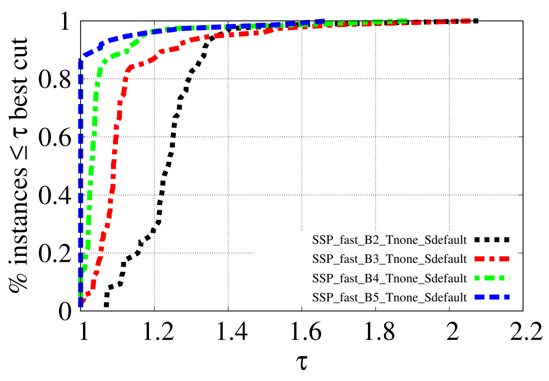
We first start by selecting the right value for the constant budget b. For this, we are evaluating the base configuration of SSP with five different budget values $b \in \{1, 2, 3, 4, 5\}$. In particular, we want to verify our statement of Section 4.3.1, which said that tweaking the constant budget b comes with a time-quality trade-off. More specifically, we assume that a higher value of b yields a better solution quality but comes with a worse running time and a higher memory usage.
Figure 5.2 shows the results of our experiments. For reference, we also add the balance of directly applying KaFFPa on the graphs in the balance value plots. The figure does not contain the performance profiles for the peak memory usage, because we do not stream directly from the hard disk and thus this metric should only be interpreted as a rough indication. Nonetheless, we include these plots in Figure A.2 of the appendix. Note that we remove the results for $b = 1$ from all plots, because this configuration returns unfeasible results, as it is always the fastest, but at the same time it yields an edge-cut that is up to a factor of 40 worse than the best configuration.
When analysing the results, we first observe that all configurations fail to stay below the maximum imbalance of 3% on most non-social instances with $k = 1024$. For the social graphs, only a few instances cannot maintain the balance constraint for large k. However, the imbalance never exceeds 7% regardless of the graph and the chosen budget. In fact, this behaviour can also be observed when directly applying KaFFPa on the graphs. Since our algorithm exceeds the balance constraint in fewer non-social instances compared to the direct application of KaFFPa and since these exceeding imbalances are also less pronounced, we deduce that this behaviour can be attributed to the underlying KaFFPa algorithm. In particular, we assume that switching to the eco/esocial variant would most probably improve the balance, which can be verified with the next tuning experiments.
Besides, we see that our statement about the time-quality trade-off turns out to be true for the non-social as well as for the social graphs. Increasing the constant budget b results in a better edge-cut and a worse running time. Thus, $b = 5$ yields the best solution quality for about 85% and 92% of the instances when using the fast or the fsocial variant respectively. On the other hand, we obtain with $b = 2$ the best running time for 98% of the instances on the non-social graphs and for 93% of the instances on the social graphs. The time-quality trade-off is more pronounced for low budget values, meaning that the difference between $b = 2$ and $b = 3$ is larger than between $b = 4$ and $b = 5$. For this reason, $b = 3$ seems to be the most robust choice, since it is the second fastest configuration on most instances and it yields at the same time a solution quality that lies most often closer to $b = 5$ than to $b = 2$. Furthermore, it has the second-best memory usage on all instances. For this reason, we choose $b = 3$ for the following experiments. However, we keep in mind that tuning the budget allows us to perform a feasible time-quality trade-off.



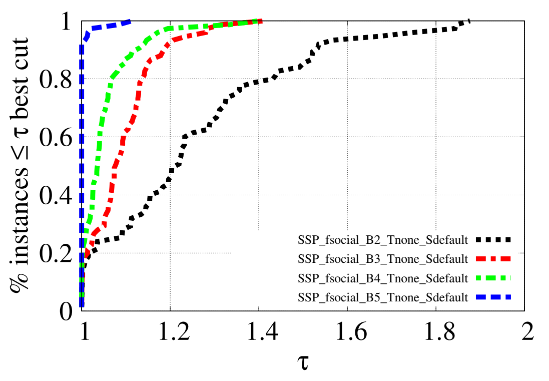
(a) Balance for the non-social graphs.



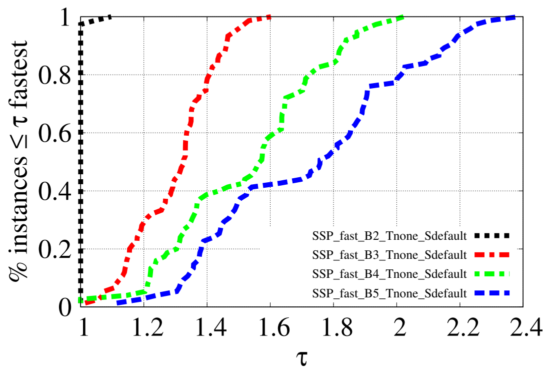
(b) Balance for the social graphs.



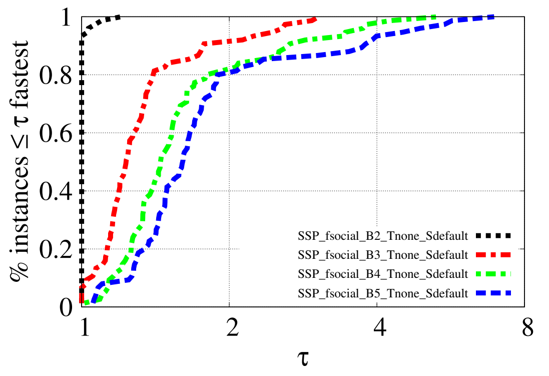
(c) Cut quality for the non-social graphs.



(d) Cut quality for the social graphs.

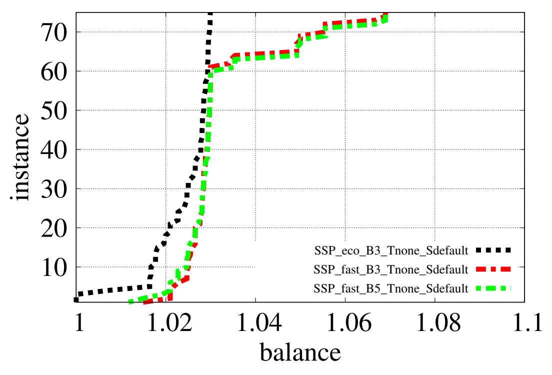


(e) Running time for the non-social graphs.

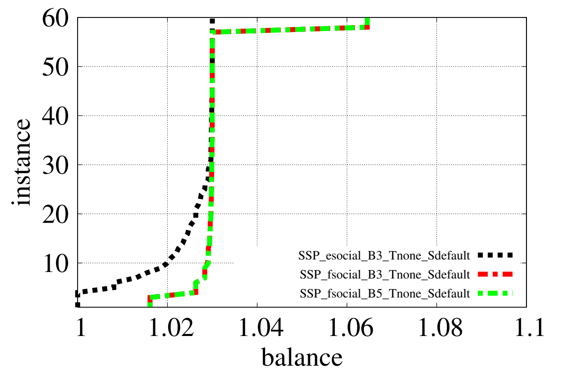


(f) Running time for the social graphs.

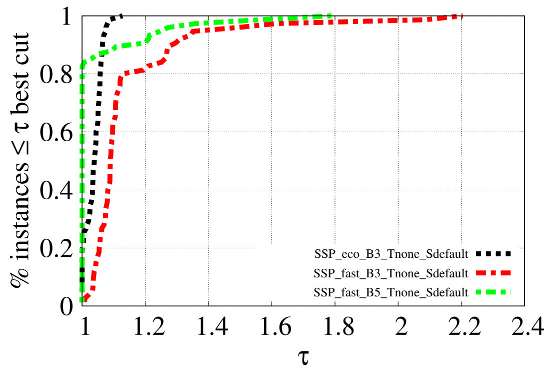
Figure 5.2: Balance value plots as well as cut quality and running time performance profiles for the budget value tuning.



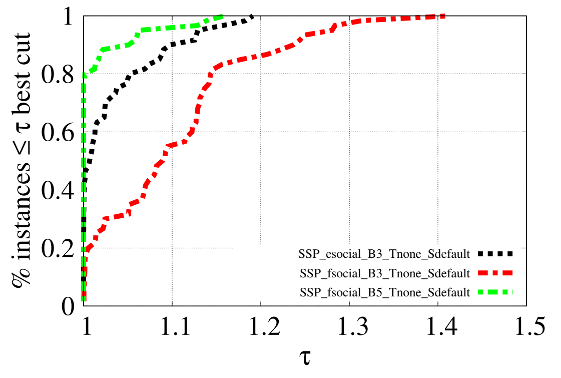
(a) Balance for the non-social graphs.



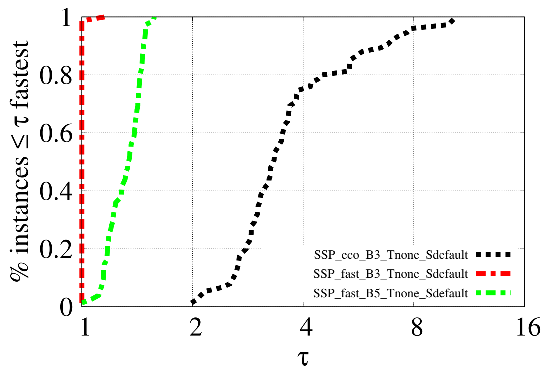
(b) Balance for the social graphs.



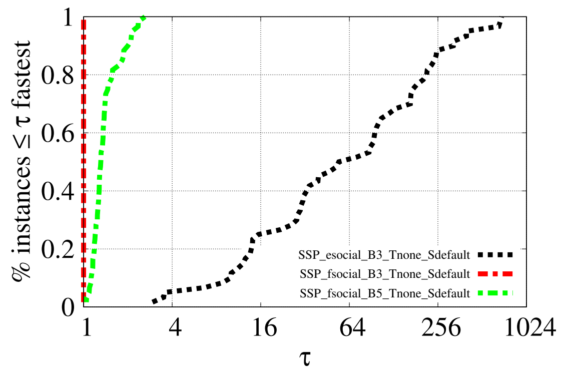
(c) Cut quality for the non-social graphs.



(d) Cut quality for the social graphs.



(e) Running time for the non-social graphs.



(f) Running time for the social graphs.

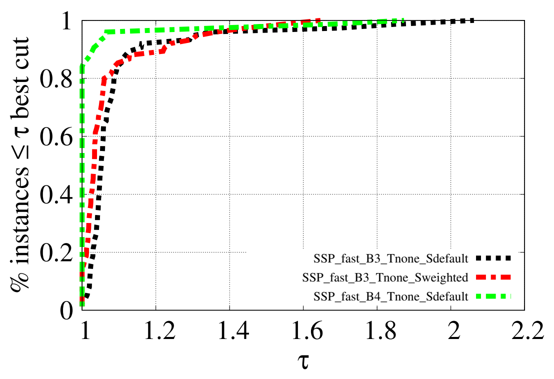
Figure 5.3: Balance value plots as well as cut quality and running time performance profiles for the KaFFPa variant tuning.

5.4.2 KaFFPa Variant

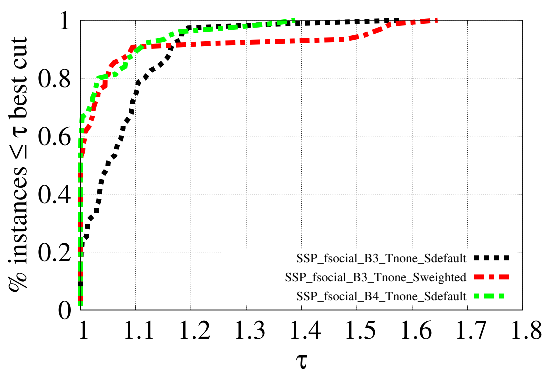
In our second tuning experiment, we want to analyse the variant of the underlying KaFFPa algorithm. On the one hand, we are interested in verifying if switching to the eco/esocial variant for the same constant budget $b = 3$ increases the solution quality while still maintaining a competitive running time. On the other hand, we want to validate our statement from the first tuning experiments, where we assumed that the imbalance of at most 3% is respected when using a stronger variant of KaFFPa. Note that we exclude the huge social graph `arabic-2005` for this tuning experiment, because otherwise the running time for the esocial variant would be too long. For a similar reason, we do not include the strong/ssocial variant of KaFFPa in our experiments, because this variant would take extremely long, even with budget edge sampling. Figure 5.3 shows the results of our experiments. We add the performance profiles for the peak memory in Figure A.3 of the appendix. For reference, we also include the results of SSP with the fast/fsocial variant and a constant budget $b = 5$ in the plots.
First, we observe that the eco/esocial variant of KaFFPa manages to maintain an imbalance of at most 3% for all instances. Thus, our assumption is confirmed for the non-social as well as for the social graphs, because we can improve the balance of the partition by using a stronger variant of KaFFPa. However, besides maintaining the balance constraint, switching to the eco/esocial variant is not a good option. In detail, the eco/esocial variant for $b = 3$ always yields a better edge-cut than its fast/fsocial counterpart. When comparing it to the fast/fsocial variant with $b = 5$, we see that the latter configuration manages to return in about 80% of the non-social and social instances the better solution quality. At the same time, it is still by far faster than the eco/esocial configuration, especially for the social graphs. Thus, it would even be possible to further improve the solution quality by increasing the constant budget b while still being faster than the eco/esocial configuration. We conclude that tuning the constant budget b is a better option than switching to the eco/esocial variant. Therefore, we continue with the fast/fsocial variant for the following experiments. However, if the balance constraint is highly important and the running time is negligible, then the eco/esocial variant might be an option.

5.4.3 Sampling Mode

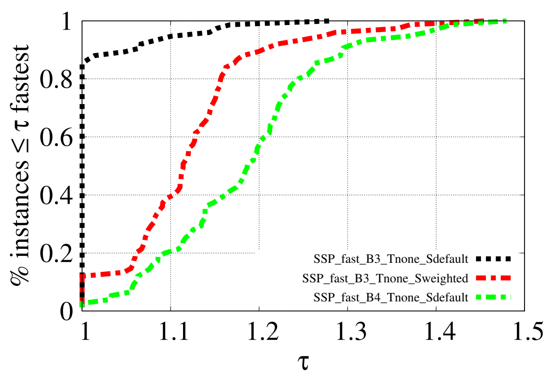
Next, we want to determine the best sampling mode for our algorithm. For this, we analyse how switching to the weighted budget approach compares to our base configuration, which uses the constant budget. For both sampling modes, we stick with $b = 3$ as well as with the fast/fsocial variant. Figure 5.4 shows the results of our experiments. We mainly focus on the edge-cut quality and the running time. Hence, we add the performance profiles for the peak memory and the balance value plots in Figure A.4 of the appendix. For reference, we also include the results of SSP with a constant budget $b = 4$ in the plots.
Focusing on the non-social results, the use of the weighted budget yields on most instances a slightly better edge-cut compared to its constant budget counterpart. However, for 90%



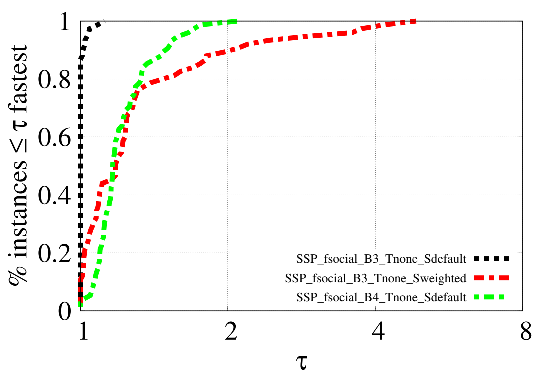
(a) Cut quality for the non-social graphs.



(b) Cut quality for the social graphs.



(c) Running time for the non-social graphs.



(d) Running time for the social graphs.

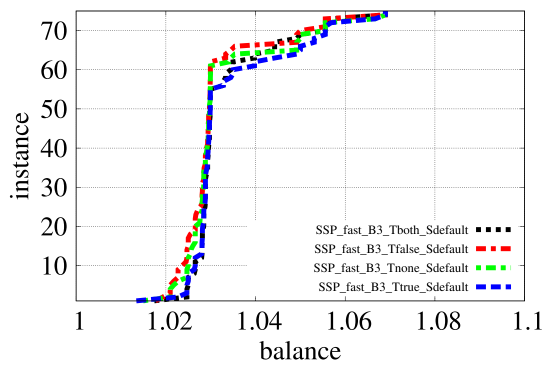
Figure 5.4: Cut quality and running time performance profiles for the sampling mode tuning.

of the instances, the constant budget is never more than 6% worse than the weighted budget in terms of solution quality. In return, the weighted budget needs about 11% more time in the median and has always a slightly worse peak memory. When comparing the weighted budget for $b = 3$ to the constant budget with $b = 4$, we observe that now the weighted budget is slightly faster and uses less memory on most instances but yields also a slightly worse edge-cut quality. Thus, we deduce that using a weighted budget with $b = 3$ is a compromise between the constant budgets $b = 3$ and $b = 4$. This observation validates the proof in Section 4.3.2, which said that the use of the weighted sampling mode yields an overall budget B with $bn \leq B \leq (b + 1)n$. More specifically, we can conclude that the change in the sampling mode has no significant effect and that the observed time-quality trade-off only originates from the fact that we sample different numbers of edges.
A similar observation can be made for the social graphs. In detail, the improvement in the edge-cut quality is slightly more pronounced when switching for $b = 3$ from a constant to a weighted budget, except for the huge graph `arabic-2005`, where the solution quality is up to a factor of about 1.65 worse. At the same time, the loss in terms of running time and

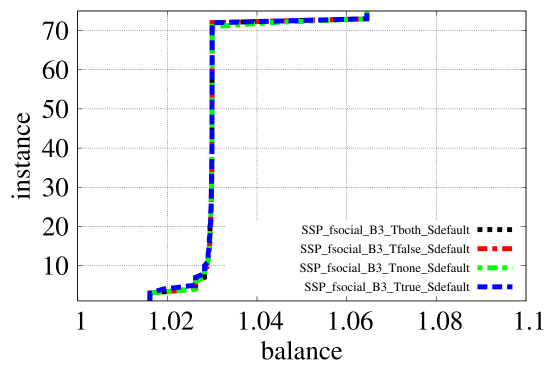
memory efficiency is also larger. In fact, when excluding the huge graph `arabic-2005`, the weighted budget with $b = 3$ is more comparable to the constant budget $b = 4$ than to the constant budget $b = 3$. Nonetheless, we come to the same conclusion, that switching to a weighted budget has no significant positive impact on the results other than the fact that we sample a different number of edges. Especially for the social graphs, we can see that increasing the constant budget is a much better choice than switching to the weighted budget. Thus, we stick with the constant budget for the following experiments.

5.4.4 Contraction Mode

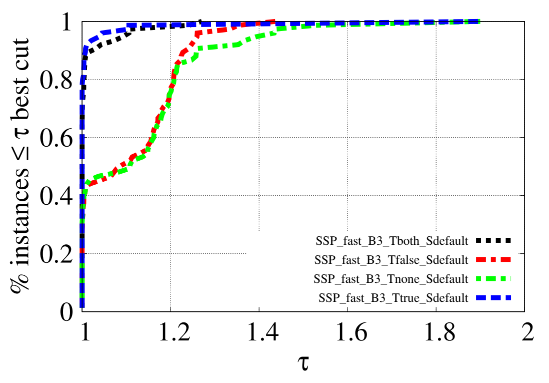
Finally, we compare the four different twin contraction modes while keeping all other parameters fix, meaning that we still use the fast/fsocial variant as well as a constant budget $b = 3$. For simplicity, we use the terms `Tnone`, `Ttrue`, `Tfalse` and `Tboth` to refer to no, only true, only false and both twin types contraction respectively. Figure 5.5 shows the results of our experiments. We add the performance profiles for the peak memory in Figure A.5 of the appendix. We there also include the k-improvement plots for the edge-cut quality and the k-speed-up plots to give a better understanding of the results.
For the non-social graphs, we observe that the pairs `Tboth` and `Ttrue` as well as `Tnone` and `Tfalse` behave very similarly, whereas the first two configurations yield overall better results in terms of edge-cut quality and running time. The latter two configurations are up to a factor of 2.5 slower for some instances. This pairwise similarity can be explained by analysing for every graph how much the number of nodes n is reduced when contracting true or false twins as shown in Table 5.2. Nearly all non-social graphs are not affected by the false twin contraction. This explains why the pairs `Tboth` and `Ttrue` as well as `Tnone` and `Tfalse` behave extremely similarly. Nonetheless, the results for the non-social graphs validate our statement in Section 4.4, where we assumed that performing twin contractions as a preprocessing step is a good heuristic. In fact, we deduce that contracting at least true twins improves the algorithm for any k in terms of solution quality and running time as shown by the k-improvement plots and the k-speed-up plots of the non-social graphs in the appendix.
Regarding the social graphs, `Tboth` and `Ttrue` have no clear dominance over the other two configurations. This can again be explained with the help of Table 5.2, because most social graphs are barely reduced by the true twin contraction. The only exception is the graph `hollywood-11`, for which `Tboth` and `Ttrue` are the most dominant in terms of edge-cut quality and running time. Therefore, we can assume that if a graph can be highly reduced by the true twin contraction, it makes most sense to apply `Tboth` or `Ttrue`. Even if this is not the case, the latter two configurations can usually keep up with `Tnone`. The only exception is the huge graph `arabic-2005`, for which the true twin contraction speeds up the computation, but in return we obtain a worse edge-cut up to a factor of about 2.1. Surprisingly, even if the number of nodes of this huge graph can be reduced to 69.47% of its original size when applying false twin contraction, `Tfalse` is never the fastest configuration for this graph regardless of k and it also yields for most k a worse



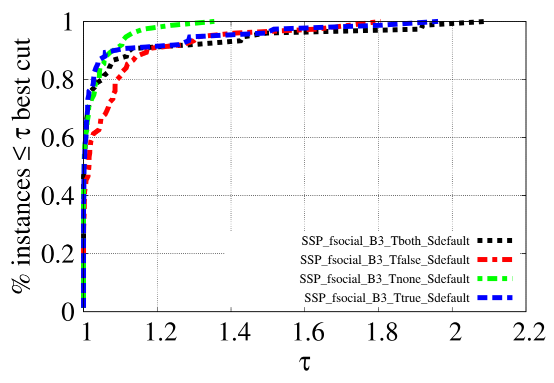
(a) Balance for the non-social graphs.



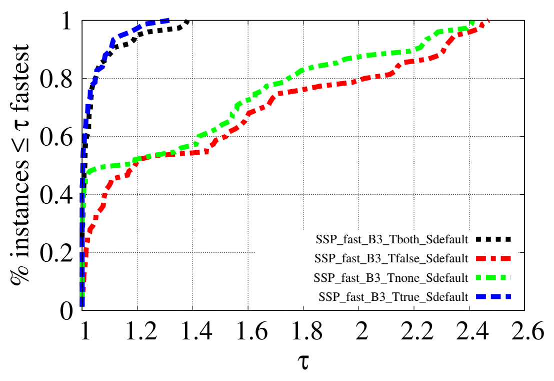
(b) Balance for the social graphs.



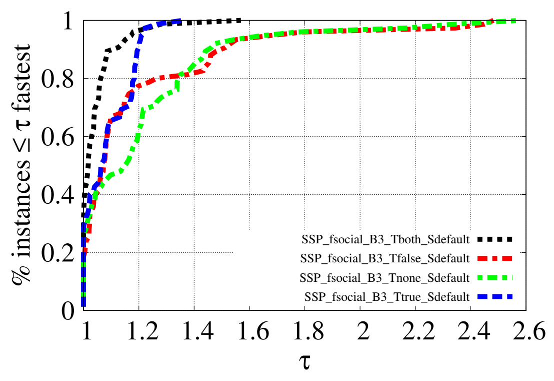
(c) Cut quality for the non-social graphs.



(d) Cut quality for the social graphs.



(e) Running time for the non-social graphs.



(f) Running time for the social graphs.

Figure 5.5: Balance value plots as well as cut quality and running time performance profiles for the contraction mode tuning.

Graph	**n**	$\mathbf{r_t}$	$\mathbf{r_f}$
	Non-Social Graphs		
bcsstk30	28 924	32.12%	99.75%
pdb1HYS	36 417	33.91%	100.00%
GaAsH6	61 349	100.00%	100.00%
cant	62 208	99.55%	100.00%
shipsec5	179 860	16.92%	100.00%
	Social Graphs		
Texas80	31 560	99.94%	99.93%
kron_g500-simple-logn20	1 048 576	99.98%	93.41%
hollywood-2011	2 180 759	54.24%	99.94%
enwiki-2013	4 206 785	99.90%	95.12%
arabic-2005	22 744 080	92.00%	69.47%

Table 5.2: The values r_t and r_f indicate the percentage of n that remains after having performed respectively true or false twin contraction. Note that the twins are identified based on our implementation as described by Equation 4.5. In addition, we apply the size-constraint on the twin clusters with $k = 64$, as outlined in Section 4.4.

solution quality than `Tnone`. Thus, we cannot make the same conclusion for `Tfalse` as for `Ttrue`, because false twin contraction seems to be never a good choice for any of the circumstances.

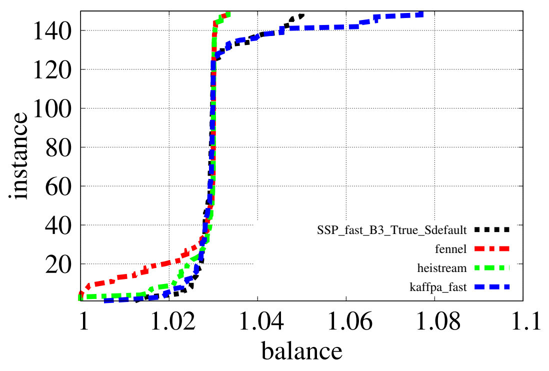
When comparing `Tboth` and `Ttrue`, we cannot observe any huge differences. `Tboth` yields a slightly better balance on non-social graphs and `Ttrue` typically performs slightly better for low values of k. In fact, when looking at the k-improvement plots and the k-speed-up plots, `Ttrue` seems to be overall more robust. In addition, its loss in edge-cut quality for `arabic-2005` is less pronounced. Combined with the fact that we could not find any convincing argument for contracting false twins, we decide to prefer `Ttrue` over `Tboth`. In comparison to `Tnone`, `Ttrue` performs clearly better for the non-social graphs. `Ttrue` is also a better overall choice for social graphs, except for low k and for the huge graph `arabic-2005`. Note that the peak memory is overall slightly better for `Tnone`, but the improvement is negligible. Therefore, we conclude that the best sampling mode amongst all graphs is to only contract true twins, which we also use for the following test experiments. Nonetheless, we keep in mind that there may be scenarios where no twin contraction might be clearly better in terms of edge-cut quality, as we have seen for the huge graph `arabic-2005`.

5.5 Test Experiments

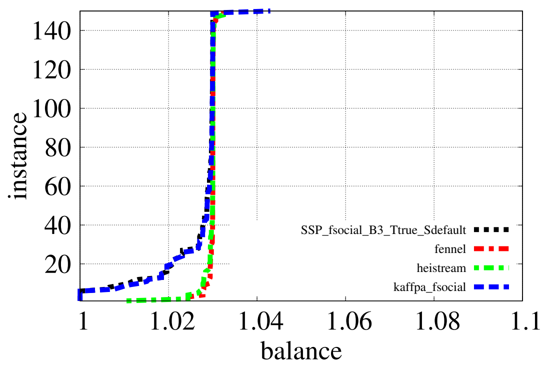
We now compare the final configuration of our algorithm to the state-of-the-art competitors. More specifically, we configure SSP with the fast/fsocial variant of KaFFPa, a constant budget $b = 3$ and the true twin contraction. For the competitors, we select KaFFPa, Fennel and HeiStream. Before analysing the test experiments, our expectation is that KaFFPa should yield the best solution quality on most instances while being generally the slowest. Fennel should be the best in terms of running time, but it should return the worst edge-cuts. HeiStream is expected to be a compromise between these two algorithms and we strive to beat HeiStream with our algorithm. Figure 5.6 shows the results of our experiments. We add the performance profiles for the peak memory in Figure A.6 of the appendix. For a better understanding of the results, we also include the k-improvement plots for the edge-cut quality and the k-speed-up plots in Figure 5.7.

First of all, we observe that SSP fails to stay below the maximum imbalance of 3% for about 23% of the non-social instances with $k = 256$ and for about 60% of the non-social instances with $k = 1024$. All these unbalanced instances belong to the six smallest non-social graphs in the data set in terms of number of nodes n. All instances of the largest four graphs maintain the balance constraint for any k. A similar behaviour can be observed for KaFFPa, however the imbalances are more pronounced. Thus, the preprocessing of SSP helps to improve the balance for the underlying KaFFPa algorithm. For the social graphs, SSP manages to respect the balance constraint for all instances and KaFFPa only exceeds the maximum imbalance of 3% in one single case.

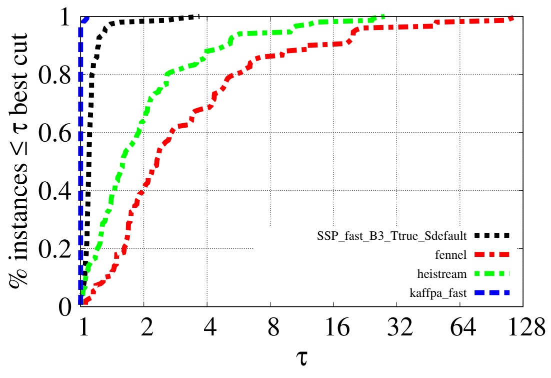
Regarding the non-social graphs, KaFFPa yields as expected the best edge-cut for nearly all instances, but it is overall the slowest. Fennel is the fastest algorithm for about 65.33% of all non-social instances but performs the worst in terms of edge-cut quality. The results show that SSP manages to be overall better than HeiStream for the non-social graphs, especially for low k. In detail, SSP is never more than 15% worse than KaFFPa in terms of solution quality for about 80% of the instances. The remaining 20% are mostly for $k = 4$ with some few instances being for $k = 16$. We can validate these observations when looking at the cut k-improvement plot in Figure 5.7. There, we can also see that SSP has an improvement up to about 200% over the edge-cut quality of HeiStream for low k, which diminishes for large k, since the solution qualities of KaFFPa and HeiStream approach each other for increasing k. SSP yields in about 87.33% of the instances a better edge-cut than HeiStream. All remaining instances where HeiStream beats SSP belong to graphs with less than 65 000 nodes. The median edge-cut quality of HeiStream is about 43.66% worse than that of SSP. In terms of running time, SSP and HeiStream are each the fastest for about 17.33% of the instances. Compared to the other algorithms, HeiStream is especially fast for $k = 1024$, which can also be verified when looking at the k-speedup plot. In contrast, SSP is the fastest on the three largest non-social graphs with $k = 64$ or $k = 246$. The dominance of Fennel in terms of running time diminishes for increasing k, whereas the speedup of HeiStream and SSP increases. In direct comparison, SSP is faster than HeiStream for about 64% of



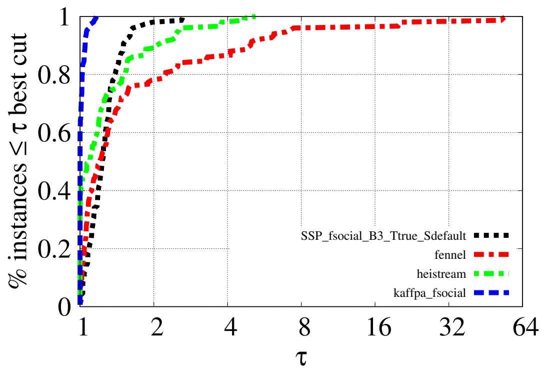
(a) Balance for the non-social graphs.



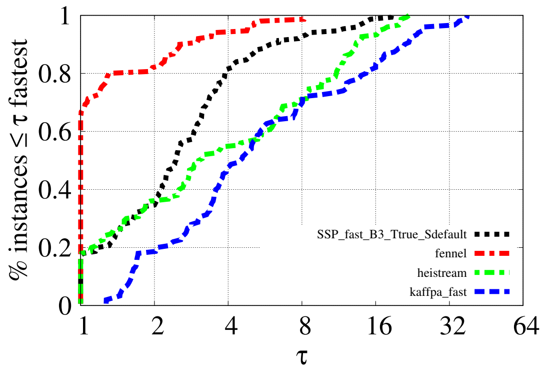
(b) Balance for the social graphs.



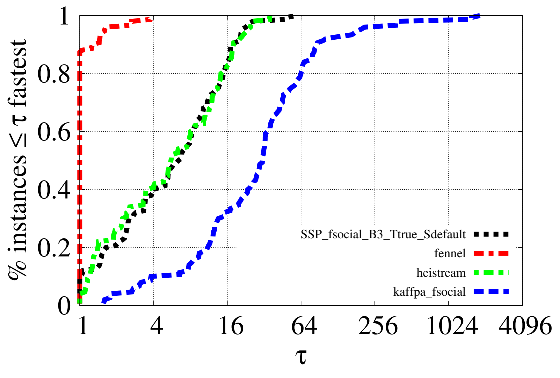
(c) Cut quality for the non-social graphs.



(d) Cut quality for the social graphs.

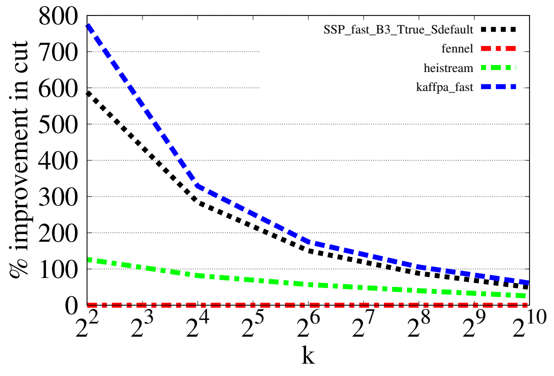


(e) Running time for the non-social graphs.

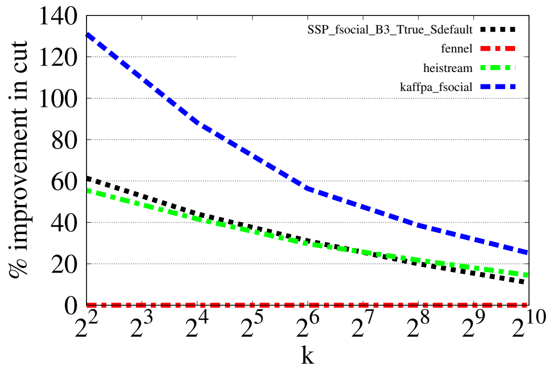


(f) Running time for the social graphs.

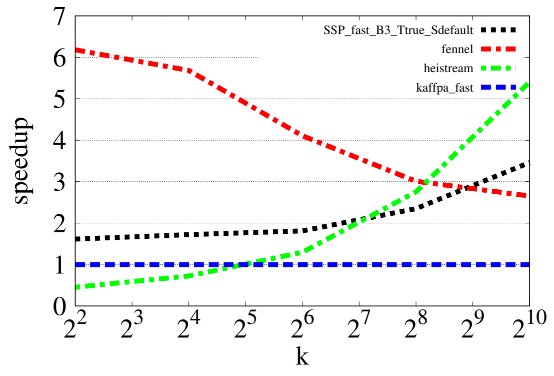
Figure 5.6: Balance value plots as well as cut quality and running time performance profiles for the final test experiments.



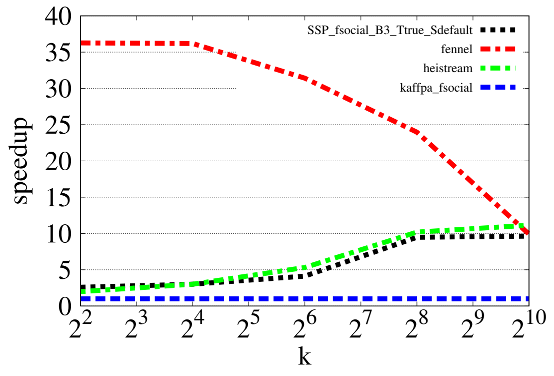
(a) Cut k-improvement for the non-social graphs.



(b) Cut k-improvement for the social graphs.



(c) Time k-speed-up for the non-social graphs.



(d) Time k-speed-up for the social graphs.

Figure 5.7: k-improvement and time k-speed-up plots for the final test experiments.

all non-social instances. Note that SSP performs the worst in terms of edge-cut quality and running time on the graph `Ga3As3H12`, which is also the only non-social graph where the true twin contraction has no effect, as indicated by Table 5.3. Since SSP dominates HeiStream the most on the non-social graph `ldoor`, which is the graph with the highest true twin reduction, it seems that a good performance of SSP is directly correlated to the reduction effect of the true twin contraction.

For the social graphs, we get a different picture. KaFFPa still yields overall the best edge-cut quality, but the improvement is less pronounced and KaFFPa is far slower than the streaming algorithms. Fennel is even more dominant in terms of running time compared to the non-social graphs, since it is the fastest for about 87.33% of all social instances and the gap to the other competitors is also larger. SSP and HeiStream are the fastest for 9.3% and 4% respectively. The k-speedup plot of Figure 5.7 indicates again that HeiStream and SSP have the best relative speedup for $k = 1024$, whereas the dominance of Fennel in terms of running time decreases for high k. However, the most surprising observation is that Fennel yields a better edge-cut quality than SSP for 61.33% of the instances. The biggest

Graph	**n**	$\mathbf{r_t}$	**Graph**	**n**	$\mathbf{r_t}$
Non-Social Graphs			Social Graphs		
smt	25 710	28.90%	Texas84	36 371	99.98%
bcsstk32	44 609	33.22%	Penn94	41 554	99.96%
Ga3As3H12	61 349	100.00%	livemocha	104 103	99.99%
crankseg_2	63 838	18.80%	libimseti-sorted	220 970	100.00%
boneS01	127 224	31.18%	actor-collaboration	382 219	50.70%
bmwcra_1	148 770	36.21%	rhg2d_128	1 000 000	97.50%
audikw1	943 695	33.31%	rgg2d_25	1 000 000	99.07%
ldoor	952 203	14.45%	dewiki-2013	1 532 354	99.92%
Flan_1565	1 564 794	33.33%	kron_g500-simple-logn21	2 097 152	99.96%
Bump_2911	2 852 430	34.08%	orkut	3 072 441	99.94%

Table 5.3: The value r_t indicates the percentage of n that remains after having performed true twin contraction. Note that the twins are identified based on our implementation as described by Equation 4.5. In addition, we apply the size-constraint on the twin clusters with $k = 64$, as outlined in Section 4.4.

exceptions are the two artificial graphs `rhg2d_128` and `rgg2d_25`, where Fennel is up to a factor of 55 worse. Similarly, the solution quality of HeiStream is always better than that of SSP except for the two artificial graphs, where it is up to a factor of four worse. When excluding these two artificial graphs, SSP yields an edge-cut that is on average 17% worse than HeiStream and 7% worse than Fennel. Therefore, SSP is overall dominated by Fennel on the social graphs, which is a huge difference to the behaviour observed for the non-social graphs. There exist two potential reasons for this observation. On the one hand, the dominance of KaFFPa in terms of solution quality is far less pronounced for social graphs, which has also a negative effect on SSP. On the other hand, in contrast to the non-social graphs, most social graphs are barely affected by the true twin contraction, as shown in Table 5.3. Even the most reduced graph `actor-collaboration` is less reduced than nine out of the ten non-social graphs. Nonetheless, SSP still keeps up with HeiStream in terms of running time, since it is faster for exactly 50% of the social instances, which mostly have a low k.
As mentioned, the peak memory usage plots in the appendix only serve as an indication, because we load for all streaming algorithms the entire graph into the main memory. We observe a similar behaviour for the non-social and the social graphs. Fennel uses the least memory on all instances. HeiStream beats SSP on the largest two non-social and social graphs respectively, but it performs much worse on the smaller graphs, where it even uses more memory than KaFFPa. In contrast, SSP is always more memory efficient than KaFFPa. With the help of the peak memory k-improvement plot, we observe that Fennel and SSP improve for large k in terms of memory efficiency in comparison to HeiStream.

CHAPTER 6

Discussion

6.1 Conclusion

In this work, we introduce a new streaming model for the graph partitioning problem that first reduces the size of the input graph G to $O(n)$ by applying budget edge sampling on the streamed nodes and then further simplifies the graph by contracting twins. The resulting graph is partitioned by an underlying in-memory multilevel algorithm, which is in our case KaFFPa, before making the final node assignments during the uncontraction.
Based on our tuning experiments, we conclude that using the same low constant budget b for every node yields the best results. Tuning this parameter b allows us to perform a feasible time-quality trade-off, which enables us to adjust our algorithm to specific use cases. Besides, we provide evidence that performing true twin contraction as a preprocessing step is an effective heuristic in most cases. However, the same cannot be said for false twin contraction. We also conclude that stronger versions of the underlying in-memory multilevel partitioner need much more time in return for little edge-cut improvement, meaning that it is best to stick to the fastest version.
In comparison to current state-of-the-art competitors, our algorithm yields promising results on graphs from scientific applications. More specifically, SSP beats HeiStream on most instances of the selected data set in terms of solution quality, running time and peak memory. Thus, we manage to achieve our goal of further closing the current gap between high-quality in-memory multilevel algorithms and fast streaming approaches. Note that HeiStream is overall better than SSP for $k = 1024$, especially because it is better in respecting the balance constraint. On social networks, we obtain a different picture, because SSP underperforms compared to the competitors in terms of edge-cut quality. There are two reasons for this observation. On the one hand, the dominance of KaFFPa in terms of solution quality is less pronounced on social networks. This has also a negative influence on SSP, since it highly depends on KaFFPa as its in-memory partitioner. On the other hand, the social networks of our data set were barely affected by the true twin contraction. SSP performs in general the best when the graph is strongly reduced by the true twin contraction. Besides, SSP generally favours high k and large, but not huge graphs.

6.2 Future Work

The introduction of budget edge sampling to the streaming context of the graph partitioning problem as well as the establishment of the twin contraction heuristic cover, to the best of our knowledge, previously untouched parts of the research field. Thus, there exist many new approaches to further refine our proposed algorithm. Since SSP falls short on social networks in terms of solution quality, a first tackling point is to improve the edge-cut of the partition on those graphs.
Starting with the sampling phase, one can come up with other sampling methods, such as the use of an amortized budget that allows other nodes to sample more than b edges, if there exist nodes sampling less than b edges due to their low degree. With this approach, we would still respect the overall budget constraint, but we would sample more efficiently, since we would not waste budget on low-degree nodes. Another idea is to introduce the buffered streaming approach to our algorithm, meaning that one streams δ nodes simultaneously and samples $\delta \cdot b$ edges among their union of edges. Instead of randomly sampling the edges, we could use the local information of the streamed buffer to determine which edges might be better suited for being sampled. Here, the main focus lies in finding the right value for δ. Since the weighted budget case described in Section 4.3.2 did not perform much differently from the default constant budget case other than sampling a different number of edges, a more experimental idea is to take the opposite approach of weighting the budget of a node by the inverse of its degree. This intentionally disadvantages high degree nodes and concentrates the overall budget on low degree structures. However, it remains to be analysed whether this approach will lead to a measurable improvement in solution quality.
Regarding the contraction phase, the most promising improvement is to use a better hash function for determining true and false twins. The current implementation of summing up the squares of the node ids may lead to nodes being falsely contracted together. Reducing the hash collisions most probably has a direct impact on the solution quality. To guarantee perfect twin contraction, one could also verify the correctness of the computed twin ids by restreaming the graph. Another idea to improve the solution quality is to first apply the twin contraction on the input graph and then perform the budget edge sampling on the contracted graph. However, the most challenging part of this approach is to maintain $O(n)$ memory, which most probably requires also restreaming the graph.
In terms of running time, a simple improvement is to handle isolated nodes separately. Since it is irrelevant to which block these nodes are assigned, one can simplify the problem by removing them from the input graph and then assigning them to any block after the partitioning, while still respecting the balance constraint. We can proceed similarly for nodes of extremely large degree. In contrast to isolated nodes, it is not completely irrelevant to which block these nodes are assigned, because it is often optimal to put them in the block with the smallest size. However, assigning them to any block does not really affect the edge-cut, since all blocks have roughly the same size due to the existing balance constraint.

APPENDIX A

Appendix

A.1 Command-Line Arguments

We list below the used command-line arguments of every competitor during the final test experiments. Note that `[GRAPH]`, `<k>`, `<seed>` and `<variant>` are placeholders and get replaced with the respective configuration of every instance.

```
$ ./SSP_mem_Ttrue [GRAPH] --variant=<variant> --budget=3 --k=<k>
  ↪ --seed=<seed>
```

```
$ ./kaffpa [GRAPH] --preconfiguration=<variant> --k=<k>
  ↪ --seed=<seed>
```

```
$ ./heistream [GRAPH] --k=<k> --seed=<seed> --stream_buffer=131072
  ↪  --stream_allow_ghostnodes --ram_stream
```

```
$ ./onepasspartition [GRAPH] --one_pass_algorithm=fennel --k=<k>
  ↪ --seed=<seed> --ram_stream
```

A.2 Sampling Modes Comparison

We already observed in Section 4.3.2 that, when choosing the same value of b for both sampling methods, the weighted case has a slightly larger overall budget B than the constant case. However, we want to further compare both sampling methods to better understand their effects in different scenarios. More specifically, we want to mathematically define for both sampling methods the *expected* sampled degree $d_s(v)$ of a given node v, which is the expected number of edges sampled that have one endpoint in v. For this, we first define for every edge $e \in E$ the binary variable x_e as follows:

$$\forall e \in E : x_e = \begin{cases} 1 & \text{if } e \text{ is sampled} \\ 0 & \text{otherwise} \end{cases} \tag{A.1}$$

This means that we can set $d_s(v) = \sum_{e=\{v,u\}\wedge u\in V} x_e$ for a given node v. Note that if v is a low-degree node with $d(v) \leq b_v$, we trivially get $d_s(v) = d(v)$ for both sampling methods. Thus, in the following, we assume that $d(v) > b_v$. For similar reasons, we assume that $\forall u \in N(v) : d(u) > b_u$. For simplicity, we omit the rounding for the weighted case, i.e. we assume that the result of the weighted division is already always integer.

Theorem A.1
For a given parameter b and given node $v \in V$ with $d(v) > b_v \wedge \forall u \in N(v) : d(u) > b_u$, the expected value for the sampled degree $d_s(v)$ is:

$$E[d_s(v)] = \begin{cases} b \cdot \left(1 + \sum_{u\in N(v)} \frac{d(v)-b}{d(v)d(u)}\right) & \textit{for the constant budget} \\ d(v) \cdot \frac{bn}{m} \cdot \left(1 - \frac{bn}{4m}\right) & \textit{for the weighted budget} \end{cases} \tag{A.2}$$

Proof of Theorem A.1
First of all, because of the linearity of expectation, we can write:

$$E[d_s(v)] = E\left[\sum_{e=\{v,u\}\wedge u\in V} x_e\right] = \sum_{e=\{v,u\}\wedge u\in V} E[x_e] = \sum_{e=\{v,u\}\wedge u\in V} P(x_e)$$

Note that $P(x_e)$ is the probability of sampling the edge $e = \{v, u\}$ at least once. For further simplification, we differentiate between $P_v(e)$ and $P_u(e)$, where $P_v(e)$ is the probability of sampling the edge e when streaming v and $P_u(e)$ is the probability of sampling the edge e when streaming u. With the help of basic combinatorial rules, we get:

$$\begin{aligned} P(x_e) = P_v(x_e) + P_u(x_e) - P_v(x_e) \cdot P_u(x_e) &= \frac{\binom{d(v)-1}{b_v-1}}{\binom{d(v)}{b_v}} + \frac{\binom{d(u)-1}{b_u-1}}{\binom{d(u)}{b_u}} - \frac{\binom{d(v)-1}{b_v-1}}{\binom{d(v)}{b_v}} \cdot \frac{\binom{d(u)-1}{b_u-1}}{\binom{d(u)}{b_u}} \\ &= \frac{b_v}{d(v)} + \frac{b_u}{d(u)} - \frac{b_v}{d(v)} \cdot \frac{b_u}{d(u)} \end{aligned}$$

We first focus on the constant budget. Since every node has the same budget b, we can say that $P(x_e) = \frac{b}{d(v)} + \frac{b}{d(u)} - \frac{b^2}{d(v)d(u)}$. Thus we get for the expected value:

$$\begin{aligned} E[d_s(v)] = \sum_{e=\{v,u\}\wedge u\in V} P(e) &= \sum_{u\in N(v)} \left(\frac{b}{d(v)} + \frac{b}{d(u)} - \frac{b^2}{d(v)d(u)}\right) \\ &= b \cdot \left(1 + \sum_{u\in N(v)} \frac{d(v)-b}{d(v)d(u)}\right) \end{aligned}$$

For the weighted case, we can insert Equation 4.2 while omitting the rounding, meaning that we get $P(x_e) = \frac{bn}{m} - \left(\frac{bn}{2m}\right)^2$. The expected value can therefore be written as:

$$\begin{aligned} E[d_s(v)] = \sum_{e=\{v,u\}\wedge u\in V} P(e) &= \sum_{u\in N(v)} \left(\frac{bn}{m} - \left(\frac{bn}{2m}\right)^2\right) \\ &= d(v) \cdot \frac{bn}{m} \cdot \left(1 - \frac{bn}{4m}\right) \end{aligned}$$

□

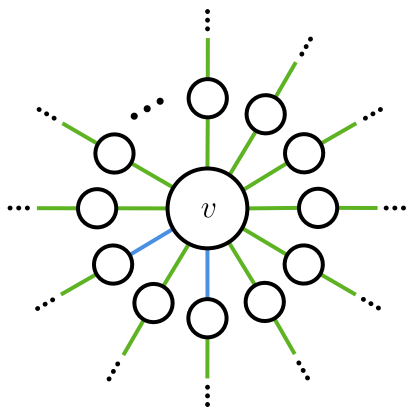
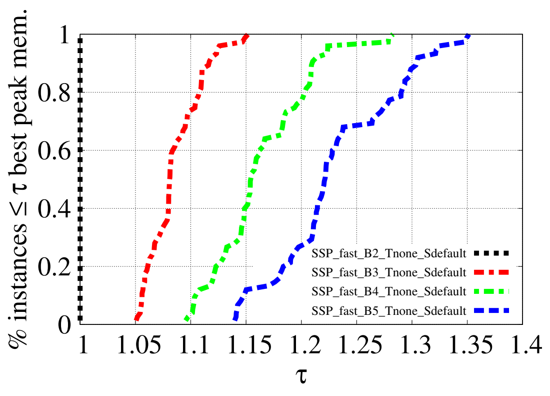


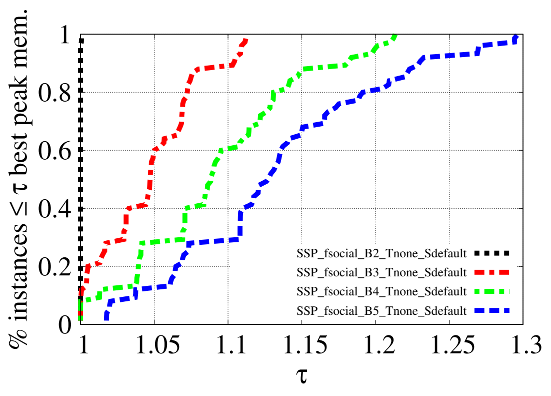
Figure A.1: A visualization of sampling the edges of a star node v with leaves of degree two and a constant budget per node $b \geq 2$. Edges sampled while streaming one of the leaf nodes are green and edges sampled while streaming v are blue.

Theorem A.1 indicates that the expected value of edges sampled for a node v is inversely proportional to the degree of its neighbours when using the constant budget. In contrast, when using the weighted budget, the expected value only depends proportionally on the degree of v. To give a more intuitive example for this observation, let us consider the extreme scenario of a (large) clique with $c + 1$ nodes. Thus, every node in the clique has a degree of c. Again, we assume that $c > b$. This means we get $E[d_s(v)] = b \cdot \left(1 + \frac{c-b}{c}\right) < 2b$ when using the constant budget. Thus, the inverse proportionality to the degree of the neighbouring nodes has a negative effect in this scenario, because the expected number of edges sampled for a node in the clique is always below $2b$. This is *independent* of the size of the clique and therefore the constant budget fails to properly sample large cliques. In contrast, we get $E[d_s(v)] = c \cdot \frac{bn}{m} \cdot \left(1 - \frac{bn}{4m}\right)$ for the weighted case, i.e. the expected number of edges sampled for a node in the clique increases proportionally with c.
On the other hand, if the degree of the neighbours is extremely low, as for example in the case of a star node v, then the use of a constant budget actually yields a higher expected number of edges sampled. Let us consider that $\forall u \in N(v) : d(u) \leq b$ but we have $d(v) > b$. Then we sample *all* of the edges of v when using the constant budget, since all neighbouring nodes sample all of their edges. An example of this scenario is visualized in Figure A.1. In contrast, the expected number of edges sampled for the node v is completely independent of the degrees of the neighbouring nodes when using the weighted budget, with the exception that every leaf samples at least one edge.
Another consequence of Theorem A.1 is that $E[d_s(v)] > b$ when using the constant budget, meaning that we always sample at least b edges. Thus, the constant budget is better in maintaining the connectivity of the input graph G, because low-degree structures with an overall degree less or equal to b are sampled entirely. In contrast, using the weighted budget, we do not have such a constant lower bound. It is only because of the rounding that we sample a minimum of one edge per node. Therefore, sampling with the weighted budget may break low-degree structures like paths into many different pieces.

A.3 Further Results

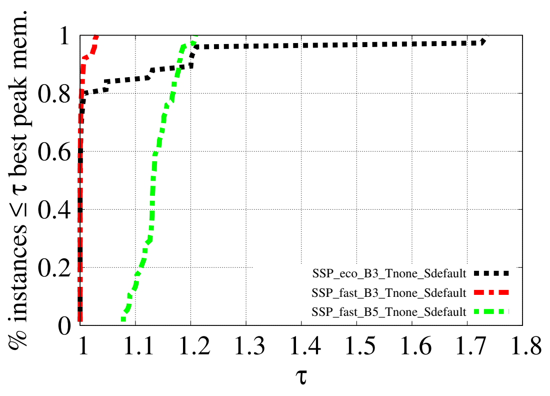


(a) Peak memory for the non-social graphs.

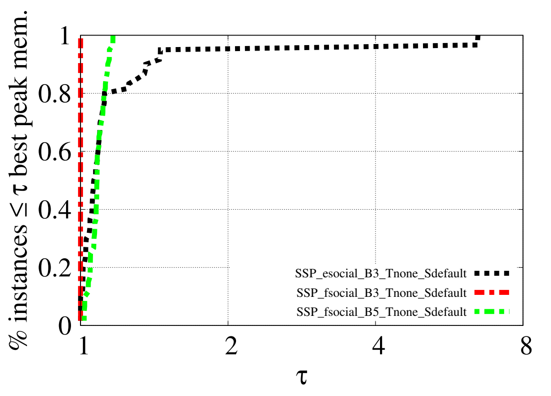


(b) Peak memory for the social graphs.

Figure A.2: Peak memory performance profiles for the budget value tuning.

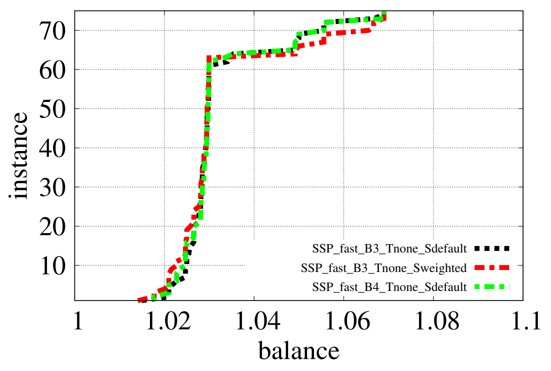


(a) Peak memory for the non-social graphs.

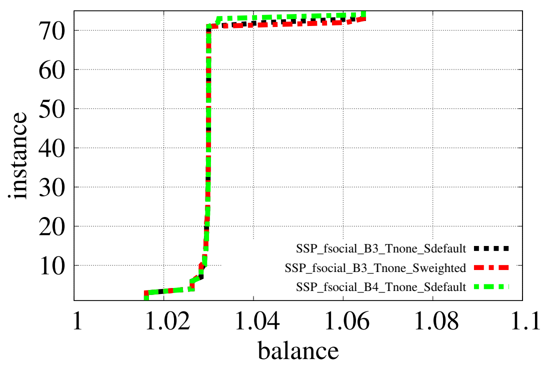


(b) Peak memory for the social graphs.

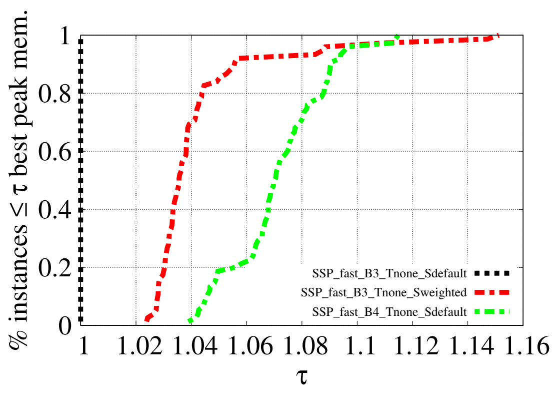
Figure A.3: Peak memory performance profiles for the KaFFPa variant tuning.



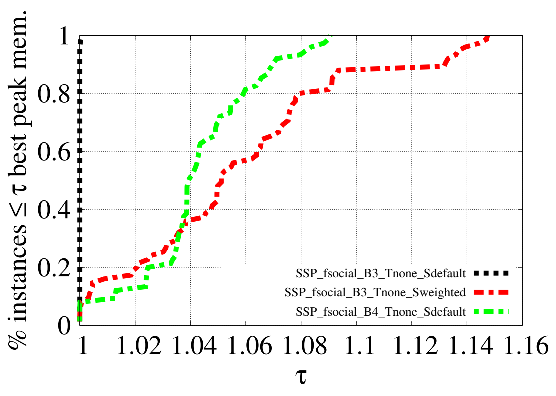
(a) Balance for the non-social graphs.



(b) Balance for the social graphs.

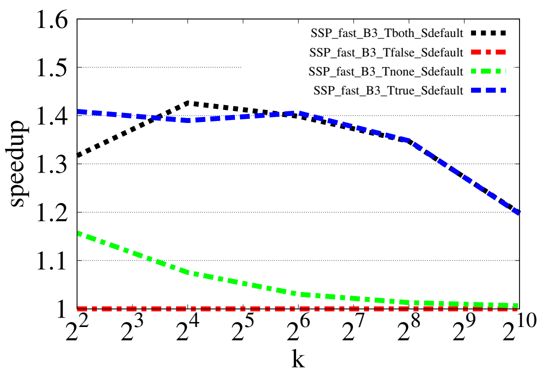


(c) Peak memory for the non-social graphs.

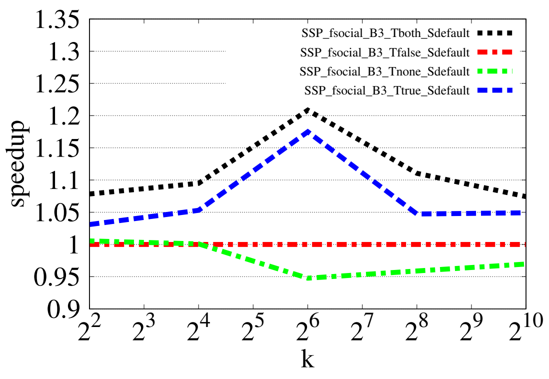


(d) Peak memory for the social graphs.

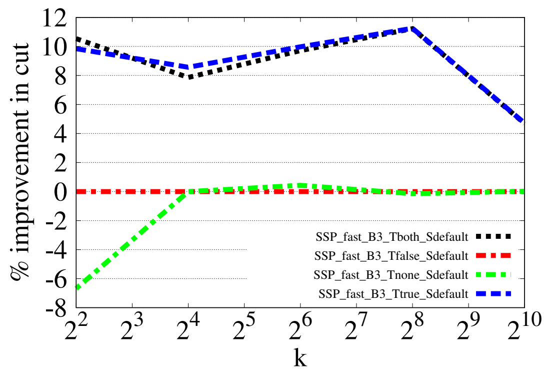
Figure A.4: Peak memory performance profiles and balance plots for the sampling mode tuning.



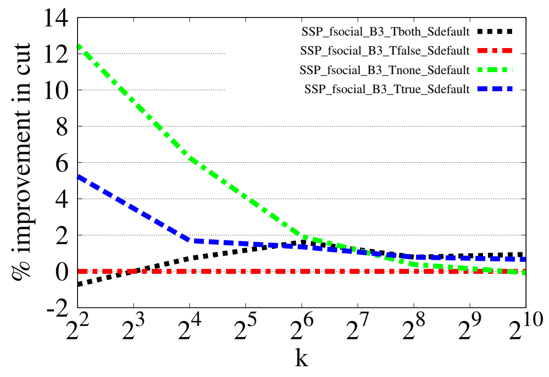
(a) Time k-speed-up for the non-social graphs.



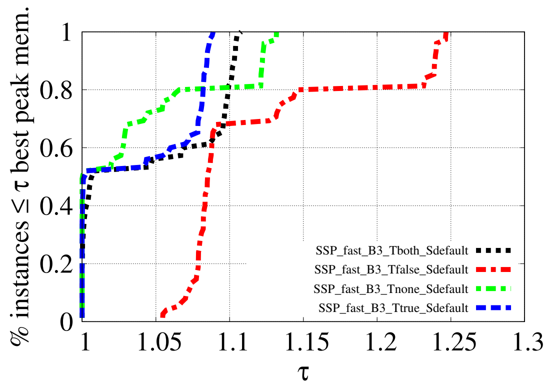
(b) Time k-speed-up for the social graphs.



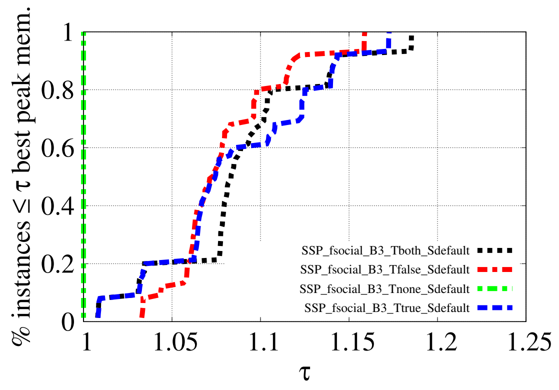
(c) Cut k-improvement for the non-social graphs.



(d) Cut k-improvement for the social graphs.

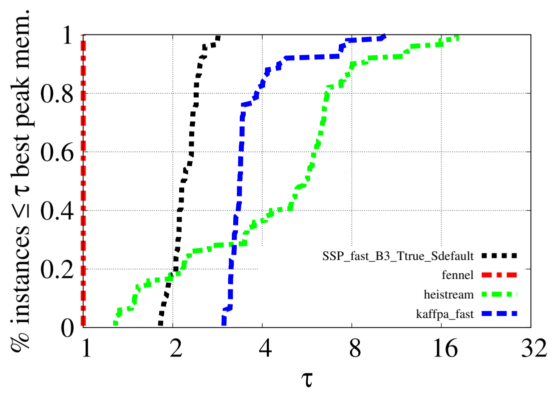


(e) Peak memory for the non-social graphs.

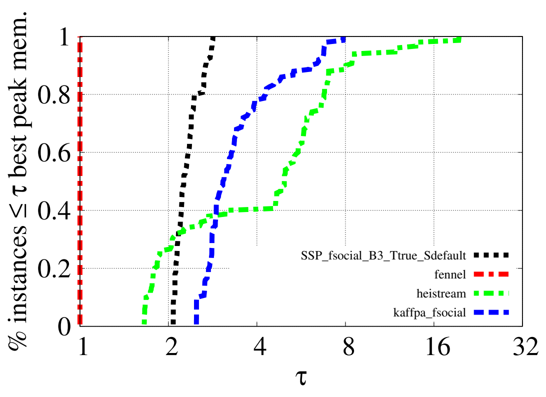


(f) Peak memory for the social graphs.

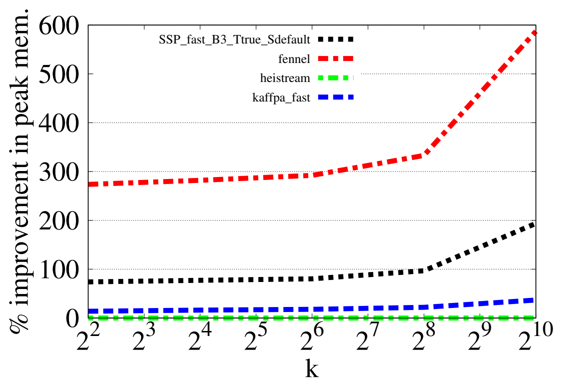
Figure A.5: Peak memory performance profiles as well as cut k-improvement and time k-speed-up plots for the contraction mode tuning.



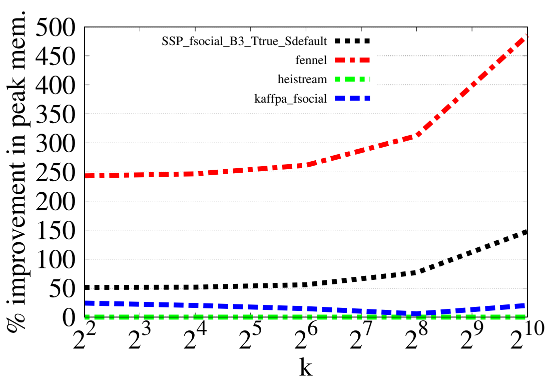
(a) Peak memory for the non-social graphs.



(b) Peak memory for the social graphs.



(c) Mem. k-improvement for the non-social graphs.



(d) Mem. k-improvement for the social graphs.

Figure A.6: Peak memory performance profiles as well as peak memory k-improvement plots for the final test experiments.

Zusammenfassung

Ein gemeinsames Merkmal vieler skalierbarer Algorithmen, welche auf Graphen für die unterschiedlichsten Anwendungen arbeiten, besteht darin, dass sie eine Partitionierung der Graphen voraussetzen. Dies bedeutet, dass die Knoten eines Graphen in k Blöcke von ungefähr gleicher Größe aufgeteilt werden sollen, wobei die Summe der Gewichte der zwischen diesen Blöcken verlaufenden Kanten zu minimieren ist. Der geeignete Algorithmus zur Lösung dieses Partitionierungsproblems hängt jedoch stark vom verfügbaren Speicher des Rechners ab. Aufgrund der wachsenden Größe vieler realer Graphen haben speicherinterne Partitionierer oft Schwierigkeiten, große Netzwerke auf Rechnern mit geringen Speicherkapazitäten zu partitionieren. Aus diesem Grund steigt in letzter Zeit das Interesse an der Verwendung von strömenden Algorithmen für die Graphenpartitionierung, welche einen geringen Speicherbedarf haben, aber typischerweise nur Lösungen von geringer Qualität liefern. In dieser Arbeit gehen wir dieses Qualitätsproblem an, indem wir einen neuen Algorithmus vorschlagen, der ein erweitertes Strömungsmodell verwendet, bei dem alle Knotenzuweisungen erst nach der Strömung des gesamten Graphen durchgeführt werden. Um dies zu erreichen, wird nur eine Teilmenge der Kanten jedes Knotens verwendet, so dass wir eine vereinfachte Version des Eingabegraphen mit $O(n)$ Speicher darstellen können. Wir vereinfachen den Graphen weiter, indem wir Zwillinge, d.h. Knoten mit gleicher Nachbarschaft, kontrahieren. Der daraus resultierende, kontrahierte Graph wird dann durch einen mehrstufigen speicherinternen Algorithmus partitioniert. In dem letzten Schritt werden die Zwillingskontraktionen aufgehoben und die endgültigen Knotenzuweisungen vorgenommen. Der von uns vorgeschlagene strömende Algorithmus übertrifft den aktuellen Stand der Technik für Graphen aus wissenschaftlichen Anwendungen in Bezug auf die Qualität der Kantenschnitte und die Laufzeit.

Bibliography

[1] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. *Theory Comput. Syst.*, 39(6):929–939, 2006. doi: 10.1007/s00224-006-1350-7. URL `https://doi.org/10.1007/s00224-006-1350-7`.

[2] Amel Awadelkarim and Johan Ugander. Prioritized restreaming algorithms for balanced graph partitioning. In Rajesh Gupta, Yan Liu, Jiliang Tang, and B. Aditya Prakash, editors, *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, pages 1877–1887. ACM, 2020. doi: 10.1145/3394486.3403239. URL `https://doi.org/10.1145/3394486.3403239`.

[3] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner, editors. *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*, volume 588 of *Contemporary Mathematics*, 2013. American Mathematical Society. ISBN 978-0-8218-9038-7. doi: 10.1090/conm/588. URL `https://doi.org/10.1090/conm/588`.

[4] Casey Battaglino, Robert Pienta, and Richard Vuduc. GraSP: distributed streaming graph partitioning. In *1st High Performance Graph Mining workshop, Sydney, 10 August 2015*. Barcelona Supercomputing Center, 2015. doi: 10.5821/hpgm15.3. URL `https://doi.org/10.5821/hpgm15.3`.

[5] Charles-Edmond Bichot and Patrick Siarry. *Graph Partitioning*. John Wiley & Sons, Ltd, 2013. ISBN 9781118601181. doi: https://doi.org/10.1002/9781118601181. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118601181`.

[6] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: compression techniques. In Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, pages 595–602. ACM, 2004. doi: 10.1145/988672.988752. URL `https://doi.org/10.1145/988672.988752`.

[7] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. In Lasse Kliemann and Peter Sanders, editors, *Algorithm Engineering - Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 117–158. 2016. doi: 10.1007/978-3-319-49487-6_4. URL `https://doi.org/10.1007/978-3-319-49487-6_4`.

[8] Ümit V. Çatalyürek, Karen D. Devine, Marcelo Fonseca Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. More recent advances in (hyper)graph partitioning. *ACM Comput. Surv.*, 55(12):253:1–253:38, 2023. doi: 10.1145/3571808. URL `https://doi.org/10.1145/3571808`.

[9] Timothy A. Davis and Yifan Hu. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, 2011. doi: 10.1145/2049662.2049663. URL `https://doi.org/10.1145/2049662.2049663`.

[10] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato Fonseca F. Werneck. Customizable route planning. In Panos M. Pardalos and Steffen Rebennack, editors, *Experimental Algorithms - 10th International Symposium, SEA 2011, Kolimpari, Chania, Crete, Greece, May 5-7, 2011. Proceedings*, volume 6630 of *Lecture Notes in Computer Science*, pages 376–387. Springer, 2011. doi: 10.1007/978-3-642-20662-7_32. URL `https://doi.org/10.1007/978-3-642-20662-7_32`.

[11] Kamal Eyubov, Marcelo Fonseca Faraj, and Christian Schulz. FREIGHT: fast streaming hypergraph partitioning. In Loukas Georgiadis, editor, *21st International Symposium on Experimental Algorithms, SEA 2023, July 24-26, 2023, Barcelona, Spain*, volume 265 of *LIPIcs*, pages 15:1–15:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. doi: 10.4230/LIPIcs.SEA.2023.15. URL `https://doi.org/10.4230/LIPIcs.SEA.2023.15`.

[12] Marcelo Fonseca Faraj and Christian Schulz. Recursive multi-section on the fly: Shared-memory streaming algorithms for hierarchical graph partitioning and process mapping. In *IEEE International Conference on Cluster Computing, CLUSTER 2022, Heidelberg, Germany, September 5-8, 2022*, pages 473–483. IEEE, 2022. doi: 10.1109/CLUSTER51413.2022.00057. URL `https://doi.org/10.1109/CLUSTER51413.2022.00057`.

[13] Marcelo Fonseca Faraj and Christian Schulz. Buffered streaming graph partitioning. *ACM J. Exp. Algorithmics*, 27:1.10:1–1.10:26, 2022. doi: 10.1145/3546911. URL `https://doi.org/10.1145/3546911`.

[14] Charles M. Fiduccia and Robert M. Mattheyses. A linear-time heuristic for improving network partitions. In James S. Crabbe, Charles E. Radke, and Hillel Ofek, editors, *Proceedings of the 19th Design Automation Conference, DAC '82, Las Vegas,*

Nevada, USA, June 14-16, 1982, pages 175–181. ACM/IEEE, 1982. doi: 10.1145/800263.809204. URL `https://doi.org/10.1145/800263.809204`.

[15] Jonas Fietz, Mathias J. Krause, Christian Schulz, Peter Sanders, and Vincent Heuveline. Optimized hybrid parallel lattice boltzmann fluid flow simulations on complex geometries. In Christos Kaklamanis, Theodore S. Papatheodorou, and Paul G. Spirakis, editors, *Euro-Par 2012 Parallel Processing - 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings*, volume 7484 of *Lecture Notes in Computer Science*, pages 818–829. Springer, 2012. doi: 10.1007/978-3-642-32820-6_81. URL `https://doi.org/10.1007/978-3-642-32820-6_81`.

[16] Daniel Funke, Sebastian Lamm, Ulrich Meyer, Manuel Penschuck, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free massively distributed graph generation. *J. Parallel Distributed Comput.*, 131:200–217, 2019. doi: 10.1016/j.jpdc.2019.03.011. URL `https://doi.org/10.1016/j.jpdc.2019.03.011`.

[17] M. R. Garey, David S. Johnson, and Larry J. Stockmeyer. Some simplified np-complete problems. In Robert L. Constable, Robert W. Ritchie, Jack W. Carlyle, and Michael A. Harrison, editors, *Proceedings of the 6th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1974, Seattle, Washington, USA*, pages 47–63. ACM, 1974. doi: 10.1145/800119.803884. URL `https://doi.org/10.1145/800119.803884`.

[18] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95, page 28–es, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897918169. doi: 10.1145/224170.224228. URL `https://doi.org/10.1145/224170.224228`.

[19] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. Engineering a scalable high quality graph partitioner. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*, pages 1–12. IEEE, 2010. doi: 10.1109/IPDPS.2010.5470485. URL `https://doi.org/10.1109/IPDPS.2010.5470485`.

[20] Nazanin Jafari, Oguz Selvitopi, and Cevdet Aykanat. Fast shared-memory streaming multilevel graph partitioning. *J. Parallel Distributed Comput.*, 147:140–151, 2021. doi: 10.1016/j.jpdc.2020.09.004. URL `https://doi.org/10.1016/j.jpdc.2020.09.004`.

[21] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

doi: 10.1137/S1064827595287997. URL `https://doi.org/10.1137/S1064827595287997`.

[22] Jérôme Kunegis. KONECT: the koblenz network collection. In Leslie Carr, Alberto H. F. Laender, Bernadette Farias Lóscio, Irwin King, Marcus Fontoura, Denny Vrandecic, Lora Aroyo, José Palazzo M. de Oliveira, Fernanda Lima, and Erik Wilde, editors, *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013, Companion Volume*, pages 1343–1350. International World Wide Web Conferences Steering Committee / ACM, 2013. doi: 10.1145/2487788.2488173. URL `https://doi.org/10.1145/2487788.2488173`.

[23] Yongsub Lim and U Kang. MASCOT: memory-efficient and accurate sampling for counting local triangles in graph streams. In Longbing Cao, Chengqi Zhang, Thorsten Joachims, Geoffrey I. Webb, Dragos D. Margineantu, and Graham Williams, editors, *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*, pages 685–694. ACM, 2015. doi: 10.1145/2783258.2783285. URL `https://doi.org/10.1145/2783258.2783285`.

[24] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Partitioning (hierarchically clustered) complex networks via size-constrained graph clustering. *J. Heuristics*, 22(5):759–782, 2016. doi: 10.1007/s10732-016-9315-8. URL `https://doi.org/10.1007/s10732-016-9315-8`.

[25] Joel Nishimura and Johan Ugander. Restreaming graph partitioning: simple versatile algorithms for advanced balancing. In Inderjit S. Dhillon, Yehuda Koren, Rayid Ghani, Ted E. Senator, Paul Bradley, Rajesh Parekh, Jingrui He, Robert L. Grossman, and Ramasamy Uthurusamy, editors, *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*, pages 1106–1114. ACM, 2013. doi: 10.1145/2487575.2487696. URL `https://doi.org/10.1145/2487575.2487696`.

[26] Md Anwarul Kaium Patwary, Saurabh Kumar Garg, and Byeong Kang. Window-based streaming graph partitioning algorithm. In *Proceedings of the Australasian Computer Science Week Multiconference, ACSW 2019, Sydney, NSW, Australia, January 29-31, 2019*, pages 51:1–51:10. ACM, 2019. doi: 10.1145/3290688.3290711. URL `https://doi.org/10.1145/3290688.3290711`.

[27] François Pellegrini and Jean Roman. SCOTCH: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In Heather M. Liddell, Adrian Colbrook, Louis O. Hertzberger, and Peter M. A. Sloot, editors, *High-Performance Computing and Networking, International Conference and Exhibition, HPCN Europe 1996, Brussels, Belgium, April 15-19, 1996, Proceedings*, volume 1067 of *Lecture Notes in Computer Science*, pages 493–498. Springer,

1996. doi: 10.1007/3-540-61142-8_588. URL https://doi.org/10.1007/3-540-61142-8_588.

[28] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3), 2007. doi: 10.1103/physreve.76.036106. URL https://doi.org/10.1103%2Fphysreve.76.036106.

[29] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In Blai Bonet and Sven Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, pages 4292–4293. AAAI Press, 2015. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9553.

[30] Veeranjaneyulu Sadhanala, Yu-Xiang Wang, and Ryan J. Tibshirani. Graph sparsification approaches for laplacian smoothing. In Arthur Gretton and Christian C. Robert, editors, *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016, Cadiz, Spain, May 9-11, 2016*, volume 51 of *JMLR Workshop and Conference Proceedings*, pages 1250–1259. JMLR.org, 2016. URL http://proceedings.mlr.press/v51/sadhanala16.html.

[31] Peter Sanders and Christian Schulz. Engineering multilevel graph partitioning algorithms. In Camil Demetrescu and Magnús M. Halldórsson, editors, *Algorithms - ESA 2011 - 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings*, volume 6942 of *Lecture Notes in Computer Science*, pages 469–480. Springer, 2011. doi: 10.1007/978-3-642-23719-5_40. URL https://doi.org/10.1007/978-3-642-23719-5_40.

[32] Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*, volume 7933 of *Lecture Notes in Computer Science*, pages 164–175. Springer, 2013. doi: 10.1007/978-3-642-38527-8_16. URL https://doi.org/10.1007/978-3-642-38527-8_16.

[33] Peter Sanders and Christian Schulz. KaHIP v3.00 – Karlsruhe high quality partitioning user guide. *CoRR*, abs/1311.1714, 2020. URL http://arxiv.org/abs/1311.1714.

[34] Kirk Schloegel, George Karypis, and Vipin Kumar. Graph partitioning for high performance scientific simulations. Technical report, 2000. URL https://hdl.handle.net/11299/215407.

[35] Christian Schulz. *High Quality Graph Partitioning*. PhD thesis, Karlsruhe Institute of Technology, 2013. URL `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000035713`.

[36] Christian Schulz and Darren Strash. Graph partitioning: Formulations and applications to big data. In Sherif Sakr and Albert Y. Zomaya, editors, *Encyclopedia of Big Data Technologies*. Springer, 2019. doi: 10.1007/978-3-319-63962-8_312-2. URL `https://doi.org/10.1007/978-3-319-63962-8_312-2`.

[37] Alan J. Soper, Chris Walshaw, and Mark Cross. A combined evolutionary search and multilevel optimisation approach to graph-partitioning. *J. Glob. Optim.*, 29(2):225–241, 2004. doi: 10.1023/B:JOGO.0000042115.44455.f3. URL `https://doi.org/10.1023/B:JOGO.0000042115.44455.f3`.

[38] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In Qiang Yang, Deepak Agarwal, and Jian Pei, editors, *The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, Beijing, China, August 12-16, 2012*, pages 1222–1230. ACM, 2012. doi: 10.1145/2339530.2339722. URL `https://doi.org/10.1145/2339530.2339722`.

[39] Ole Tange. GNU parallel: The command-line power tool. *login Usenix Mag.*, 36(1), 2011. URL `https://www.usenix.org/publications/login/february-2011-volume-36-number-1/gnu-parallel-command-line-power-tool`.

[40] Amanda L. Traud, Peter J. Mucha, and Mason A. Porter. Social structure of facebook networks. *CoRR*, abs/1102.2166, 2011. URL `http://arxiv.org/abs/1102.2166`.

[41] Charalampos E. Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. FENNEL: streaming graph partitioning for massive scale graphs. In Ben Carterette, Fernando Diaz, Carlos Castillo, and Donald Metzler, editors, *Seventh ACM International Conference on Web Search and Data Mining, WSDM 2014, New York, NY, USA, February 24-28, 2014*, pages 333–342. ACM, 2014. doi: 10.1145/2556195.2556213. URL `https://doi.org/10.1145/2556195.2556213`.

[42] Chris Walshaw. Multilevel refinement for combinatorial optimisation problems. *Ann. Oper. Res.*, 131(1-4):325–372, 2004. doi: 10.1023/B:ANOR.0000039525.80601.15. URL `https://doi.org/10.1023/B:ANOR.0000039525.80601.15`.

[43] Chris Walshaw and Mark Cross. JOSTLE: parallel multilevel graph-partitioning software–an overview. *Mesh partitioning techniques and domain decomposition techniques*, 10:27–58, 2007. doi: 10.4203/CSETS.17.2. URL `https://doi.org/10.4203/CSETS.17.2`.

[44] Tianyi Wang, Yang Chen, Zengbin Zhang, Tianyin Xu, Long Jin, Pan Hui, Beixing Deng, and Xing Li. Understanding graph sampling algorithms for social network analysis. In *31st IEEE International Conference on Distributed Computing Systems Workshops (ICDCS 2011 Workshops), 20-24 June 2011, Minneapolis, Minnesota, USA*, pages 123–128. IEEE Computer Society, 2011. doi: 10.1109/ICDCSW.2011.34. URL `https://doi.org/10.1109/ICDCSW.2011.34`.