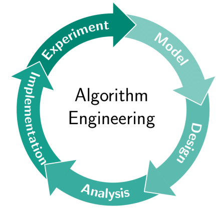
KIT
Karlsruhe Institute of Technology



Star Partitioning

Improving Multilevel Graph Partitioning using Peripheral Nodes

Master's Thesis of

Nikolai Maas

Advisors: Prof. Dr. Peter Sanders
Dr. Tobias Heuer

08.02.2023

Institute of Theoretical Informatics, Algorithm Engineering
Department of Informatics
Karlsruhe Institute of Technology

I declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned and I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I adhered to the rules for ensuring good scientific practice in their current version.

Karlsruhe, February 8th, 2023

Abstract

The *balanced graph partitioning problem* is to partition the nodes of a graph into k disjoint blocks of bounded weight, while simultaneously minimizing an objective function defined on the edges (e.g., the number of edges connecting two blocks). The graph partitioning problem is NP-hard and there also exists no constant factor approximation. The partitioning community has long focused on instances with a regular structure, e.g., mesh graphs or instances from circuit design. However, it becomes more and more important to find high-quality solutions for instances with an irregular structure, such as those derived from social networks. Surprisingly, we found a subclass of these instances where current state-of-the-art partitioning algorithms compute solutions that are far from optimal. The identified instances – referred to as *star instances* – are characterized by a core of highly-connected nodes (core nodes) with only sparse connections to the remaining nodes (peripheral nodes).

In this work, we consider the graph partitioning problem restricted to star instances. Our main theoretical result is an $(R + 1)$-approximation that is achieved by solving a min-knapsack problem for each block of the partition, where R is the ratio of an approximation algorithm for the min-knapsack problem.

On the practical side, we identify shortcomings of existing partitioning algorithms for star instances. These algorithms are all based on the *multilevel* scheme, which consists of three phases: In the coarsening phase, edge contractions are used to reduce the size of the input graph until it is small enough. Afterwards, an initial partition is computed which is thereafter refined while reversing the contractions. We then integrate our theoretical findings into the shared-memory multilevel algorithm Mt-KaHyPar to find high quality partitions for star instances. The main components of our new solution are (i) several strategies to identify peripheral nodes, (ii) a two-hop clustering scheme to efficiently reduce the number of peripheral nodes in the coarsening phase, and (iii) the integration of our approximation algorithm into initial partitioning to find high-quality solutions.

Compared to the state-of-the-art, our resulting configuration improves the quality by 50% (up to a factor of ten) on 60% of the identified star instances. However, the improvement on star instances comes at the cost of reduced quality on general graphs. We therefore propose to use our new techniques within an portfolio of algorithms and use effectiveness tests to show that it provides high quality solutions on a diverse set of benchmark instances.

Zusammenfassung

Das Problem der *balancierten Graphpartitionierung* besteht darin, einen Graph in k disjunkte Blöcke mit beschränktem Gewicht zu unterteilen, wobei eine Zielfunktion auf den Kanten minimiert wird (z.B. die Anzahl der Kanten zwischen den Blöcken). Das Graphpartitionierungs-Problem ist NP-schwer und es existiert auch keine Approximation mit einem konstanten Faktor. Die Partitionierungs-Community hat sich lange auf Instanzen mit regulärer Struktur konzentriert, beispielsweise Gittergraphen oder Instanzen aus dem Schaltkreisentwurf. Jedoch wird es immer wichtiger, auch für Instanzen mit irregulärer Struktur hochqualitative Lösungen zu finden, beispielsweise Graphen von sozialen Netzwerken. Zu unserem Erstaunen fanden wir eine Teilklasse von Instanzen, auf denen die Lösungen, die von Partitionierungs-Algorithmen auf dem Stand der Technik berechnet werden, weit vom Optimum entfernt sind. Diese Instanzen – bezeichnet als *Stern-Instanzen* – sind durch einen Kern von hochgradig verbundenen Knoten (Kernknoten) mit nur wenigen Kanten zu den übrigen Knoten (periphere Knoten) gekennzeichnet.

Diese Arbeit befasst sich mit dem Graphpartitionierungs-Problem eingeschränkt auf Stern-Instanzen. Unser wichtigstes theoretisches Ergebnis ist eine $(R+1)$-Approximation, welche wir durch das Lösen eines Min-Knapsack-Problems für jeden Block der Partition erreichen. Dabei ist R der Approximationsfaktor eines Approximationsalgorithmus für das Min-Knapsack-Problem.

In praktischer Hinsicht identifizieren wir Probleme existierender Partitionierungs-Algorithmen mit Stern-Instanzen. Diese Algorithmen basieren auf dem *Multilevel-Paradigma*, welches aus drei Phasen besteht: Während dem *Coarsening* wird der Graph durch Kantenkontraktionen verkleinert. Sobal eine ausreichend geringe Größe erreicht ist, wird eine initiale Lösung berechnet. Diese wird im Anschluss verfeinert, während die Kontraktionen in umgekehrter Reihenfolge rückgängig gemacht werden. Anschließend integrieren wir unsere theoretischen Ergebnisse in den parallelen Multilevel-Algorithmus Mt-KaHyPar, um hochwertige Lösungen auf Stern-Instanzen zu finden. Die Haupbestandteile unserer neuen Lösung sind (i) mehrere Strategien zur Erkennung peripherer Knoten, (ii) eine Technik zur *Two-Hop*-Clusterbildung, die während dem Coarsening effizient die Anzahl peripherer Knoten reduziert, und (iii) der Einsatz unseres Approximationsalgorithmus, um eine hochwertige initiale Lösung zu berechnen.

Im Vergleich zum Stand der Technik erreicht unsere Konfiguration eine Verbesserung um 50% (bis zu einem Faktor von zehn) auf 60% der Stern-Instanzen. Allerdings ist zugleich die Qualität auf einigen der anderen Instanzen schlechter. Um dies zu lösen, integrieren wir unsere Techniken in ein Algorithmen-Portfolio und zeigen mit angepassten Effektivitäts-Tests, dass damit hochqualitative Lösungen auf einer diversen Menge von Benchmark-Instanzen erreicht werden.

Acknowledgments

I want to thank my supervisor Tobias Heuer for our intensive and productive collaboration. We work together on graph- and hypergraph partitioning since my bachelor thesis and I learned a lot from him. While we had different opinions on some occasions, the resulting discussion always lead to an improvement or new insight. Clearly, this thesis would not have been possible without such a mentor, who is as important for staying motivated as he is for further improving the content of the work. Also, I would like to thank Prof. Peter Sanders for the directions and new ideas he brought into our meetings.

Finally, I want to thank my family and my friends for their love, patience and support. Although this is easy to forget, it is something to be grateful for.

Contents

1. Introduction 7
1.1. Problem Statement . . . 8
1.2. Contribution . . . 9
1.3. Outline . . . 10

2. Preliminaries 11
2.1. Graphs . . . 11
2.2. Graph Partitioning . . . 11
2.3. Hypergraphs . . . 11
2.4. Hypergraph Partitioning . . . 12
2.5. Knapsack Problems . . . 12
2.6. Weighted Bipartite Matching Problem . . . 13
2.7. Partition Problem . . . 13

3. Related Work 14
3.1. Multilevel Graph Partitioning . . . 14
3.2. The Mt-KaHyPar Partitioning Framework . . . 15
3.3. Knapsack Problems . . . 16
3.4. Weighted Bipartite Matching . . . 18

4. The Star Partitioning Optimization Problem 19
4.1. Problem Definition . . . 20
4.2. Star Partitioning with Degree One Nodes . . . 22
4.3. Star Partitioning with Arbitrary Degree . . . 26
4.4. Possible Approaches for Better Approximations . . . 29

5. Engineering a Multilevel Star Partitioner 35
5.1. Detection of Peripheral Nodes . . . 37
5.2. Coarsening . . . 39
5.3. Initial Partitioning . . . 43
5.4. An Efficient Data Structure for Separated Nodes . . . 46
5.5. Generalization to Hypergraphs . . . 47

6. Experimental Results 49
6.1. Setup and Methodology . . . 49
6.2. Parameter Tuning . . . 51
6.3. Evaluation of Final Configurations . . . 60
6.4. Evaluation of Algorithm Portfolios . . . 64

7. Conclusion 68
7.1. Future Work . . . 68

References 70

A. Visualization of a Star-like Graph 75

B. Detailed Composition of Benchmark Sets 78

1. Introduction

There are many application areas where graphs need to be partitioned such that the blocks are of roughly equal weight and the edges between different blocks are minimized. It is an important preprocessing step for parallel computations that can be modeled as a graph, which arise, e.g., in scientific computing [6, 69]. Here, the task is to compute a distribution of the work to processors such that the load is balanced and the communication overhead is minimized, which is accurately modeled by the objective function used for graph partitioning. Further applications include route planning where partitioning the road network into cells allows for significant speedups [20]. In VLSI design, partitioning the circuit into smaller clusters allows to reduce the complexity while minimizing the length of connecting wires [37, 42].

In this work, we consider the *balanced* graph partitioning problem where the weight of each block is bounded by $1 + \varepsilon$ times the average block weight. As graph partitioning is NP-hard [10], heuristic algorithms are used in practice. The most successful heuristic that is used by state-of-the-art partitioners is the *multilevel paradigm* [30, 60]. Here, a hierarchy of graphs with decreasing size is built by repeatedly finding a matching or clustering of highly-connected nodes and subsequently contracting them. When the graph is small enough, an initial partition is computed. Afterwards, the partition is refined while undoing the contractions, allowing for both coarse- and fine-grained improvements on the respective levels of the hierarchy.

In recent years, partitioning algorithms for solving the graph partitioning problem have turned into complex systems using a variety of different techniques such as flow-based refinement or evolutionary techniques [14]. In addition, parallel partitioning algorithms have been developed that produce partitions competitive with the highest-quality sequential algorithms using only a fraction of the time [2, 52]. Moreover, both the size of the benchmark sets on which these systems are evaluated as well as the size of the included instances has increased, allowing for more resilient experimental results [2, 30].

Although such large benchmarks sets are used for evaluating current partitioning algorithms, we found a specific set of instances where all existing multilevel algorithms produce solutions significantly worse compared to an astonishingly simple and unorthodox approach: To calculate a bipartition ($k = 2$), we sort the nodes by degree and place the first half in one block and the second half in the other block.[1]

On a benchmark set that consists of 172 graphs [26, 30], we found eight instances where the degree-based partitioning technique significantly outperforms all existing graph partitioning algorithms. Table 1 shows the edge cuts produced by the tested algorithms relative to the degree-based partitioning approach.[2] One of the identified instances is the Twitter graph with over one billion edges. Here, the degree-based partitioning approach achieves an improvement by a factor of 1.66 over the best multilevel algorithm. This is very surprising: It is not intuitive why sorting the nodes by degree should yield a good partition.

The explanation is in the very specific structure of the graphs (referred to as *star graphs*): Instances such as those derived from social networks exhibit a highly skewed distribution of node degrees, e.g. in many cases the node degrees adhere to a power law distribution. However, in addition to their node degree distribution, star graphs contain both a dense *core* of high degree nodes and a large number of *peripheral* nodes with low degree, with only few edges between peripheral nodes (see Figure 1 for a visualization).

[1] In addition, we perform one V-cycle [67] on the partition as a refinement step. This means that we apply a coarsening round where contractions are restricted to the computed partition and project the partition to the coarse graph. This then allows to use traditional multilevel refinement techniques during the uncontraction.

[2] We use parameters $k = 2$ and $\varepsilon = 0.03$ and report for each partitioner the best result out of 10 repetitions. For more details on the used configurations we refer to Ref. [30].

Graph	Nodes	Edges	Degree-Based	Mt-KaHyPar-D	Mt-KaHyPar-D-F	KaFFPa-StrongS	Metis-K	ParMetis	KaMinPar	Mt-KaHIP	ParHIP
wiki-Vote	7115	101K	5342	2.93	2.92	2.51	2.93	2.93	2.93	2.93	3.05
soc-Epinions1	76K	406K	31K	1.50	1.51	1.61	2.06	1.83	2.08	1.58	1.93
soc-Slashdot0811	77K	469K	43K	2.21	2.30	2.08	2.58	2.59	2.61	1.97	3.04
soc-Slashdot0902	82K	504K	46K	2.14	2.17	1.89	2.50	2.52	2.52	1.91	2.74
rmat_n16_m22	65K	4194K	660K	1.25	1.27	1.07	2.98	2.77	1.87	1.19	1.71
rmat_n16_m23	65K	8389K	1315K	1.17	1.15	0.97	3.04	2.77	1.59	1.12	1.48
rmat_n16_m24	66K	17M	2742K	1.09	1.09	0.98	2.97	2.72	1.52	1.05	1.40
kron_g500-logn20	1049K	45M	505K	5.80	5.78	-	31.10	19.97	6.11	4.48	9.14
twitter-2010	42M	1203M	94M	1.66	-	-	-	-	2.56	1.73	1.67

Table 1: Comparison of the degree-based partitioning technique to the state-of-the-art multilevel partitioners Mt-KaHyPar [30], KaFFPa [61], Metis [38], ParMetis [41], KaMinPar [26], Mt-KaHIP [2] and ParHIP [52]. For each instance, the table contains the number of nodes, the number of edges and the number of cut edges produced by the degree-based partitioning technique for $k = 2$. The remaining columns show the number of cut edges produced by the other evaluated algorithms relative to the degree-based cut.

As the importance of social network graphs in applications is increasing, developing solutions to efficiently handle such instances is of outmost interest. In this work, we thus consider the *star partitioning problem*, i.e., graph partitioning restricted to star graphs. This includes both a theoretical analysis of star instances as well as the construction of algorithms for star partitioning.

1.1. Problem Statement

In this thesis we explore possibilities to augment graph partitioning algorithms such that they achieve better results on star graphs. The first task is to investigate whether a theoretical model of such instances allows to develop approximation algorithms with better guarantees than for general graph partitioning.

Then, it should be analyzed why current state-of-the-art partitioning algorithms compute solutions far from optimal on star graphs. For this, the phases of the multilevel paradigm should be revisited and possibilities for adapting them to star graphs should be explored. The work should be integrated into the graph and hypergraph partitioning framework *Mt-KaHyPar* (**M**ulti-**t**hreaded **Ka**rlsruhe **Hy**pergraph **Pa**rtitioning). The goal is to achieve significantly improved quality on star instances without decreasing the quality on other instances more than necessary.

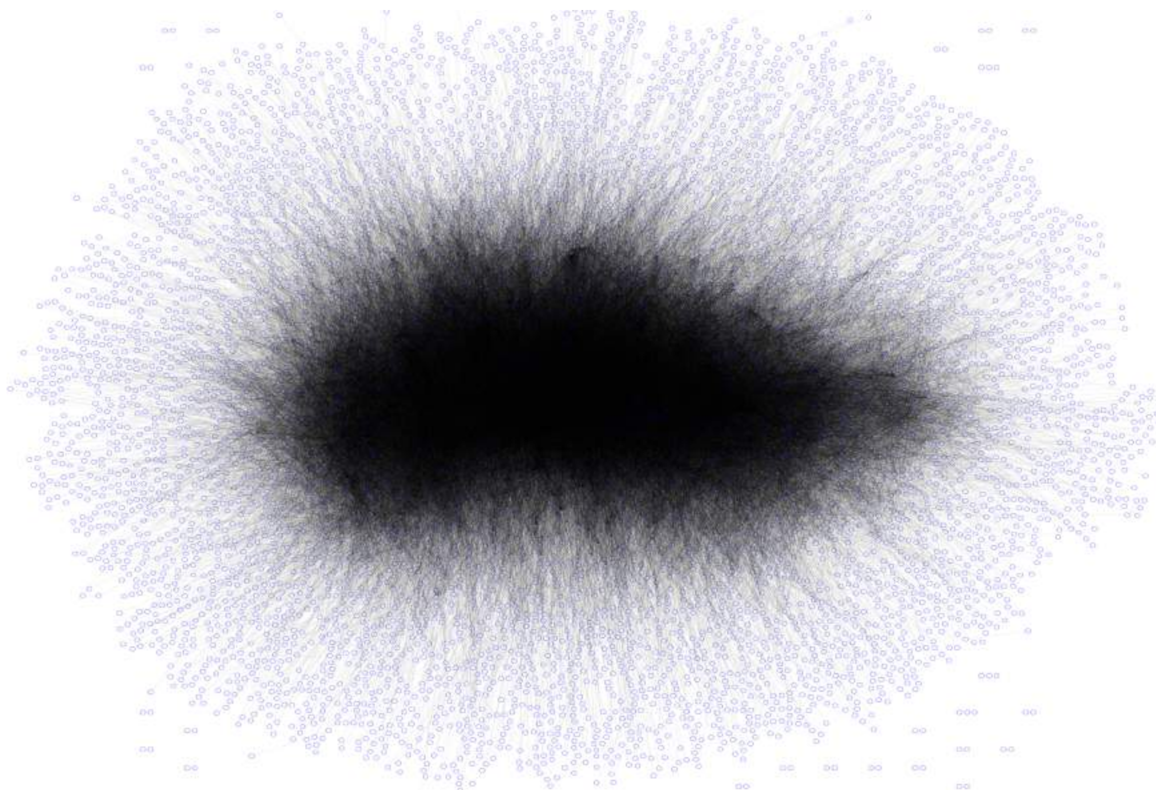


Figure 1: Visualization of the `wiki-Vote` instance.

1.2. Contribution

We develop a formal definition of star instances and provide a reduction of star partitioning to the *fixed core partitioning problem*. We show that several variants of this problem are NP-hard. However, our main contribution is an $(R+1)$-approximation for fixed core partitioning, where R is the approximation ratio of an algorithm for the min-knapsack problem.

One of the main shortcomings of existing multilevel algorithms is that peripheral nodes are contracted onto core nodes in the coarsening phase. Due to the increased node weight, this forces initial partitioning to separate the core in multiple blocks and thus induce a large cut. In our practical implementation, we therefore develop several strategies to detect peripheral nodes and then apply different coarsening strategies to core and peripheral nodes. For the latter, we develop specialized two-hop coarsening techniques where contraction partners are not required to be adjacent. Further, we show that on star-like instances the quality of the initial partitions can be significantly improved using our algorithm for fixed core partitioning.

To evaluate these techniques, we integrate our algorithms into the shared-memory (hyper-)graph partitioning framework Mt-KaHyPar. We compare our new star partitioning approach to Mt-KaHyPar on a large and diverse benchmark set. The results show that we can improve the quality on star-like instances significantly compared to the state-of-the-art. On 40% of the star instances, we achieve an improvement of more than a factor of two, and up to a factor of ten for some of the instances. Since these techniques provide worse results for some of the other instances, we additionally develop an algorithm portfolio and use effectiveness tests to show that the portfolio achieves high quality.

1.3. Outline

We introduce the required notation and basic definitions in Section 2. In Section 3 we give a summary of related work in the area. We analyze approximation algorithms for star instances within a theoretical framework that is developed in Section 4. Then, we describe different possible strategies for the detection of peripheral nodes and the integration into the multilevel paradigm in Section 5. We present the experimental evaluation in Section 6 and summarize the most important findings and directions for future work in Section 7.

2. Preliminaries

2.1. Graphs

Definition 2.1 (Graph)**.** *An undirected weighted graph* $G = (V, E, c, \omega)$ *consists of a set of nodes* V *with a weight function* $c\colon V \to \mathbb{R}_{\geq 0}$ *and a set of edges* E *with a weight function* $\omega\colon E \to \mathbb{R}_{>0}$. *Each edge* $e \in E$ *is a subset of the nodes with size* 2.

Given two nodes u and $v \in V$, we define $\omega(u, v) \coloneqq \omega(\{u, v\})$ if $\{u, v\} \in E$ and $\omega(u, v) \coloneqq 0$ if $\{u, v\} \notin E$. A node v is *incident* to an edge e if $v \in e$ and $V' \subseteq V$ is incident to e if $e \cap V' \neq \emptyset$. $I(v)$ denotes the set of all incident edges of v and $d(v) \coloneqq |I(v)|$ the *degree* of v. Two nodes u and v are *adjacent* if there is an edge $e = \{u, v\}$.

For subsets $V' \subseteq V$ and $E' \subseteq E$ we define

$$c(V') \coloneqq \sum_{v \in V'} c(v)$$
$$\omega(E') \coloneqq \sum_{e \in E'} \omega(e)$$

2.2. Graph Partitioning

Definition 2.2 (k-way partition)**.** *A* k*-way partition of a graph* G *is a partition of its nodes into* k *disjoint blocks* $\Pi = \{V_1, \dots, V_k\}$ *such that* $\bigcup_{i=1}^{k} V_i = V$ *and* $V_i \neq \emptyset$ *for* $i = 1, \dots, k$.

Note that a 2-way partition is also called a *bipartition*. A k-way partition $\Pi = \{V_1, \dots, V_k\}$ is ε*-balanced* if every block $V_i \in \Pi$ satisfies the *balance constraint* $c(V_i) \leq L_{max} \coloneqq (1 + \varepsilon) \left\lceil \frac{c(V)}{k} \right\rceil$ for some parameter ε. An edge e is *cut* if there are two different blocks V_i, V_j that are both incident to e. The set of all cut edges is denoted by $E(\Pi)$. We define the *cut* of a partition as $\mathrm{cut}(\Pi) \coloneqq \sum_{e \in E(\Pi)} \omega(e)$.

Definition 2.3 (Graph Partitioning Problem)**.** *The* k*-way graph partitioning problem is to find an* ε*-balanced* k*-way partition* Π *of a graph* G *with minimal cut.*

Sometimes, it is useful to consider partitions of a subset of nodes. We call a k-way partition $\Psi = \{P_1, \dots, P_k\}$ of a subset $P \subseteq V$ a prepacking. Further, for a given partition $\Pi = \{V_1, \dots, V_k\}$ we denote by $\Pi[P] \coloneqq \{V_1 \cap P, \dots, V_k \cap P\}$ the restriction of Π to P.

2.3. Hypergraphs

Hypergraphs are a generalization of graphs where an edge can consist of more than two nodes.

Definition 2.4 (Hypergraph)**.** *An undirected weighted hypergraph* $H = (V, E, c, \omega)$ *consists of a set of hypernodes* V *with a weight function* $c\colon V \to \mathbb{R}_{\geq 0}$ *and a set of hyperedges* E *with a weight function* $\omega\colon E \to \mathbb{R}_{>0}$. *Each hyperedge* $e \in E$ *is a non-empty subset of the hypernodes.*

A hypernode contained in a hyperedge is called a *pin* of the hyperedge. The *size* of a hyperedge e is its cardinality $|e|$. Incidence and adjacency are defined analogous to graphs. For subsets $V' \subseteq V$ and $E' \subseteq E$ we define

$$c(V') \coloneqq \sum_{v \in V'} c(v)$$
$$\omega(E') \coloneqq \sum_{e \in E'} \omega(e)$$

2.4. Hypergraph Partitioning

Definition 2.5 (k-way partition). *A k-way partition of a hypergraph H is a partition of its hypernode set into k disjoint blocks $\Pi = \{V_1, \ldots, V_k\}$ such that $\bigcup_{i=1}^{k} V_i = V$ and $V_i \neq \emptyset$ for $i = 1, \ldots, k$.*

The *connectivity set* of a hyperedge e is denoted by $\Lambda(e, \Pi) := \{V_i \in \Pi \mid e \cap V_i \neq \emptyset\}$ and the *connectivity* by $\lambda(e, \Pi) := |\Lambda(e, \Pi)|$. e is *cut* if $\lambda(e, \Pi) > 1$. The set of all cut hyperedges is denoted by $E(\Pi) := \{e \in E \mid \lambda(e, \Pi) > 1\}$.

Using this, the *hypergraph partitioning problem* is defined analogous to the graph partitioning problem. However, different objective functions are used in hypergraph partitioning. The *cut-net* metric is defined as

$$\text{cut}(\Pi) := \sum_{e \in E(\Pi)} \omega(e)$$

Another important objective function is the (λ - 1)-metric or *connectivity* metric:

$$(\lambda - 1)(\Pi) := \sum_{e \in E} (\lambda(e, \Pi) - 1)\omega(e)$$

2.5. Knapsack Problems

Many different types of knapsack problems can be found in literature. Classical variants consist of a set J of elements together with a weight function $c\colon J \to \mathbb{R}_{\geq 0}$, a gain function $\omega\colon J \to \mathbb{R}_{\geq 0}$. For a subset $J' \subseteq J$ we denote the weight of J' by $c(J') := \sum_{u \in J'} c(u)$ and the gain of J' by $\omega(J') := \sum_{u \in J'} \omega(u)$.

Definition 2.6 (0-1 Knapsack Problem). *For a set of elements J and a given maximum weight c_{max}, the 0-1 (max-)knapsack problem is to find a subset $S \subseteq J$ such that $c(S) \leq c_{max}$ and the gain $\omega(S)$ is maximized.*

Definition 2.7 (0-1 Min-Knapsack Problem). *For a set of elements J and a given minimum weight c_{min}, the 0-1 min-knapsack problem is to find a subset $S \subseteq J$ such that $c(S) \geq c_{min}$ and the gain $\omega(S)$ is minimized.*

The max and the min variant of the knapsack problem are closely related: For a given instance of the max-knapsack problem, there is a complementary instance of the min-knapsack problem with $c_{min} := c(J) - c_{max}$. If $S \subseteq J$ is a solution of the former, then $J \setminus S$ is a solution of the latter. Informally, the max-knapsack problem maximizes the gain of elements placed within a bin, while the complementary min-knapsack problem minimizes the gain of the elements outside of the bin. Clearly, for both variants the decision problem, which is known to be NP-complete, is equivalent. However, the effectiveness of approximation algorithms is generally different for both variants.

For a given algorithm $\mathcal{A}$ and instance J, we denote by $\mathcal{A}(J)$ the value of the solution calculated by $\mathcal{A}$ for this instance and by $OPT(J)$ the optimal value. Then we can define the approximation ratio of $\mathcal{A}$ as

$$R_{max} := \sup_J \left\{ \frac{OPT(J)}{\mathcal{A}(J)} \right\} \qquad (1) \qquad\qquad R_{min} := \sup_J \left\{ \frac{\mathcal{A}(J)}{OPT(J)} \right\} \qquad (2)$$

where the ratio is given by (1) for the maximization variant and by (2) for the minimization variant.

2.6. Weighted Bipartite Matching Problem

A matching in a graph G is a subset of edges $M \subseteq E$ such that each node is incident to at most one edge in M. We say that G is a *bipartite* graph with parts T and U if $\{T, U\}$ is a bipartition of G such that every edge $e \in E$ is incident to both T and U. G is a *complete* bipartite graph if every node in T is adjacent to every node in U. A matching M in G is called *perfect* if $|M| = |T| = |U|$. The *weighted bipartite matching problem* is about finding a matching in a bipartite graph with (depending on the variant) minimal or maximal weight.

Definition 2.8 (Minimum Weighted Bipartite Matching)**.** *Given a bipartite graph G with parts T, U of equal size and edge weights ω, the minimum weighted bipartite matching problem is to find a perfect matching M in G that minimizes $\omega(M)$.*

Definition 2.9 (Maximum Weighted Bipartite Matching)**.** *Given a bipartite graph G with edge weights ω, the maximum weighted bipartite matching problem is to find a matching M in G that maximizes $\omega(M)$.*

Note that in literature, the weighted bipartite matching problem is also called the assignment problem.

2.7. Partition Problem

Definition 2.10 (Partition Problem)**.** *Given a set of natural numbers S with total sum $t := \sum_{a \in S} a$, the partition problem is to find a subset $T \subseteq S$ such that $\sum_{a \in T} a = \frac{t}{2}$.*

The partition problem is NP-complete. As it is very simple to describe, it is useful for proving that new problems are NP-hard. Note that there is also an optimization variant where the objective function is to minimize the difference of the total sums of T and $S \setminus T$. Unsurprisingly, there are efficient heuristics as well as pseudo-polynomial-time algorithms for the partition problem [45].

3. Related Work

Graph partitioning is NP-hard [23] and it is even NP-hard to find constant factor approximations [10]. Therefore, heuristics are used in practice for large instances. We give a summary of the most successful approaches in this section. The results of this thesis are implemented within the graph and hypergraph partitioning framework *Mt-KaHyPar*. For this reason, we provide an overview of the implementation of Mt-KaHyPar in Section 3.2. In addition, we summarize existing literature on the knapsack and weighted bipartite matching problems since we use these problems in this thesis. Note that some of the following descriptions are copied verbatim from the author's bachelor thesis [50].

3.1. Multilevel Graph Partitioning

The most successful approach for solving the graph partitioning problem is the *multilevel paradigm* [30, 61, 62]. There are two main variants of the paradigm: In *direct k-way* partitioning, the algorithm has three phases (see Figure 2). First, the graph is *coarsened*, i.e. a matching or clustering algorithm is used to compute node sets that are contracted. This process is repeated until the graph has a predefined size, while still reflecting the basic structure of the input graph. The coarsening thereby constructs a hierarchy of multiple levels corresponding to graphs with decreasing size. Then, an *initial partitioning* algorithm calculates a k-way partition of the coarsest graph. Due to the smaller size, the use of expensive algorithms for the partitioning is possible. During the *refinement* phase, the partition is projected back to the original graph in reverse order of the contractions. At every level, refinement heuristics are used to further improve the quality of the solution. The most common technique is the Fiduccia-Mattheyses (FM) algorithm [19] which is based on local search.

In the *recursive bipartitioning* variant of the paradigm, the algorithm calculates a bipartition of the original graph within the same three phases. Then, the algorithm is recursively applied to each of the two blocks to obtain the final k-way partition.

Main advantages of the multilevel paradigm are that it allows for expensive initial partitioning algorithms that operate globally while preserving near-linear computation time through the coarsening of the graph, and that the refinement algorithms can escape local minima at the coarser levels while also improving the solution in detail at the finer levels.

Coarsening Algorithms. We provide more details on existing work regarding the coarsening phase, which is of particular importance for this thesis (see also Ref. [30]). Early multilevel algorithms primarily used matching-based coarsening schemes [40], contracting pairs of nodes in every coarsening step. These algorithms work well for graphs with a regular structure, e.g., finite element meshes. However, graphs with a power-law node degree distribution are difficult to coarsen with matching-based approaches: Since many low-degree nodes are adjacent to few high-degree nodes, matchings tend to be small [1]. Therefore, many recent partitioning algorithms use clustering-based coarsening schemes instead [30], grouping sets of densely connected nodes and subsequently contracting them into a single node. Here, high-degree nodes are likely to become heavy nodes in the coarsest graph, which can make it difficult to find a feasible initial partition [53]. Thus, it is common to enforce a constraint on the weight of the heaviest cluster [53, 63] or penalize the contraction of heavy nodes [12, 60].

As an alternative to clustering-based methods, LaSalle et al. proposed to complement traditional matching algorithms with a second *two-hop* matching pass [47]. Here, the constraint that matched vertices must be connected via an edge is relaxed. Instead, vertices can be matched

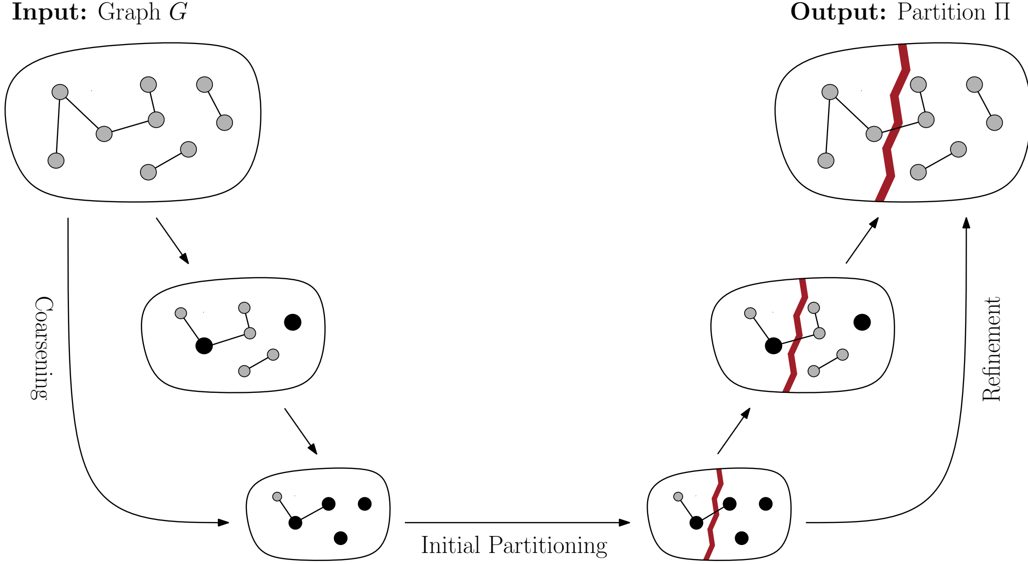


Figure 2: Graph partitioning with the multilevel paradigm

if they have a common neighbor. The authors propose to first aggregate degree one nodes and in a second step nodes with identical neighbor lists. Only if the resulting matching is not sufficiently large, any nodes with a common neighbor are matched [47].

3.2. The Mt-KaHyPar Partitioning Framework

Mt-KaHyPar is a scalable multilevel graph and hypergraph partitioner that supports multiple configurations with different trade-offs between quality and running time [27, 28, 30]: Traditional multilevel partitioning as well as a fine grained n-level approach that removes only one node for every level of the coarsening hierarchy. Further, there are configurations that are specialized to provide better running time for graph partitioning than the more generalized hypergraph partitioning algorithm. Mt-KaHyPar is capable of finding high quality solutions with better running time than most competitors and high scalability [30]. Our work is integrated into the direct k-way mode of the multilevel graph partitioner, which we thus describe in more detail. Note that the following algorithms are implemented in a highly scalable way, for details on this we refer to [27, 30].

Mt-KaHyPar adds an initial *preprocessing* phase to the three phases of the multilevel paradigm. Here, a community detection algorithm is applied [8, 27, 32]. This allows to use information about the global structure in the coarsening phase by restricting contractions to the detected communities. Contraction pairs are chosen using the *heavy-edge* rating function $r(u, v) := \sum_{e \in I(u) \cap I(v)} \frac{\omega(e)}{|e|-1}$. The coarsening algorithm works in passes, choosing per pass a contraction partner for each node according to the heavy-edge rating function. The passes are repeated until only $t = 160k$ nodes remain. To avoid imbalanced inputs to the initial partitioning phase, vertices that reach a certain weight threshold are not contracted further [27].

Even in direct k-way mode, the initial partitioning phase uses a recursive bipartitioning algorithm internally, as this produced better results compared to flat approaches [27, 63]. In every recursive step, a bipartition of the current subgraph is calculated in the following way: The graph is coarsened further until it reaches a size of 320 nodes. Then, a bipartition is calculated using a portfolio of different partitioning algorithms and selecting a balanced result with the best objective function [27]. The resulting partition is uncoarsened using a label propagation algorithm and a 2-way FM local search algorithm.

To ensure that the resulting partition is balanced Mt-KaHyPar uses an *adaptive imbalance parameter* for the bipartitioning steps which is defined as follows: Let G_{V_i} be the current subgraph for which a k'-way partition should be calculated. Then

$$\varepsilon' := \left((1+\varepsilon)\frac{c(V)}{k} \cdot \frac{k'}{c(V_i)}\right)^{\frac{1}{\lceil \log_2(k') \rceil}} - 1 \tag{3}$$

is used as an imbalance parameter for the bipartition of V_i [30].

In the refinement phase, the resulting k-way partition is uncoarsened while optimizing the objective function via a label propagation algorithm and a k-way local search. The FM local search algorithm maintains a thread-local *priority queue* for each block of the partition according to the *gain* of a node and updates the gains of all nodes adjacent to a move with a *delta-gain* update strategy. To improve the running time a sophisticated *gain cache* is used that prevents expensive recalculations and an *adaptive stopping rule* which terminates the local search when a further improvement becomes unlikely [3, 27]. In addition, Mt-KaHyPar allows to add a flow-based refinement algorithm which is based on the *FlowCutter* algorithm [25, 29]. It works on block pairs and can improve them in a more global manner than local search algorithms [30].

3.3. Knapsack Problems

A large number of variants of the knapsack problem can be found in literature, ranging from the basic 0-1 knapsack problem to variants with constraints on the choice of elements [64] or multiple dimensions that can be used for assigning elements [4, 58]. However, in this thesis we only directly use the basic maximization and minimization variants.

Max-Knapsack. Having many practical applications [68], the 0-1 max-knapsack problem is well researched. While it is known to be NP-complete, there are multiple ways to find efficient approximations. Dantzig proposed a simple greedy algorithm [16]: It sorts the elements in decreasing order of their gain per weight ratio, ignoring any element that already exceeds the maximum weight. Then, two solutions S_1, S_2 are considered, returning the better one: S_1 is constructed by greedily packing the first elements until no new element fits, S_2 contains only the first element that is not part of S_1. It can be shown that this achieves an approximation ratio of 2.

Furthermore, the 0-1 knapsack problem can be solved in pseudo-polynomial time using dynamic programming [56]. For an arbitrary order of the elements, let $f(i, c)$ be the maximum possible gain using only the first i elements with weight at most c. Let u be the element at position $i + 1$. Then, we can use the relation $f(i + 1, c) = \max\{f(i, c), f(i, c - c(u)) + \omega(u)\}$ to build a tabular that results in the optimal solution, requiring a running time of $\mathcal{O}(nc_{max})$ where c_{max} is the maximum allowed weight. A very similar approach can be applied to build a tabular based on the gain instead of the weight of the elements (a tabular entry then contains the minimum possible weight for the given gain). This gives a running time of $\mathcal{O}(nP)$ where P is the maximum possible gain.

The dynamic programming approach outlined previously can also be used to construct an FPTAS [56]. The idea is to round all involved gains to an integer value bound by a polynomial (more precisely, each gain is divided by $\varepsilon\frac{\omega_{max}}{n}$, where $1 + \varepsilon$ is the approximation factor and ω_{max} the maximum gain of an element). The transformed instance can be solved in polynomial time with the described dynamic programming algorithm. It can be shown that this results in the stated approximation factor with a running time of $\mathcal{O}(\frac{1}{\varepsilon}n^3)$.

It should be noted that the approaches based on dynamic programming are not always practical due to the high memory requirements. A possible alternative is to use branch-and-bound algorithms. Although they often have an exponential worst-case running time, they can be faster in practice on many instances. An example of a branch-and-bound algorithm that works well in practice is proposed by Horowitz and Sahni [34]. Specifically, it selects the next branch greedily in order to explore the most promising solution candidates first. A general survey of knapsack problems that includes multiple generalized variants and discusses different solution approaches is given by Wilbaut, Hanafi and Salhi [68].

Min-Knapsack. Although the min-knapsack problem is closely related to the classical max-knapsack problem, it is curiously underrepresented in literature: To the best of our knowledge, there are only two existing publications that are focused specifically on this subject, Ref. [15] and Ref. [35]. However, it seems that the min-knapsack problems can be solved with techniques that are generally similar to the ones used for the maximization variant. Csirik et al. present an approximation algorithm called *GR* with approximation factor 2 that runs in linear time after sorting the elements [15]. As we will use this algorithm later on, we provide a more detailed description in the following: Consider a sequence $J = \{e_1, \dots, e_n\}$ of the elements sorted in increasing order of the gain per weight ratio $\frac{\omega(e_i)}{c(e_i)}$ and a given minimal weight c_{min}. We build a set $M \subseteq J$ with $c(M) < c_{min}$ by iterating over all elements, adding the current element if this does not exceed c_{min}. Let $N \coloneqq J \setminus M$. For each $e_i \in N$, we consider the candidate solution $S_i \coloneqq (M \cap \{e_1, \dots, e_i\}) \cup \{e_i\}$ (note that $c(S_i) \geq c_{min}$ by construction of M) and choose the solution with minimal gain. Clearly, if the elements are already sorted the algorithm can be implemented in linear time by tracking the current gain of S and storing only the index of the current minimal solution in addition to S.

While the authors give a proof for the approximation factor [15], we want to remark that the proof is rather hard to follow. We therefore provide a simplified version in the following.

Lemma 3.1. *The GR algorithm achieves an approximation ratio of* 2.

Proof. Consider an optimal solution $L \subseteq J$. Let t be the smallest index with $e_t \in L \cap N$ (exists as $c(M) < c_{min}$ implies $L \nsubseteq M$) and let $S_t = (M \cap \{e_1, \dots, e_t\}) \cup \{e_t\}$ be the according candidate solution. We define $\overline{L} \coloneqq L \setminus \{e_t\}$ and $\overline{S_t} \coloneqq S_t \setminus \{e_t\}$. Because t is chosen minimal, for each element $e_i \in \overline{L}$ it holds that either $i > t$ or $e_i \in \overline{S_t}$. Therefore, we can use the order of the elements to retrieve the inequalities

$$\frac{\omega(\overline{S_t} \setminus \overline{L})}{c(\overline{S_t} \setminus \overline{L})} \leq \frac{\omega(\overline{L} \setminus \overline{S_t})}{c(\overline{L} \setminus \overline{S_t})} \quad (4) \qquad \frac{\omega(\overline{S_t} \setminus \overline{L})}{c(\overline{S_t} \setminus \overline{L})} \leq \frac{\omega(e_t)}{c(e_t)} \quad (5)$$

Further, $c(\overline{S_t}) < c_{min} \leq c(L) = c(\overline{L}) + c(e_t)$. By subtracting $c(\overline{S_t} \cap \overline{L})$ this implies $c(\overline{S_t} \setminus \overline{L}) \leq c(\overline{L} \setminus \overline{S_t}) + c(e_t)$. We can conclude

$$\begin{aligned} \omega(\overline{S_t}) &= \omega(\overline{S_t} \cap \overline{L}) + \omega(\overline{S_t} \setminus \overline{L}) \leq \omega(\overline{S_t} \cap \overline{L}) + \frac{\omega(\overline{S_t} \setminus \overline{L})}{c(\overline{S_t} \setminus \overline{L})} \left(c(\overline{L} \setminus \overline{S_t}) + c(e_t) \right) \\ &\leq^{(4),(5)} \omega(\overline{S_t} \cap \overline{L}) + \omega(\overline{L} \setminus \overline{S_t}) + \omega(e_t) = \omega(L) \end{aligned}$$

and thus $\omega(S_t) = \omega(\overline{S_t}) + \omega(e_t) \leq \omega(L) + \omega(e_t) \leq 2\omega(L)$. □

In addition, the authors construct another approximation algorithm GR* with running time $\mathcal{O}(n^2)$ and approximation ratio 3/2 that is based on the GR algorithm [15]. GR* works by constructing a new knapsack (sub-)instance for each $e_i \in N$ and applying GR to it, adding e_i

to the calculated solution. Effectively, this means that e_i is a fixed part of that solution. Again, the final result is the best of all considered solutions.

In his master thesis, Islam showed that an FPTAS for min-knapsack can be constructed using a very similar approach to the FPTAS for max-knapsack [35]. However, in this case it is necessary to calculate a separate solution for n subproblems, resulting in a total running time of $\mathcal{O}(\frac{1}{\varepsilon}n^4)$ for an approximation ratio of $1+\varepsilon$. Further, he showed that dynamic programming is also applicable to a multi-dimensional variant of the problem.

3.4. Weighted Bipartite Matching

We have defined the weighted bipartite matching problem as finding a minimum weight perfect matching in a bipartite graph G (see Section 2.6). However, it was originally formulated in terms of a quadratic cost matrix, where an entry a_{ij} represents the costs of assigning row i to column j. This is mostly equivalent, except that the graph formulation allows for missing edges. The probably best-known algorithm for finding a weighted bipartite matching is the so-called *Hungarian method*, which was first published by Kuhn [46], using the aforementioned matrix formulation.

The Hungarian method assigns to each node u a *potential* $p(u)$ with the property that $p(u) + p(v) \leq \omega(\{u, v\})$ for every edge $\{u, v\}$. Let us call an edge with $p(u) + p(v) = \omega(\{u, v\})$ *tight*. The maximum possible value of a potential p is equal to the minimum weight of a perfect matching (and induces such a matching as a subset of all tight edges). On a high level, the Hungarian method works by updating the potential until this equality is achieved.

In terms of linear programming, every perfect matching is a *primal* solution that provides an upper bound for the minimum weight, while the potential is a *dual* solution that provides a lower bound. The Hungarian method is therefore a *primal-dual* method.

An outline of the calculations that are actually required looks as follows: We first need an initial value for p which is achieved by using the minimum incident edge weight for each node (subtracting the potential of the second incident node, if it is already assigned). In each step, we consider the subgraph G_p that contains only tight edges. To correctly update the potential, we calculate a minimum vertex cover C of G_p, which can be derived from a maximum cardinality matching in G_p. The potential is updated by the value $\delta \coloneqq \min_{\{u,v\} \text{ not incident to } C} \omega(\{u, v\})$. In the first part of G, we add δ to the potential of all nodes that are not in C, in the second part of G we subtract δ from all nodes that are contained in C. Due to the definition of δ, this will increase the size of G_p (at least one edge that is not covered by C is added).

It is possible to implement the Hungarian method with a running time of $\mathcal{O}(n^3)$ by maintaining a tree that consists of tight and non-tight edges in alternating order, allowing to efficiently update the matching via augmenting paths. We refer to [11] for a more detailed explanation.

An alternative, but similar approach to finding a minimum weighted bipartite matching is using shortest augmenting paths [11]. Here, the same node potential p is used, but the update for each step is calculated via a shortest path algorithm. Using e.g. Dijkstra with Fibonacci heaps, this approach also achieves a running time of $\mathcal{O}(n^3)$.

Note that as currently described, these approaches can only solve the minimization variant of the weighted bipartite matching problem where both parts of G need to be of equal size. However, the maximization variant can be reduced to the minimization variant by first adding dummy vertices to the smaller part with incident edges of weight zero and then reversing the edge weights, i.e. using $\omega'(e) \coloneqq \omega_{max} - \omega(e)$ where ω_{max} is the maximum edge weight in G.

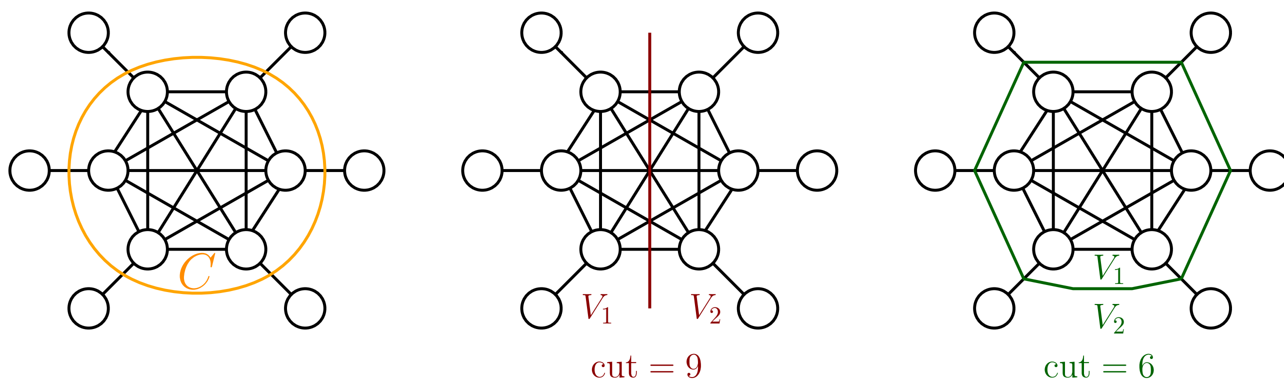


Figure 3: Example of splitting the core of a graph. The core C is a clique of size 6. Dividing the core in two halves results in a cut with value 9, while separating the peripheral nodes from the core yields a better cut with value 6.

4. The Star Partitioning Optimization Problem

Finding a constant factor approximation for the graph partitioning problem is an NP-hard problem [10]. Since real world instances are too large for exact solvers, we need to use heuristic approaches in practice. Specifically, this means that for most instances, we do not know how large the difference between an existing solution and the optimal solution is. To find possibilities for improvement for a given algorithm, we thus need to rely on experimental evaluation, e.g., by comparison to other algorithms. However, it can also happen that for a specific instance, a better solution is discovered through alternative means.

This is the case for a set of instances which we describe in the experimental evaluation – which is the primary motivation for this thesis. On these instances, we found that an astonishingly simple and unorthhodox approach can find substantially better partitions than the currently best partitioning algorithms. For example, we can find a bipartition on the Twitter graph – a graph with over 1 billion edges - that cuts only half as many edges as any multilevel algorithm. This works as follows: We sort all nodes by degree. Then, we assign the half of the nodes that has higher degree to the first block and the remaining nodes to the second block. Clearly, this behavior is very surprising, as this approach does not optimize the objective function in the general case. Thus, there needs to be a very specific structure which causes the observed behavior.

The Twitter graph is a social network with a highly skewed node degree distribution, similar to the degree distribution of other instances with the same behavior. Also, we visualized some of the smaller instances (see Appendix A). The visualization leads us to the conclusion that the single most distinguishing feature of these instances is the following: We can group the nodes into two categories, which both include a significant portion of the total number of nodes: a *dense core* that consists of highly interconnected nodes with relatively high node degrees and *peripheral nodes* with low node degrees that are mostly connected to the nodes in the core. Bipartitioning the nodes based on their degree then has the effect that the core is placed in one block, while (most of) the peripheral nodes are in the second block. Potentially, this results in a much smaller cut than dividing the core itself, which is done by most multilevel algorithms (see next paragraph). We can demonstrate this with a small example, as depicted in Figure 3: Let us consider a graph which consists of a clique C with size s – the core – and s peripheral nodes with degree one, such that each peripheral node is adjacent to exactly one node in C. Then, the partition $\{C, V \setminus C\}$ has cut s. On the other hand, a partition where both blocks contain half of C has cut $(\frac{s}{2})^2 = \frac{1}{4}s^2$. While this example is rather idealized, it demonstrates

that separating the core and the peripheral nodes might be a good idea to achieve a small cut – which also holds for larger k, although to a somewhat lesser degree.

Problems with Multilevel Partitioning. The root cause for the poor results of multilevel algorithms for these instances can be found in the coarsening phase: Commonly used coarsening algorithms use heuristics that select adjacent nodes as contraction partners [13, 27, 61]. Because peripheral nodes are typically only adjacent to core nodes, it is very likely that most of the peripheral nodes are contracted onto the core. As there are more peripheral nodes than core nodes in graphs with highly skewed degree distribution, this increases the weight of the core substantially until the balance constraint forces the partitioning algorithm to cut the core. Considering this, it is not surprising anymore that sorting the nodes by degree leads to better partitioning results for these instances.

Note that these properties are of *global* nature – it is not possible to decide whether separating core and peripheral nodes is beneficial for an analysis that considers only the local neighborhood of a node. This is because it depends on the density of the core as a whole whether it makes sense to partition the core or not. Consequently, we will see that it is challenging to construct a partitioner that efficiently exploits such structure in practice. We present several heuristics for detecting this structure in Section 5.1.

To improve on the current state, the obvious first step is to avoid this pitfall, i.e., peripheral nodes should not be contracted onto the core. In contrary, contractions within the core itself are unproblematic, as they do not destroy the global structure of the graph. If we apply this idea directly to the coarsening phase by excluding peripheral nodes from contractions, it results in a graph where only the core is contracted, i.e., there are very few nodes with a large node weight and high degree in the core as well as a large number of peripheral nodes that have small weight and low degree. Although this seems counterintuitive to the idea of multilevel graph partitioning which assumes that the coarsened graph is of small size, it is actually a useful model that allows to tackle the problem from a more theoretical point of view. In the remainder of this section, we will formally define such instances and present several theoretical results on solving such instances optimally. These results lay some of the groundwork for designing practically applicable heuristics that can be integrated into the multilevel paradigm, which we do in Section 5.

4.1. Problem Definition

In the following, we define the *star partitioning* problem as a special case of the graph partitioning problem. In essence, it describes a constraint on the structure of the considered instances. Note that the following two definitions also apply to weighted graphs, but for simplicity we focus on unweighted graphs.

Definition 4.1 (Peripheral and Core Nodes). *Let $G = (V, E)$ be an undirected graph. We call a subset of nodes $P \subseteq V$ peripheral if the subgraph induced by P contains no edges (i.e., if it is an independent set in G). We call $C := V \setminus P$ the core of G for a given set of peripheral nodes P.*

The instances we want to consider are defined by a relatively large set of peripheral nodes, i.e., there is a large subset of nodes without any internal edges. Formally, we require the core of the graph to have a much smaller size than the total number of nodes.

Definition 4.2 (α-Star Graph). *We call a graph $G = (V, E)$ with peripheral nodes P an α-star graph if $|C| \leq \alpha$ for the core C.*

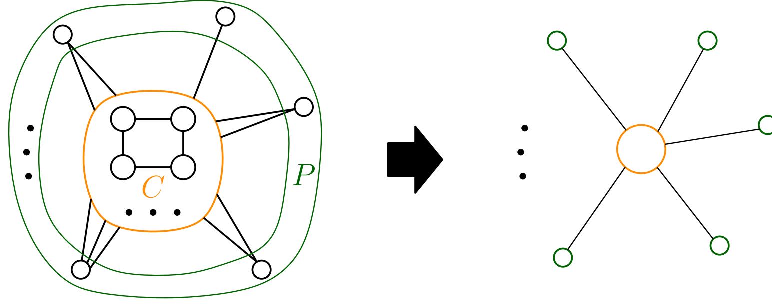


Figure 4: Illustration of the structure of a star graph. The left side depicts the original instance with the peripheral nodes P in green and the core C in orange. The right side depicts the result of contracting the core, resulting in a star.

Because we assume that α is small, we additionally use the term star graph to denote an α-star graph with small α. Intuitively, we can visualize a star graph as a dense core of highly connected nodes that is surrounded by peripheral nodes, as depicted in Figure 4 (note that in general the core could contain multiple components). From a high level perspective, the topology of the graph is star-like. More precisely, if we contract the core into a single node, the resulting graph is actually a star – hence the name star partitioning.

Recall, the graph partitioning problem is to find an ε-balanced k-way partition of G with minimal cut (see Definition 2.2). Star partitioning simply means solving the graph partitioning problem on star graphs. As we will see, for star graphs we can construct algorithms that have strong approximation guarantees with polynomial running time – which is not possible in the general case [10].

Fixed Parameter Tractability. To make use of the properties of an α-star graph, we formally allow the considered algorithms to have a running time that is fixed-parameter tractable with regards to α, i.e., a running time in $\mathcal{O}(f(\alpha)(|V| + |E|)^{\mathcal{O}(1)})$ for a computable function f, and we require that the number of blocks k is at most α. With this, we can use a simple general strategy: We consider every possible k-way partition of C in a brute-force way. As there are $\alpha^k \in \mathcal{O}(\alpha^\alpha)$ possibilities, this yields an FPT algorithm if we have an efficient procedure for assigning the peripheral nodes. For such a partition $C_1, \dots, C_k$, we then use a specialized algorithm to partition P into $P_1, \dots, P_k$, yielding a partition $\Pi = \{C_1 \cup P_1, \dots, C_k \cup P_k\}$ of G. Out of all considered partitions, we choose the partition Π with minimal cut as result. Therefore, we can reduce the problem to finding a partition of P where the partition of C is already fixed (we present a proof after the formal definition). This strategy of testing all possibilities is applicable for all following results, thus we define it once so that we do not need to repeat it for every algorithm.

Definition 4.3 (Fixed Core Partitioning, preliminary)**.** *Given a set of peripheral nodes P of a weighted graph $G := (V, E, c, \omega)$ and a k-way partition $\Pi_{\mathcal{C}} := \{C_1, \dots, C_k\}$ of its core $C = V \setminus P$. The fixed core partitioning problem is to find a k-way partition $\Pi_P := \{P_1, \dots, P_k\}$ of P such that $c(C_j \cup P_j) \leq L_{\max}$ for all $1 \leq j \leq k$ and $\delta(\Pi_P) := \sum_{j=1}^{k} \omega(P \setminus P_j, C_j)$ is minimized.*

The definition of the fixed core partitioning problem reveals that some of the structure of G is actually irrelevant for solving the problem. First, the total cut is defined by the incident edge weights between the blocks of Π_C and the peripheral nodes – for a block C_j only the total incident edge weight is important and not the number of edges. This means that for a given node $u \in V$, we are only interested in $\omega(u, C_j)$ for $j \leq k$. Second, since Π_C is fixed, edges within the core C are not relevant for the objective function of the fixed core partitioning

problem. Thus, we can remove them from the graph. Third, the remaining weight available in each block is determined by Π_C and ε, i.e., $c(P_j) \leq L_{max} - c(C_j)$. We can further simplify and simultaneously generalize the definition by directly stating the allowed maximum block weights instead of using the core nodes and ε. Taking all of this together, we can provide an adjusted definition containing only the information that is actually relevant. Note that we use similar notation to the graph partitioning formulation to indicate that these concepts are directly derived from the previous definitions, albeit being formally different.

Definition 4.4 (k-way Fixed Core Partition)**.** *Given a set of elements P with associated weights $c\colon P \to \mathbb{R}_{\geq 0}$, a k-way fixed core partition is a partition $\Pi_P = \{P_1, \ldots, P_k\}$ of P such that $c(P_j) \coloneqq \sum_{u \in P_j} c(u) \leq c_j$ for maximum allowed block weights $c_1, \ldots, c_k \in \mathbb{N}$ and $j \leq k$.*

Definition 4.5 (Fixed Core Partitioning)**.** *Given k blocks with maximum allowed weights $c_1, \ldots, c_k$ and a set of elements P with associated weight functions $c\colon P \to \mathbb{R}_{\geq 0}$ and $\omega\colon P \times [k] \to \mathbb{R}_{\geq 0}$, the fixed core partitioning problem is to find a k-way fixed core partition $\Pi_P = \{P_1, \ldots, P_k\}$ that minimizes $\delta(\Pi_P) \coloneqq \sum_{j=1}^{k} \omega(P \setminus P_j, j)$ (with $\omega(U, j) \coloneqq \sum_{u \in U} \omega(u, j)$ for $U \subseteq P$).*

With regards to terminology, we refer to the elements of P as nodes with a node and edge weight function c and ω (even though it is not a graph). We say that $u \in P$ has degree d if $\omega(u, j) \neq 0$ for d different blocks $j \in \{1, \ldots, k\}$.

Note that for the decision problem, it is equivalent to maximize the incident weights per block $\sum_{j=1}^{k} \omega(P_j, j)$. However, the two variants behave quite differently with regards to the possible approximation guarantees. We focus on the minimization variant as this corresponds to the optimization objective of graph partitioning.

It is clear that a star graph G with a set of peripheral nodes P and a partition $\Pi_C = \{C_1, \ldots, C_k\}$ of its core can be transformed into a fixed core partitioning instance: The instance with the same nodes P and associated weight functions c and ω as already defined for the partitioning instance and block weights $c_j \coloneqq L_{max} - c(C_j)$. A solution $\Pi_P = \{P_1, \ldots, P_k\}$ for this instance corresponds to a partition $\Pi = \{C_1 \cup P_1, \ldots, C_k \cup P_k\}$ of G and vice versa. Further, $\omega(\Pi) = \omega(\Pi_C) + \delta(\Pi_P)$. This is useful as the simplified structure of this problem variant also simplifies its analysis.

If we have an approximation algorithm $\mathcal{A}$ for fixed core partitioning with approximation ratio R and running time $g(|P|, k)$, we can use the approach outlined above to construct an approximation algorithm $\mathcal{A}'$ for star partitioning with approximation ratio R and running time $\mathcal{O}(\alpha^k g(|P|, k))$. We just solve the fixed core partitioning problem for every possible partition of C. To see that the resulting approximation ratio is R, we consider an optimal solution Π_{OPT}. Let $\Pi_C \coloneqq \Pi_{OPT}[C]$ be the according partition of C and let $\Pi_{\mathcal{A}'}$ be the partition of G that is calculated by $\mathcal{A}'$ using the fixed core partitioning instance derived from Π_C. Then

$$\omega(\Pi_{\mathcal{A}'}) = \omega(\Pi_C) + \delta(\Pi_{\mathcal{A}'}[P]) \leq \omega(\Pi_C) + R \cdot \delta(\Pi_{OPT}[P]) \leq R \cdot \omega(\Pi_{OPT})$$

Having established this, in the following we will use star partitioning and fixed core partitioning mostly interchangeable. Specifically, we implicitly describe approximation algorithms for star partitioning by describing the fixed core partitioning algorithm.

4.2. Star Partitioning with Degree One Nodes

As a first step, let us consider some special cases that are relatively straightforward to analyze. In the following, we assume that all peripheral nodes have degree one. Further, we say that an instance has *unit node weights* if all nodes in P have weight one. We use this terminology in the same way for star partitioning and fixed core partitioning.

Lemma 4.6 (Degree One and Unit Node Weights). *Given a fixed core partitioning instance with maximum block weights $c_1, \ldots, c_k$ and nodes P with weight functions $c\colon P \to \mathbb{R}_{\geq 0}$ and $\omega\colon P \times [k] \to \mathbb{R}_{\geq 0}$ where all nodes have weight one (i.e., $c \equiv 1$) and at most degree one, we can compute an optimal solution in $\mathcal{O}(|P| \log |P|)$ time.*

Proof. We first calculate for each block j the set K_j of nodes with non-zero edge weight to this block in linear time. Note that because the degree is bounded by one, each node is contained in at most one of the sets. Then we sort K_j with regards to ω and assign the c_j nodes with maximum edge weight to j. Remaining nodes are assigned to any block with free capacity (assuming $c(P) \leq \sum_{j=1}^{k} c_j$, otherwise the instance is infeasible).

Let $\Pi \coloneqq \{P_1, \ldots, P_k\}$ be the computed solution and $\Pi' = \{P'_1, \ldots, P'_k\}$ any other solution. Because we sorted by edge weight $\omega(P_j, j) \geq \omega(P'_j, j)$ for all j and consequently $\delta(\Pi) \leq \delta(\Pi')$. □

While this version of fixed core partitioning is rather trivial to solve, it can still provide us with some insights. Both restrictions strongly simplify the problem in different ways: The restricted degree means that for each node there is only one block we need to consider as target for the assignment. Thus each block can be optimized individually, without considering any interactions between different blocks. On the other hand, the unit node weights allow us to easily find the optimum for each block by sorting.

Note that using the same technique, we can solve a different version of the problem with unit edge weights but arbitrary node weights. Instead of sorting the nodes of each block by their edge weights, we sort in increasing order of node weights and assign the nodes greedily. However, in contrast to the previous formulation, finding a feasible solution (that does not exceed the maximum block weights) is hard in itself, thus rendering our algorithm insufficient. As shown in the following, the reason is that finding a feasible solution amounts to solving a packing problem.

Theorem 4.7. *The fixed core partitioning problem (with arbitrary node weights and unit edge weights) where all nodes have at most degree one is NP-hard. Moreover, it is NP-hard to decide whether a feasible solution even exists.*

Proof. We provide a reduction of the partition problem that asks for a partition of a set S of natural numbers into two subsets S_1 and S_2 such that $\sum_{a \in S_1} a = \sum_{b \in S_2} b$. Let $S = \{a_1, \ldots, a_n\}$ be an instance of the partition problem and let $t = \sum_{a \in S} a$ be the sum of the elements (w.l.o.g. is t even). We construct a fixed core partitioning instance with $k = 2$, block weights $c_1 = c_2 = \frac{t}{2}$ and nodes $P = \{p_1, \ldots, p_n\}$ with $c(p_i) \coloneqq a_i$. Edges can be chosen arbitrarily.

As $c(P) = t$, any feasible solution of this instance must assign a weight of $\frac{t}{2}$ to both blocks. Therefore, a feasible solution $\{P_1, P_2\}$ corresponds to a solution $\{a_i \mid p_i \in P_1\}$ of S and vice versa. □

Clearly, since it is NP-hard to determine whether a feasible solution exists, it is also impossible to provide an approximation algorithm with any guarantee regarding the solution quality. However, apart from being not very interesting, this case is also not representative of applications of the problem. The graph partitioning problem and hence the star partitioning problem includes the imbalance parameter ε that provides freedom to move nodes between blocks without ensuring perfect balance. In addition, it is possible to find a good approximation of minimal block weights, as this is equivalent to the job scheduling problem (which asks to distribute jobs to a set of identical machines such that the maximum processing time is minimized) – each block corresponds to a machine and each peripheral node corresponds to a job [31, 50].

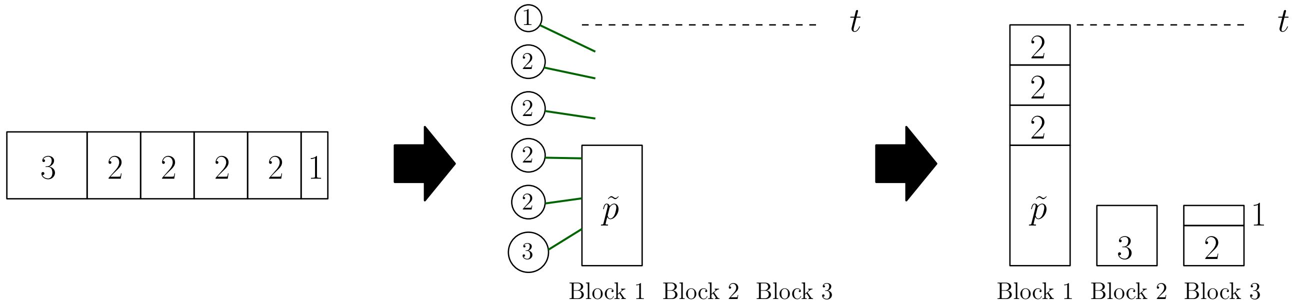


Figure 5: Example of reducing the partition problem to relaxed fixed core partitioning with degree one. The left side depicts the instance of the partition problem, which is reduced to the core partitioning instance in the center (note that for an optimal solution, $\tilde{p}$ can only be assigned to block 1). The right side depicts a solution of the core partitioning instance.

Therefore, we consider a simplified version of the problem where we can always find a balanced solution. This is achieved by relaxing the maximum allowed block weights, as defined in the following.

Definition 4.8 (Relaxed Star Partitioning)**.** *Let (G, P, k, ε') be an instance of the star partitioning problem and let $c_{max} = \max\{c(u) \mid u \in P\}$ be the maximum weight of a peripheral node. We call the instance relaxed, if $\varepsilon' \left\lceil \frac{c(V)}{k} \right\rceil \geq c_{max}$. Similarly, an instance of the fixed core partitioning problem is relaxed if $c_j - \left\lceil \frac{c(P)}{k} \right\rceil \geq c_{max}$ for every block j.*

Informally, it means that the maximum block weight $\max_j c_j$ exceeds the average block weight at least by c_{max}. Given any partition of the peripheral nodes, there is at least one block with weight below $\left\lceil \frac{c(P)}{k} \right\rceil$ to which the heaviest node can be assigned.
The relaxed problem with unit edge weights but arbitrary node weights can be solved optimally with the greedy algorithm as described above. Nevertheless, the relaxed problem is still a hard problem if we also allow arbitrary edge weights.

Theorem 4.9. *The relaxed fixed core partitioning problem (with arbitrary node and edge weights) where all nodes have at most degree one is NP-hard.*

Proof. We use an approach similar to the reduction given in Theorem 4.7: Let $S = \{a_1, \ldots, a_n\}$ be an instance of the partition problem and let $t = \sum_{a \in S} a$. Note that we can w.l.o.g. assume that the maximum weight of an element is at most $\frac{t}{2}$ (otherwise, the problem reduces to a trivial no-instance). We define $P = \{p_1, \ldots, p_n\}$ with $c(p_i) \coloneqq a_i$. Unlike before, we assign for each node a single edge to the first block with weight equal to the node weight, i.e., $\omega(p_i, 1) \coloneqq a_i$. Further, we add a dummy element $\tilde{p}$ with $c(\tilde{p}) \coloneqq \frac{t}{2}$ and one edge $\omega(\tilde{p}, 1) \coloneqq t$. We use $k = 3$ and for each block we set the maximum weight to t. The instance is considered a yes-instance exactly if there is a partition Π_P with $\delta(\Pi_P) \leq \frac{t}{2}$. An illustration of the reduction is given in Figure 5.
Note that with $c(P) = \frac{3t}{2}$, the instance is relaxed since $c_j - \left\lceil \frac{c(P)}{k} \right\rceil = t - \frac{t}{2} = \frac{t}{2} = c_{max}$ for every j. To prove correctness, let us first consider a yes-instance of the fixed core partition problem with a solution $\Pi_P = \{P_1, P_2, P_3\}$. It holds that $\tilde{p} \in P_1$, as otherwise $\delta(\Pi_P) \geq t$. Further, $c_1 = t$ and $c(P) = \frac{3t}{2}$ implies that $c(P \setminus P_1) \geq c(P) - c_1 = \frac{t}{2}$. On the other hand, because of $c(p_i) = \omega(p_i, 1)$ we have $c(P \setminus P_1) = \omega(P \setminus P_1, 1) = \delta(\Pi_P) \leq \frac{t}{2}$. Thus $c(P \setminus P_1) = \frac{t}{2}$, which means that $P \setminus P_1$ implies a solution of the partition problem instance.

For the reverse direction, let $S_1 \subseteq S$ be a solution of the partition problem instance. We can choose $P_1 := \{\tilde{p}\} \cup \{p_i \mid a_i \in S_1\}$ and observe that because the sum of S_1 is $\frac{t}{2}$, we have $c(P_1) = t$ and $\omega(P \setminus P_1, 1) = \frac{t}{2}$, i.e., P_1 induces a partition of P with value $\frac{t}{2}$. □

Note that we need a dummy element for the reduction because the highest number in S could have value $\frac{t}{2}$. In this case, we would need to choose a maximum block weight larger than $\frac{t}{2}$ for each block which would not allow to enforce a partition with block weight equal to $\frac{t}{2}$.

If we compare Theorem 4.7 and Theorem 4.9, it is important to notice that while both statements show that the problems are NP-hard, the structure of the proofs reveals one major difference: In the former, the difficulty is in finding a feasible solution. In the relaxed case, a feasible solution is trivial to find, but it is hard to find an optimal solution. More precisely, the difficulty is to select for each block a subset of nodes where the edge weights connected to this block are maximized. This is basically a knapsack problem – and as we show in the following, solving the relaxed problem variant with degree one is mostly equivalent to solving this knapsack problem.

But first, let us observe one more useful property of the relaxed problem variant: If $\Pi_P = \{P_1, \dots, P_k\}$ is a partial partition respecting the maximum block weights, we can calculate a feasible partition $\Pi'_P = \{P'_1, \dots, P'_k\}$ of P with $P_j \subseteq P'_j$ for all j by iteratively assigning each node in $P \setminus \bigcup_{j=1}^{k} P_j$ to a block with sufficient allowed weight. Therefore, it is valid to specify an approximation algorithm without precisely defining how to handle unassigned nodes. This allows us to focus the description of an algorithm on the essential part.

Theorem 4.10. *Let $\mathcal{A}$ be an approximation algorithm for the min-knapsack problem with approximation ratio R and running time $g(n)$. Then, we can construct an approximation algorithm $\mathcal{A}'$ for relaxed fixed core partitioning where all nodes have at most degree one with approximation ratio R and a running time of $\mathcal{O}(k \cdot g(|P|) + |P|)$. If g is convex, the running time is bound by $\mathcal{O}(g(|P|))$.*

Proof. First, we calculate for each block j the set $K_j := \{u \in P \mid \omega(u, j) > 0\}$ of candidates that might be assigned to block j in linear time. Note that the K_j are disjoint as the node degrees are bound by one. For each j we consider the min-knapsack instance with elements K_j, where the weight of an element u is $c(u)$ and the gain is $\omega(u, j)$. The minimum required weight for the instance is $w_j := c(K_j) - c_j$. Let $\mathcal{A}(K_j, w_j)$ be the solution calculated by $\mathcal{A}$. We define $P_j := K_j \setminus \mathcal{A}(K_j, w_j)$ and output $\Pi_P := \{P_1, \dots, P_k\}$. Nodes not assigned by Π_P are assigned to blocks with sufficient allowed weight (as explained above).

$\mathcal{A}(K_j, w_j) \geq w_j$ implies $c(P_j) \leq c(K_j) - w_j = c_j$ by definition of w_j, thus Π_P is a feasible solution. Consider an optimal solution $\Pi_{OPT} = \{P'_1, \dots, P'_k\}$. Then, $\delta(\Pi_{OPT}) = \sum_{j=1}^{k} \omega(K_j \setminus P'_j, j)$. Note that $K_j \setminus P'_j$ is a solution of the min-knapsack instance for block j. Using the approximation ratio of $\mathcal{A}$, we therefore get

$$\delta(\Pi_P) = \sum_{j=1}^{k} \omega(K_j \setminus P_j, j) = \sum_{j=1}^{k} \omega(\mathcal{A}(K_j, w_j), j) \leq \sum_{j=1}^{k} R \cdot \omega(K_j \setminus P'_j, j) = R \cdot \delta(\Pi_{OPT})$$

If g is convex, due to the K_j being disjoint the running time is $\mathcal{O}(|P|) + \sum_{j=1}^{k} g(|K_j|) \leq \mathcal{O}(|P|) + g(\sum_{j=1}^{k} |K_j|) = \mathcal{O}(|P|) + g(|P|) \in \mathcal{O}(g(|P|))$. □

To provide some concrete examples using Theorem 4.10: We can achieve an approximation ratio $R = 2$ with a running time of $\mathcal{O}(|P| \log |P|)$ when using the GR algorithm or an approximation ratio $R = \frac{3}{2}$ with a running time of $\mathcal{O}(|P|^2)$ when using the GR* algorithm.

It is important to emphasize that the knapsack problem considered here is the inverted version of the standard knapsack problem, which asks for a subset of elements that maximizes the gain for a given maximum allowed weight. Instead, the 0-1 min-knapsack problem is the standard 0-1 knapsack problem with inverted optimization objective. The reason we need to use this variant is that it matches the optimization objective for fixed core partitioning, which is formulated in terms of minimizing the cut edges. If instead we consider fixed core partitioning with the inverted objective (i.e., maximizing the block-internal edges) we can achieve a similar result to Theorem 4.10 by using the standard knapsack problem.

We could also ask whether this is the best we can do, i.e., whether it is possible to achieve a better approximation using an approach that does not rely on solving the min-knapsack problem. At least for fixed core partitioning, the answer is no: Both problems are effectively equivalent. We do not provide a detailed proof, as it is rather technical and not particularly insightful. But the idea is to construct a reduction that works similar to the one described in Lemma 4.6, which leads to a statement that is the inversion of Theorem 4.10: Any algorithm that solves relaxed fixed core partitioning with degree one with approximation ratio R leads to an algorithm that solves the min-knapsack problem with the same approximation ratio and only constant running time overhead, provided the original algorithm has polynomial running time.

4.3. Star Partitioning with Arbitrary Degree

In the following, we lift the degree-restriction used in the previous section and allow arbitrary node degrees for peripheral nodes. As we will see, this makes it harder to provide good approximation guarantees for $k \geq 3$. A good first step is to choose for a node u the block j that maximizes $\omega(u, j)$, but it is obvious that this is not optimal.

In general, the structure of the problem resembles a matching problem (although with weights). At first, let us consider the case with unit node weights and arbitrary edge weights – which effectively *is* a maximum weighted bipartite matching problem.

Theorem 4.11. *Given a fixed core partitioning instance with maximum block weights $c_1, \ldots, c_k$ and nodes P with weight functions $c\colon P \to \mathbb{R}_{\geq 0}$ and $\omega\colon P \times [k] \to \mathbb{R}_{\geq 0}$ where all nodes have weight one, we can compute an optimal solution in $\mathcal{O}(k^3|P|^3)$ time.*

Proof. We construct an equivalent instance of the maximum weighted bipartite matching problem (see Definition 2.8): We define the considered graph as the complete bipartite graph $G = (L \cup R, E)$ with parts $L := \bigcup_{j=1}^{k} B_j$ and $R := P$. Each B_j is a set of $\min\{c_j, |P|\}$ nodes (the size of B_j is the equivalent of the maximum block weight for the matching), such that the B_j are pairwise disjoint. For each pair $u \in P$, $b_j \in B_j$, the edge weight is defined as $\omega(\{u, b_j\}) := \omega(u, j)$.

Let $\Pi_P = \{P_1, \ldots, P_k\}$ be a solution of the fixed core partitioning instance. We define a matching M as $M := \bigcup_{j=1}^{k} M_j$, where M_j is an arbitrary matching between B_j and P_j with size $|P_j|$ (exists because $|P_j| = c(P_j) \leq c_j \leq |B_j|$). Note that $\omega(M_j) = \omega(P_j, j)$. Consequently, due to the P_j and B_j being disjoint, M is a matching with value $\omega(M) = \sum_{j=1}^{k} \omega(P_j, j)$.

Considering the other direction, it is easy to see that a given matching M similarly induces a solution $\{P_1, \ldots, P_j\}$ with $\omega(M) = \sum_{j=1}^{k} \omega(P_j, j)$. In summary, a maximum weighted matching in G induces a feasible partition $\Pi_P = \{P_1, \ldots, P_j\}$ that maximizes $\sum_{j=1}^{k} \omega(P_j, j)$ and is thus optimal.

With regards to the running time, the maximum weighted bipartite matching problem can be solved with the Hungarian method in $\mathcal{O}(|L \cup R|^3)$ time [21]. Due to $|L| \leq k|P|$ and $|R| = |P|$, the total running time is in $\mathcal{O}(k^3|P|^3)$. □

It should be noted that the constructed matching instance has some specific properties: All nodes in a block B_j have the same neighbors with identical edge weights. It might be possible to use an optimized maximum weight matching algorithm that makes use of this to achieve a better running time.

Before considering the most general (relaxed) case, there is one more simplified variant that is interesting to consider: The case with $k = 2$, which we call the *star bipartitioning* problem. We can observe that for a given node u, either $\omega(u, 1)$ or $\omega(u, 2)$ is necessarily included in the cut. Thus we can effectively ignore whether a node is connected to both blocks – only the difference between the edge weights is actually important. We formalize this by defining for a given node $u \in P$ the *minimum adjacent weight* $\omega_{min}(u) \coloneqq \min_{j \leq k} \omega(u, j)$. The definition extends to sets using $\omega_{min}(U) \coloneqq \sum_{u \in U} \omega_{min}(u)$.

Lemma 4.12. *Let (P, c, ω) be an instance of the fixed core partitioning problem. Then there is an equivalent instance (P, c, ω') with the same block weights, where $\delta_\omega(\Pi_P) = (k-1)\omega_{min}(P) + \delta_{\omega'}(\Pi_P)$ for each solution Π_P. If $k = 2$, the nodes of the new instance have at most degree one.*

Proof. We define $\omega'(u, j) \coloneqq \omega(u, j) - \omega_{min}(u)$ for $u \in P$, $j \leq k$. Consequently, if $k = 2$ then either $\omega'(u, 1) = 0$ or $\omega'(u, 2) = 0$ for each node u. Thus, u has at most degree one. Consider a solution $\Pi_P = \{P_1, \ldots, P_k\}$. For $u \in P$ we define the assigned block as $b(u) \coloneqq i$ with $u \in P_i$.

$$\begin{aligned}\delta(\Pi_P) &= \sum_{j=1}^{k} \omega(P \setminus P_j, j) = \sum_{u \in P} \sum_{j \neq b(u)} \omega(u, j) = \sum_{u \in P} \left((k-1)\omega_{min}(u) + \sum_{j \neq b(u)} \omega(u, j) - \omega_{min}(u) \right) \\ &= (k-1)\omega_{min}(P) + \sum_{u \in P} \sum_{j \neq b(u)} \omega'(u, j) = (k-1)\omega_{min}(P) + \delta_{\omega'}(\Pi_P)\end{aligned}$$

□

For $k = 2$, the problem can be reduced to the case where nodes have at most degree one as described in Section 4.2 and solved as described in Theorem 4.10. Effectively, in the relaxed case with $k = 2$ we solve a knapsack problem for the block that has higher load, placing all remaining nodes in the second block.

The bipartitioning variant is not only of theoretical interest, but might also be important in practice: Most multilevel partitioners apply recursive bipartitioning either directly on the input graph or as part of the initial partitioning phase [13, 39, 61, 63].

We now turn to the general case with $k \geq 2$ for which we can construct an approximation algorithm with a straightforward approach.

Theorem 4.13. *Let $\mathcal{A}$ be an approximation algorithm for the min-knapsack problem with approximation ratio R and running time $g(n)$. Then, we can construct an approximation algorithm $\mathcal{A}'$ for relaxed fixed core partitioning with approximation ratio $R + 1$ and a running time of $\mathcal{O}(k \cdot g(|P|) + |P|)$ (or $\mathcal{O}(g(|P|))$ if g is convex).*

Proof. We use the same construction as in Theorem 4.10, except that we adjust the definition of the candidate set K_j of a block. We choose K_j such that it contains the $u \in P$ where $\omega(u, j)$ is maximal for $j \leq k$. If there are multiple blocks with maximal adjacent weight to u, we pick one at random (such that the K_j are still a partition of P). In summary, we solve for each block a knapsack problem containing the nodes with the strongest connection to this block.

To prove the approximation ratio, we first define for a node $u \in K_j$ the *internal weight* $\text{int}(u) := \omega(u, j)$ and the *external weight* $\text{ext}(u) := \sum_{i \neq j} \omega(u, i)$, extending the definition to sets as usual. For a given solution $\Pi_P = \{P_1, \dots, P_k\}$ calculated by $\mathcal{A}'$, it holds that $P_j \subseteq K_j$ (remember, we allow Π_P to be only a partial partition) and thus

$$\delta(\Pi_P) = \sum_{j=1}^{k} \omega(K_j \setminus P_j, j) + \text{ext}(K_j) = \text{int}(\bigcup_j K_j \setminus P_j) + \text{ext}(P) \tag{6}$$

Let $\Pi_{OPT} = \{P'_1, \dots, P'_k\}$ be an optimal solution. Reformulating the approximation guarantee of the knapsack algorithm $\mathcal{A}$ in these terms, we get $\text{int}(K_j \setminus P_j) \leq R \cdot \text{int}(K_j \setminus P'_j)$ for each block. If $\text{ext}(P) \leq \frac{1}{R}\text{int}(\bigcup_j K_j \setminus P_j)$, then

$$\delta(\Pi_P) \leq^{(6)} (1 + \frac{1}{R})\text{int}(\bigcup_j K_j \setminus P_j) \leq (R+1)\text{int}(\bigcup_j K_j \setminus P'_j) \leq (R+1)\delta(\Pi_{OPT})$$

If $\text{ext}(P) > \frac{1}{R}\text{int}(\bigcup_j K_j \setminus P_j)$, then $\text{int}(\bigcup_j K_j \setminus P_j) \leq R \cdot \text{ext}(P)$ and thus

$$\delta(\Pi_P) \leq^{(6)} (R+1)\text{ext}(P) \leq (R+1)\delta(\Pi_{OPT})$$

□

To summarize the approach, we achieve a somewhat reasonable approximation ratio by simply ignoring that nodes are connected to more than one block. Instead we try to assign each node to the block it is most strongly connected to. This result is quite remarkable, as there exists no constant factor approximations for the general graph partitioning problem [10].

Algorithm 1: Fixed Core Partitioning

```
1  Function fixedCorePartitioning(P, c, ω, {c_1, ..., c_k})
     Input: Nodes P, weight functions c and ω, maximum allowed block weights c_1, ..., c_k
2    ω' ← subtractMinIncidentWeight(ω)                     // as in Lemma 4.12
3    P_1, ..., P_k ← ∅, ..., ∅
4    U ← ∅                                                 // unassigned nodes
5    foreach j ≤ k do
6        K_j ← {p ∈ P | j = argmax_{i≤k} ω'(p, i)}  // random tie breaking if max is ambiguous
7        L_j ← minKnapsack(K_j, c, ω', c_j)
8        P_j ← K_j \ L_j
9        U ← U ∪ L_j
10   sortDescendingByWeightRatio(U, c, ω')             // sort by max_j ω'(p, j) · c(p)^{-1}
11   foreach p ∈ U do
12       I ← {j ≤ k | c(p) ≤ c_j − c(P_j)}          // blocks where p can still be assigned
13       j ← argmax_{i∈I} ω'(p, i)
14       P_j ← P_j ∪ {p}                                   // assign greedily
     Output: Π_P = {P_1, ..., P_k}
```

In practice, this approach should be combined with a greedy algorithm to move nodes that were not assigned by the knapsack algorithm to other blocks. This is illustrated in Algorithm 1 which is a high-level description of our implementation (note that the actual implementation is parallel). We use the GR algorithm for the min knapsack problem and thus achieve an approximation ratio of 3 with a running time of $\mathcal{O}(|P| \log |P|)$.

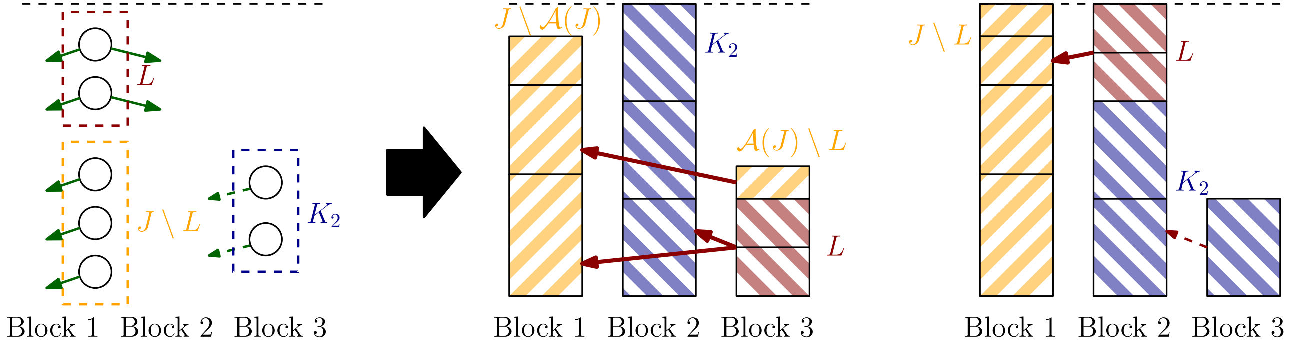


Figure 6: Illustration that the bound given in Theorem 4.13 is tight. The left side depicts the structure of the constructed instance. The edges to the different blocks are represented by arrows. The center shows the approximate solution and the right side the optimal solution. Edges that are cut by the respective solution are marked in red.

While we already argued that the approximation ratio R of the min-knapsack algorithm is a lower bound on the achievable approximation ratio in Section 4.2, there is still a lot of room for improvement between $R+1$ and R.

Before we discuss other approaches, let us examine whether the bound of $R+1$ for the presented approach is tight. To do this, we need an example input that achieves this ratio depending on the min-knapsack algorithm $\mathcal{A}$. For simplicity, we assume that there is a (family of) knapsack instance(s) J with an optimal solution L such that $\mathcal{A}(J) \supseteq L$ and $\omega(\mathcal{A}(J)) = (R-\varepsilon)\omega(L)$ for the solution calculated by $\mathcal{A}$, with arbitrary small ε. We define a core partitioning instance where the first block is basically equivalent to this knapsack instance, i.e., has the same maximum weight, candidate set $K_1 = J$, and the incident edge weights are equal to the gains of the knapsack instance. Let every node in $J \setminus L$ be incident to block 1, while nodes $u \in L$ are also incident to the second block with $\omega(u, 2) := \omega(u, 1) - 1$. Further, let K_2 be large enough to completely fill block 2, but with very low internal edge weight. As illustrated in Figure 6, the calculated solution $\Pi_{\mathcal{A}}$ will assign $\mathcal{A}(J)$ neither to block 1 nor to block 2, while the optimal solution Π_{OPT} assigns $J \setminus L$ to block 1 and L to block 2. Therefore $\delta(\Pi_{OPT}) = \omega(L, 1) + \tau$, where τ is the incident weight of the nodes that $\delta(\Pi_{OPT})$ removes from block 2. Thus

$$\begin{aligned}\delta(\Pi_{\mathcal{A}}) &= \omega(\mathcal{A}(J), 1) + \omega(\mathcal{A}(J), 2) = (R-\varepsilon)\omega(L, 1) + \omega(L, 2) \\ &= (R+1-\varepsilon)\omega(L, 1) - |L| = (R+1-\varepsilon)(\delta(\Pi_{OPT}) - \tau) - |L|\end{aligned}$$

We can scale the involved weights such that τ and $|L|$ are negligible and hence $\frac{\delta(\Pi_{\mathcal{A}})}{\delta(\Pi_{OPT})} \to R+1$.

4.4. Possible Approaches for Better Approximations

So far, we have seen that solving the general fixed core partitioning problem involves two distinct *kinds* of decisions to be made: First, we need to select for each node a block it should preferably be assigned to and subsequently, calculate an actual packing of the nodes into the block using a knapsack algorithm. The former decision resembles a matching problem (compare Theorem 4.11), while the latter is a min-knapsack problem. Through this point of view, we can attribute the approximation ratio of $R+1$ achieved in Theorem 4.13 as follows: We achieve a 2-approximation with regards to the matching problem and an R-approximation with regards to the knapsack problem, summing to a total approximation ratio of $R+1$.

It might be helpful to formalize this idea: We already know (more or less) how to handle the knapsack problem. Thus we want to focus on the matching part of the problem. Of course we

need to be aware that for achieving the best possible result, it might not work to solve both problems in isolation, as both decisions potentially influence each other. However, an isolated analysis might be quite helpful with regards to the best achievable approximation ratio. If we can not solve both parts of the problem independently, it seems very unlikely that we can find a better solution for the fixed core partitioning problem itself.

We can achieve this by relaxing the knapsack problem as follows: Given an instance of the fixed core partitioning problem with nodes P and a block j with candidate set $K_j \subseteq P$ and maximum weight c_j, we define a *relaxed packing* of K_j (in the sense of an LP-relaxation) as a function $f\colon K_j \to [0,1]$ with $\sum_{u \in K_j} c(u)f(u) \leq c_j$. The *gain* of f is $\Delta(f) \coloneqq \sum_{u \in K_j} \omega(u,j)f(u)$. Using this, we define the *gain* of K_j as $\Delta(K_j) \coloneqq \sup_f \Delta(f)$, i.e., the gain of an optimal packing. Fortunately, a relaxed optimal packing is simple to calculate.

Lemma 4.14. *We use $\omega(u) \coloneqq \omega(u,j)$ for convenience. Let $(u_1, \dots, u_m)$ be an ordering of K_j such that $\frac{\omega(u_i)}{c(u_i)} \geq \frac{\omega(u_j)}{c(u_j)}$ for $i \leq j$. Let i_0 be maximal such that $\sum_{i=1}^{i_0} c(u_i) \leq c_j$ and let g be defined as*

$$g(u_i) \coloneqq \begin{cases} 1, & \text{if } i < i_0 \\ \frac{1}{c(u_i)}(c_j - \sum_{i=1}^{i_0} c(u_i)), & \text{if } i = i_0 \\ 0, & \text{if } i > i_0 \end{cases}$$

Then, g is a relaxed packing that maximizes $\Delta(g)$, i.e., $\Delta(K_j) = \Delta(g)$.

Proof. Note that $\sum_{u \in K_j} c(u)g(u) \leq c_j$. Basically, we solve some linear inequalities. Let i,j be fixed with $i < j \leq m$ and let f be a relaxed packing retrieved from g with $f(u) = g(u)$ for $u \in K_j \setminus \{u_i, u_j\}$. By the definition of g and f being a relaxed packing, $f(u_i) \leq g(u_i)$ and $c(u_i)f(u_i) + c(u_j)f(u_j) \leq c(u_i)g(u_i) + c(u_j)g(u_j) \Rightarrow f(u_j) - g(u_j) \leq \frac{c(u_i)}{c(u_j)}(g(u_i) - f(u_i))$, thus

$$\begin{aligned} \Delta(f) - \Delta(g) =& \omega(u_i)\left(f(u_i) - g(u_i)\right) + \omega(u_j)\left(f(u_j) - g(u_j)\right) \leq \omega(u_i)\left(f(u_i) - g(u_i)\right) \\ &+ \omega(u_j)\frac{c(u_i)}{c(u_j)}\left(g(u_i) - f(u_i)\right) = \left(\omega(u_i) - \omega(u_j)\frac{c(u_i)}{c(u_j)}\right)\left(f(u_i) - g(u_i)\right) \leq 0 \end{aligned}$$

due to $\omega(u_i) - \omega(u_j)\frac{c(u_i)}{c(u_j)} \geq 0$ and $f(u_i) - g(u_i) \leq 0$. The statement follows by induction. $\square$

Informally, we retrieve the optimal relaxed packing through sorting the nodes by their gain per weight ratio, greedily adding nodes to the packing and splitting the first node that does not completely fit in the packing.

Definition 4.15 (Fixed Core Partitioning with Over-Committing)**.** *Given nodes P and block weights c_j, the fixed core partitioning problem with over-committing is to find a partition $\Pi = \{K_1, \dots, K_k\}$ that minimizes the relaxed cut $\Delta(\Pi) \coloneqq \sum_{j=1}^{k} \omega(P,j) - \Delta(K_j)$.*

As intended, we can observe that this provides a lower bound for the non-relaxed problem, i.e., $\min_\Pi \Delta(\Pi) \leq \min_\Pi \delta(\Pi)$. Essentially, it allows us to pretend we can solve the knapsack problem optimally, instead focusing on the matching part of the problem. Note that is not necessary anymore to relax the maximum block weights, as finding a valid solution is trivial – any partition is a valid solution. Now, we can formally analyze the approach given in Theorem 4.13 with regards to the variant with over-committing. We will see that the definition of over-committing corresponds well to the intuition that we have given for the approximation ratio.

Lemma 4.16. *Assigning each node u to a block that maximizes $\omega(u, j)$ yields a 2-approximation for fixed core partitioning with over-committing.*

Proof. Analogous to Theorem 4.13 with $R = 1$. □

While there is still some resemblance to a matching problem (i.e., choosing which node should be assigned to which block), the main difference compared to the case with unit node weights we have seen in Theorem 4.11 is exactly the arbitrary possible weights: The change applied to a matching can be modeled as a so-called alternating path that moves one node per step. This is not possible with arbitrary weight, as moving one node to another block might necessitate removing multiple other nodes to make space for the new node. This means that calculating the best possibility for assigning a node can involve an arbitrarily complex tree of moves, which seems unlikely to be possible in polynomial time. Considering this, it is not surprising that the problem is still NP-hard.

Theorem 4.17. *The fixed core partitioning problem with over-committing is NP-hard, even if it is restricted to unit edge weights.*

Proof. Once more, we provide a reduction to the partition problem. Let $S = \{a_1, \dots, a_n\}$ be an instance of the partition problem with total sum t, where t is w.l.o.g. even. We construct a fixed core partitioning instance with $k = 2$, block weights $c_1 = c_2 = \frac{t}{2}$ and nodes $P = \{p_1, \dots, p_n\}$. The node weights are $c(p_i) = a_i$ and the edge weights are $\omega(p_i, 1) = \omega(p_i, 2) = 1$. We consider the instance a yes-instance exactly if there is a partition Π with $\Delta(\Pi) \leq n$.

Note that $\sum_{j=1}^{k} \omega(P, j) = 2n$. Hence, given a yes-instance of the fixed core partitioning problem with $\{K_1, K_2\}$ as solution, $\Delta(\Pi) \leq n$ implies $\Delta(K_1) + \Delta(K_2) \geq n$. Further, $\Delta(K_j) = |K_j|$ holds if and only if $c(K_j) \leq c_j = \frac{t}{2}$ (otherwise, $\Delta(K_j) < |K_j|$). Consequently, $|K_1| + |K_2| = n = \Delta(K_1) + \Delta(K_2)$ and thus $c(K_1) = c(K_2) = \frac{t}{2}$, which means that K_1 is a solution of the partition problem instance. In the reverse direction, a solution $S_1 \subseteq S$ of the partition problem instance corresponds to a solution $\Pi = \{S_1, P \setminus S_1\}$ of the fixed core partitioning instance with $\Delta(\Pi) = n$. □

Considering the structure of the provided reduction, it seems that although the variant with over-committing allows using an optimal local packing for each block, we still need to solve something similar to a packing problem in order to decide how to assign the nodes to the blocks. While Theorem 4.17 does not necessarily exclude good approximations (approximating the partition problem works quite well [44]), it is not clear how to find one either.

Improving the Approximation. In the following, we propose an approach for fixed core partitioning with over-committing that could provide better results than Lemma 4.16. Consider an instance with nodes P and block weights $c_1, \dots, c_k$. For a given node $u \in P$, let $b_1, \dots, b_k$ be an order of the blocks such that $\omega(u, b_1) \geq \omega(u, b_2) \geq \dots \geq \omega(u, b_k)$. We define $\beta_j(u) := b_j$ for $j \leq k$, i.e., $\beta_1(u)$ is the block maximizing $\omega(u, \beta_1(u))$, $\beta_2(u)$ the block where the incident edge weight has the second largest value, and so on.

We propose to use Algorithm 2 for fixed core partitioning with over-committing. The idea is to consider both $\beta_1(u)$ and $\beta_2(u)$ as possible assignment for u, i.e., we consider *two slots* for each node – hence we call it the two-slot algorithm. The algorithm maintains for each block j a candidate set K_j of nodes that might be assigned to block j. K_j is represented as a balanced binary tree where each node has an assigned key. We initialize the candidate sets by inserting each node u to $K_{\beta_1(u)}$ using the key $\frac{\beta_1(u)}{c(u)} - \frac{\beta_2(u)}{2c(u)}$. As long as there is at least one block j where the node with minimum key u in K_j exceeds the maximum allowed block weight (i.e., an

Algorithm 2: Two-Slot Algorithm

```
1  Function twoSlotPartitioning(P, c, ω, {c_1, ..., c_k})
     Input: Nodes P, weight functions c and ω, maximum allowed block weights c_1, ..., c_k
2    K_1, ..., K_k ← ∅, ..., ∅                      // intialize candiate sets as binary trees
3    U ← ∅                                                        // unassigned nodes
4    foreach u ∈ P do
5      K_{β_1(u)}.insert(u, β_1(u)/c(u) − β_2(u)/2c(u))   // assign u to β_1(u) with key β_1(u)/c(u) − β_2(u)/2c(u)
6    while there is a K_j with c(K_j) − c(K_j.min()) ≥ c_j do
7      u ← K_j.min()
8      K_j.remove(u)
9      if j = β_1(u) then
10       K_{β_2(u)}.insert(u, β_2(u)/c(u))            // assign u to β_2(u) with key β_2(u)/c(u)
11     else if j = β_2(u) then
12       U ← U ∪ {u}                                                    // unassign u
13   greedily assign the nodes in U
     Output: Π = {K_1, ..., K_k}
```

optimal relaxed packing of K_j does not contain u), we update the assignment as follows: We remove u from K_j. If j is equal to $\beta_1(u)$, we move u to the second "slot" by assigning u to $\beta_2(u)$ using the key $\frac{\beta_2(u)}{c(u)}$. If u was already assigned to $\beta_2(u)$, we unassign u instead. Afterwards, remaining unassigned nodes are assigned in a greedy fashion (note that this step does not affect any approximation guarantees). Clearly, the running time of Algorithm 2 is in $\mathcal{O}(|P|\log|P|)$: Each operation on a binary tree requires logarithmic time and the algorithm uses a constant number of operations per node.

In its current form, the algorithm does not provide an improvement for the general case. However, we can prove an approximation guarantee for the special case where all nodes have weight one. Note that fixed core partitioning and fixed core partitioning with over-committing are effectively equivalent in the case with weight one nodes, as it is never necessary to "split" a node that does not fit in the block.

Theorem 4.18. *Given a fixed core partitioning instance with maximum block weight $c_1, \ldots, c_k$ and nodes P with weight functions $c\colon P \to \mathbb{R}_{\geq 0}$ and $\omega\colon P \times [k] \to \mathbb{R}_{\geq 0}$ where all nodes have weight one, Algorithm 2 computes a solution where the cut is within a factor of $\frac{3}{2}$ of the optimal solution.*

Proof. Let $\Pi_A = \{K_1, \ldots, K_k\}$ be the solution computed by Algorithm 2 and let $\Pi_{OPT} = \{P_1, \ldots, P_k\}$ be an optimal solution. Note that Π_A is a feasible partition because the loop condition (Line 6) and $c \equiv 1$ imply that $c(K_j) \leq c_j$ for all j. In the following, we only consider the nodes that are assigned differently in Π_A and Π_{OPT}. For this, let $\overline{K_j} \coloneqq K_j \setminus P_j$ be the differently assigned nodes in block j and let $\overline{K} \coloneqq \bigcup_{j\leq k} \overline{K_j}$. We will prove that $\delta(\Pi_A[\overline{K}]) \leq \frac{3}{2}\delta(\Pi_{OPT}[\overline{K}])$ by constructing a directed graph $G = (\overline{K}, E)$ and showing that the inequality holds for each component of G.

To categorize the nodes, let $S_1 \coloneqq \{u \in \overline{K} \mid u \in K_{\beta_1(u)}\}$ be the nodes assigned to their first slot, let $S_2 \coloneqq \{u \in \overline{K} \mid u \in K_{\beta_2(u)}\}$ be the nodes assigned to their second slot and let $S_3 \coloneqq \overline{K} \setminus (S_1 \cup S_2)$ be the remaining nodes. To define the edges of G, we construct for a given node u the outwards edges as follows: If $u \in S_2 \cup S_3$ and $u \in P_{\beta_1(u)}$ (i.e., Π_{OPT} provides a better assignment for u) than we add an outwards edge from u to a node in $\overline{K_{\beta_1(u)}}$. Similarly,

if $u \in S_3$ and $u \in P_{\beta_2(u)}$ we add an outwards edge from u to a node in $\overline{K_{\beta_2(u)}}$. Otherwise, u has no outwards edge. Consider a j such that $\overline{K_j}$ has at least one inwards edge. Then, $|K_j| = c_j$ as otherwise Algorithm 2 would have assigned at least one source node of an inwards edge to K_j. Further, the total inwards degree of $\overline{K_j}$ is bound by $|P_j \setminus K_j|$ by construction (we add at most one edge per node in $P_j \setminus K_j$). This means the inwards degree is at most $|\overline{K_j}|$. This property allows us to choose the edges such that each node has inwards degree at most one.

Since both the inwards and the outwards degree of every node is bound by one, each component of the graph is either a single node, a path or a cycle (with each edge oriented in the same direction). Further, nodes in S_1 have outwards degree zero while nodes in S_3 have inwards degree zero – otherwise, the source node of the inwards edge would have been assigned to the block of the target node. To analyze the added cut per component, we use the notation $\omega_A(u) := \omega(u, j)$ with $u \in K_j$ for the weight of the edge where the algorithm assigned u and $\omega_{OPT}(u) := \omega(u, j)$ with $u \in P_j$ for the weight of the edge where the optimal solution assigned u. Similarly, $\delta_A(u) := \sum_{j \leq k} \omega(u, j) - \omega_A(u)$ denotes the value u adds to $\delta(\Pi_A)$ and $\delta_{OPT}(u) := \sum_{j \leq k} \omega(u, j) - \omega_{OPT}(u)$ denotes the value u adds to $\delta(\Pi_{OPT})$. We extend the definitions to sets of nodes by summing the values.

Consider a component of G consisting of a single node u and assume $u \in P_j$ (i.e., Π_{OPT} assigns u to j). If $u \in S_1 \cup S_2$ then $d_{out}(u) = 0$ implies that $j \neq \beta_1(u)$ and thus $\delta_A(u) \leq \delta_{OPT}(u)$. If $u \in S_3$ then $d_{out}(u) = 0$ implies that $j \notin \{\beta_1(u), \beta_2(u)\}$. Therefore, $\omega(u, j) \leq \frac{1}{3} \sum_{i \leq k} \omega(u, i)$ which implies $\delta_{OPT}(u) \geq \frac{2}{3} \sum_{i \leq k} \omega(u, i)$ and thus $\delta_A(u) \leq \sum_{i \leq k} \omega(u, i) \leq \frac{3}{2} \delta_{OPT}(u)$.

Consider a component $C = (V_C, E_C)$ that is a cycle. Each node of the cycle is in S_2 since it has both an inwards and an outwards edge. Let (u, v) be an edge in E_C. Since u was not assigned to $\beta_1(u)$, the inequality $\omega(v, \beta_2(v)) \geq \omega(u, \beta_1(u)) - \frac{1}{2}\omega(u, \beta_2(u))$ holds. Summing this over all nodes and rearranging the equation we get $\frac{3}{2} \sum_{u \in V_C} \omega(u, \beta_2(u)) \geq \sum_{u \in V_C} \omega(u, \beta_1(u))$. From this, we can conclude $\delta_{OPT}(V_C) \geq \frac{2}{3} \sum_{u \in V_C} \omega(u, \beta_1(u))$ and

$$\delta_A(V_C) - \delta_{OPT}(V_C) = \sum_{u \in V_C} \omega(u, \beta_1(u)) - \sum_{u \in V_C} \omega(u, \beta_2(u)) \leq \frac{1}{3} \sum_{u \in V_C} \omega(u, \beta_1(u))$$

Thus $\delta_A(V_C) - \delta_{OPT}(V_C) \leq \frac{1}{2} \delta_{OPT}(V_C)$ which means $\delta_A(V_C) \leq \frac{3}{2} \delta_{OPT}(V_C)$.

Consider a component $C = (V_C, E_C)$ that is a path $[u = v_1, v_2, \ldots, w = v_l]$, with edges oriented towards w. Then $u \in S_2 \cup S_3$, $v_2, \ldots, v_{l-1} \in S_2$ and $w \in S_1 \cup S_2$. First, we consider the case $w \in S_2$. From the assignments of the nodes, we get the following inequalities: First, $\omega_{OPT}(w) \leq \omega(w, \beta_2(w))$ and $\delta_{OPT}(w) \geq \omega(w, \beta_1(w)) + \omega(w, \beta_2(w))$. For $i \in \{2, \ldots, l-1\}$ we have $\omega(v_{i+1}, \beta_2(v_{i+1})) \geq \omega(v_i, \beta_1(v_i)) - \frac{1}{2}\omega(v_i, \beta_2(v_i)) = \omega_{OPT}(v_i) - \frac{1}{2}\omega(v_i, \beta_2(v_i))$. For $i = 1$, the outwards edge of v_1 can lead to block $\beta_1(v_1)$ or to block $\beta_2(v_1)$. In the first case, we also get $\omega(v_2, \beta_2(v_2)) \geq \omega_{OPT}(v_1) - \frac{1}{2}\omega(v_i, \beta_2(v_1))$. Otherwise, we get $\omega(v_2, \beta_2(v_2)) \geq \omega(v_1, \beta_2(v_1)) = \omega_{OPT}(v_1) \geq \omega_{OPT}(v_1) - \frac{1}{2}\omega(v_1, \beta_2(v_1))$. Thus

$$\begin{aligned}
\delta_A(V_C) - \delta_{OPT}(V_C) &= \sum_{i=1}^{l} \omega_{OPT}(v_i) - \omega_A(u) - \sum_{i=2}^{l} \omega(v_i, \beta_2(v_i)) \\
\leq \sum_{i=1}^{l} \omega_{OPT}(v_i) - \sum_{i=1}^{l-1} \Big(\omega_{OPT}(v_i) - \frac{1}{2}\omega(v_i, \beta_2(v_i))\Big) &= \omega_{OPT}(w) + \frac{1}{2} \sum_{i=1}^{l-1} \omega(v_i, \beta_2(v_i)) \\
\leq \frac{1}{2} \Big(\omega(w, \beta_1(w)) + \omega(w, \beta_2(w)) + \sum_{i=1}^{l-1} \omega(v_i, \beta_2(v_i))\Big) &\leq \frac{1}{2} \delta_{OPT}(V_C)
\end{aligned}$$

which implies $\delta_A(V_C) \leq \frac{3}{2} \delta_{OPT}(V_C)$.

In case of $w \in S_1$, we get different inequalities: $\omega_{OPT}(w) \leq \omega(w, \beta_2(w))$ with $\delta_{OPT}(w) \geq \omega(w, \beta_1(w))$. If we handle the two cases for the outwards edge of v_1 similar to before, we get

$\omega(w, \beta_1(w)) - \frac{1}{2}\omega(w, \beta_2(w)) \geq \omega_{OPT}(v_{l-1}) - \frac{1}{2}\omega(v_{l-1}, \beta_2(v_{l-1}))$ and $\omega(v_{i+1}, \beta_2(v_{i+1})) \geq \omega_{OPT}(v_i) - \frac{1}{2}\omega(v_i, \beta_2(v_i))$ for $i \in \{1, \dots, l-2\}$. Therefore

$$\begin{aligned}
\delta_A(V_C) - \delta_{OPT}(V_C) &= \sum_{i=1}^{l} \omega_{OPT}(v_i) - \omega_A(u) - \sum_{i=2}^{l-1} \omega_A(v_i) - \omega(w, \beta_1(w)) \\
&\leq \sum_{i=1}^{l} \omega_{OPT}(v_i) - \sum_{i=1}^{l-2} \Big(\omega_{OPT}(v_i) - \frac{1}{2}\omega(v_i, \beta_2(v_i)) \Big) - \omega(w, \beta_1(w)) \\
&\leq \sum_{i=1}^{l} \omega_{OPT}(v_i) - \sum_{i=1}^{l-1} \Big(\omega_{OPT}(v_i) - \frac{1}{2}\omega(v_i, \beta_2(v_i)) \Big) - \frac{1}{2}\omega(w, \beta_2(w)) \\
&= \omega_{OPT}(w) - \frac{1}{2}\omega(w, \beta_2(w)) + \frac{1}{2}\sum_{i=1}^{l-1} \omega(v_i, \beta_2(v_i)) \leq \frac{1}{2}\sum_{i=1}^{l} \omega(v_i, \beta_2(v_i)) \leq \frac{1}{2}\delta_{OPT}(V_C)
\end{aligned}$$

and we can conclude $\delta_A(V_C) \leq \frac{3}{2}\delta_{OPT}(V_C)$. □

Regarding the case with arbitrary weights, the proof for Theorem 4.18 uses that every node has weight one to construct a graph on the nodes. However, it could be possible to adapt the proof to arbitrary weights by splitting the nodes such that the (sub-)nodes in a single component have equal weight. Instead, the reason that Algorithm 2 does not achieve a $\frac{3}{2}$ approximation for the general case is that it does not handle the last node of a block well: For every block there might be one node where only a fraction of its weight is assigned by the according relaxed packing. This can be used to construct an instance where moving such a node to another block yields a factor two improvement.

While the current version of the algorithm does not achieve the same approximation guarantee for arbitrary weights, it indicates that it might be possible to use the two-slot technique for constructing a generalized approximation algorithm. Specifically, this approach could be combined with a knapsack algorithm to solve the fixed core partitioning problem with arbitrary weights. For this, there are at least two possible approaches: We could apply the two-slot algorithm to compute an initial assignment and then apply the knapsack algorithm to each block in isolation. Second, we could replace the use of binary trees with the knapsack algorithm, i.e., we adjust Algorithm 1 such that in each step the knapsack algorithm is applied to one of the blocks and the nodes that are not included in the result are moved to their second slot (or unassigned). This would likely require quadratic running time but might provide better results.

5. Engineering a Multilevel Star Partitioner

In the following, we discuss the practical aspects of building a partitioning algorithm that can exploit the properties of star graphs. The ultimate goal is to develop an approach that yields significant improvements for such instances and works also good for all other graphs. We discuss different strategies for adapting the phases of the multilevel paradigm from a theoretical perspective and from that we derive a practical implementation. This includes the detection of peripheral nodes and all phases of the multilevel scheme: a coarsening scheme handling core and peripheral nodes differently, the integration of a fixed core partitioning algorithm into initial partitioning and the handling of peripheral nodes during refinement.

While developing a formal model for *star graphs* and discussing the partitioning problem on such graphs from a a theoretical perspective in Section 4, we already noted that this model does not directly map to most real-world instances. Instead, we will use the more informal term *star-like* to refer to instances with a significant amount of nodes that can be considered peripheral. Specifically, we are interested in instances where we can achieve improvements over traditional multilevel partitioning by handling peripheral nodes differently. Note that these real-world instances are different from the definition of star graphs in at least two aspects: First, there can be edges connecting peripheral nodes, which is not the case in our theoretical model. We can assume that these edges are sparse enough to not be relevant for the overall structure of the graph, but they can still affect the resulting cut of the partition. Second, the core of real-world instances tends to be of comparable size to the peripheral nodes. This means that the proposed star partitioning algorithms are not directly applicable, as their running time is exponential in the size of the core. However, we will see that they are still useful as a component within the initial partitioning.

Traditional Multilevel Partitioning. As a first step, we explain how a traditional multilevel algorithm behaves on star-like instances. In the *coarsening* phase, we apply a series of edge contractions to the graph until the graph is small enough for initial partitioning. Although there are many different coarsening approaches (see Ref. [14] for an overview), most have in common that they iterate over all nodes and then try to find a contraction partner for each node according to a rating function. Since there are many peripheral nodes and they are mostly adjacent to core nodes, this results in the peripheral nodes being contracted onto the core nodes. Consequently, the core nodes become heavy nodes in the smallest graph, destroying the star-like structure in the process (see Appendix A for a visualized example). Moreover, the weight of the heaviest node in the smallest graph is often restricted by an upper weight limit to make it easier for initial partitioning to find a feasible solution. Since peripheral nodes are contracted onto core nodes, we might not be able to effectively reduce the size of the graph due to the weight constraint.[3]

Clearly, this makes it impossible for the *initial partitioning* to exploit the structure of the star graph. For a good solution, the core often needs to be placed in one block of the partition (see Section 4). But this is prohibited by the weight of the contracted nodes – even if a part of the peripheral nodes remains uncontracted, the contracted nodes still increase the weight of the core such that it can not be assigned to a single block anymore. Because the global structure of the partition is determined by the initial partitioning, it is also unlikely that the *refinement* can improve it significantly. Most refinement algorithms search for local improvements by moving nodes greedily to other blocks [14]. As core nodes are heavy nodes in the smallest graph, placing

[3] This is particularly likely to occur for coarsening algorithms that use a global contraction order and apply contractions between core nodes first. Afterwards, all remaining contraction partners for peripheral nodes have already reached the weight limit.

them in one block would involve a large number of moves, which is unlikely to occur in practice. Considering this, it is not surprising that we can achieve large improvements over traditional multilevel algorithms on star-like instances, as we already demonstrated in the introduction and later in our experimental evaluation in more detail.

Algorithm 3: Multilevel Partitioning for Star-like Graphs

```
1  Function partition(G, ε, k)
       Input: Graph G = (V, E, c, ω), imbalance parameter ε, number of blocks k
2      if peripheral nodes detection as preprocessing then
3          P ← detectPeripheralNodes(G)
4          C_H ← coarsen(G[V \ P], ε, k)                    // contraction hierarchy of core
5      else
6          C_H, P ← coarsenAndDetectPeripheralNodes(G, ε, k)
7      P_H ← twoHopCoarsen(P, C_H, ε, k)         // contraction hierarchy of peripheral nodes
8      Π_C ← initialPartition(C_H.coarsest, P_H.coarsest, ε, k)
9      Π_P ← partitionFixedCore(Π_C, P_H, ε, k)                        // using Algorithm 1
10     Π ← refine(Π_C + Π_P, C_H + P_H, ε, k)              // refinement on complete graph
       Output: k-way partition Π
```

Algorithm Overview. In the following, we develop a *star partitioning* algorithm, i.e., a variant of the multilevel scheme that makes effective use of the properties of star-like graphs. A high-level overview of our approach is given in Algorithm 3. In order to exploit the properties of star-like graphs, it is necessary that the overall structure of the graph is preserved until initial partitioning – which is the most important feature of a star partitioning algorithm. To achieve this, we can either avoid contractions of peripheral nodes onto the core or undo problematic contractions before initial partitioning. Considering the second option, it is unclear which contraction we should revert and whether or not this has any advantage compared to forbidding the contraction. Therefore, we will focus on the first approach, i.e., avoiding contractions of peripheral nodes. Thus, the first step of the algorithm is to detect peripheral nodes and separate them from the core of the graph. Then, the regular coarsening is only applied to core nodes – which preserves the structure of the graph and also ensures that the separated nodes do not cause additional running time overhead. The separation of the peripheral nodes can be done as a preprocessing step (see Line 3), or alternatively by identifying nodes that should be separated during coarsening (see Line 6). In both cases we need to rely on heuristics, as there is no unambiguous way for detecting such nodes. We discuss this in detail in Section 5.1.

Once the coarsening of the core is done, the graph actually resembles the definition of a star graph, which means that we might be able to apply the techniques we developed in Section 4. However, the size of the coarsened graph would still be too large to directly apply a parametrized approach (which is exponential in the size of the core). For this reason, we instead develop approaches that modify existing techniques for initial partitioning such that they also consider peripheral nodes in Section 5.3. To handle the peripheral nodes efficiently, we apply a *two-hop coarsening* algorithm (compare Ref. [47]), contracting the separated nodes to a similar size as the core. This is discussed in Section 5.2. The initial partitioning then uses the coarsest level of the contraction hierarchy of both the core and the peripheral nodes. Afterwards, we apply our proposed fixed core partitioning algorithm to optimize the assignment of the peripheral nodes. During the uncontraction of the graph, we use traditional refinement algorithms on both the core of the graph and the peripheral nodes.

5.1. Detection of Peripheral Nodes

We already noted that we relaxed the definition of peripheral nodes for real-world instances, allowing them to be adjacent to each other now. This provides us with some leeway in the design of our peripheral node detection algorithm. Our basic idea for differentiating them from the core of the graph is the following: Because the effect of peripheral nodes on the cut is small, they can be moved to make room for nodes of higher degree. Consequently, the most distinctive feature of peripheral nodes is their low incident edge weight. In the following, we will present several techniques based on the incident edge weight of a node. This includes a local approach by comparing adjacent nodes as well as using statistical properties of the graph or a community detection algorithm.

Dynamic Detection in the Coarsening Phase. Our first approach is to directly compare the incident weight of neighbors. For a given node u, let $r(u) := \frac{\omega(I(u))}{c(u)}$ be the ratio of the weight of all incident edges of u relative to its node weight (w.l.o.g. $c(u) \neq 0$). We expect that $r(u)$ is comparatively low for peripheral nodes and comparatively high for core nodes. This might seem counter-intuitive since core nodes usually have higher node weight then peripheral nodes. However, consider an unweighted input graph: Here, core nodes have a high number of incident edges and peripheral nodes a low number of incident edges, while all nodes have equal weight. We can expect that the ratios are similar for coarser levels.[4]

Assume that the current coarsening pass considers u and v as possible contraction partners. To avoid contractions between core nodes and peripheral nodes, we compare the ratios of both nodes: We forbid the contraction if $\max\{\frac{r(u)}{r(v)}, \frac{r(v)}{r(u)}\} \geq s$ where s is a tuning parameter. However, our main goal is not to forbid such contractions, but to detect whether or not a node should be separated. This is a straightforward extension – we separate a node u if $\frac{r(v)}{r(u)} \geq s$ for every neighbor v of u (which means that there is no allowed contraction for u anyways). More precisely, we apply this detection step when searching for a contraction partner for a particular node in the coarsening phase (note that the nodes are processed in parallel). Then, the detected nodes can be separated while the calculated contractions are applied to the remainder of the graph.

There is one additional design choice that we consider: A node u can be adjacent to nodes that were separated on a previous level. It is unclear how this should affect the decision to contract or separate u. One possibility is to never separate a node that is adjacent to already separated nodes – which consistent with the notion that the peripheral nodes approximate an independent set – but could be too restrictive in practice. On the other hand, we can ignore separated nodes and proceed as described before. In our actual implementation, we decided to use a combination of both approaches. We define a second threshold $t \in [0, 1]$ and only separate a node u if $\omega(I'(u)) \leq t \cdot \omega(I(u))$, where $I'(u)$ is the set of incident edges restricted to already separated nodes and $I(u)$ is the set of all incident edges.

Detection based on Outlier Detection of Statistical Properties. While the described approach is simple to integrate into the existing coarsening algorithm, there is also an obvious drawback: It considers only the neighbors of a node and has no global view on the structure of an instance. If a node has low ratio r compared to its neighbors, it does not necessarily imply

[4]This might raise the question why $c(u)$ is used as divisor instead of e.g. multiplier. However, this is necessary to ensure that the technique works consistently for different coarsening levels. If $c(u)$ would be used as multiplier, contracting two nodes would increase the ratio $r(u)$ of the resulting contracted node u by a factor of up to 4 in comparison to the two input nodes. Also, our fixed core partitioning algorithm orders the nodes based on $r(u)$ – thus we can view $r(u)$ as the "priority" used for assigning u.

that the ratio is low compared to other nodes of the graph. This problem can be avoided by using an approach that considers the incident weight ratios of all nodes. The idea is to consider the distribution of the ratio r and identify peripheral nodes as "outlier" with low value. However, defining the distribution with regards to the nodes does not work – since the majority of the nodes of a star-like graph are peripheral, they are not actually outliers. Fortunately, the problem can be solved by changing the perspective. While the peripheral *nodes* are no outliers, this is not true of their incident *edges*: The vast majority of edges in a star-like graph is incident to core nodes, making every edge that is incident to a peripheral node an outlier.

More precisely, we associate each node u to $\omega(I(u))$ observations of the value $r(u)^{-1} = \frac{c(u)}{\omega(I(u))}$.[5] For a graph $G = (V, E)$, the average than is $\mu(G) := \frac{1}{\omega(E)} \sum_{u \in V} \omega(I(u)) r(u)^{-1} = \frac{c(V)}{\omega(E)}$ (i.e., the inverse density) and the standard deviation is $\sigma(G) := \frac{1}{\omega(E)} \sum_{u \in V} \omega(I(u))(r(u)^{-1} - \mu(G))^2$. Using the average and the standard deviation combined with a threshold parameter d, we consider a node peripheral if $r(u)^{-1} \geq \mu(G) + d \cdot \sigma(G)$. Note that these values are comparatively stable under contractions, i.e., the average value $\mu(G) = \frac{c(V)}{\omega(E)}$ changes only by the weight of the contracted edges.

The presented approach can be implemented as a preprocessing technique that runs before coarsening. We use parallel sums for the average and standard deviation. Then, we iterate over the nodes in parallel to mark all peripheral nodes, subsequently separating them from the core.

Detection based on Community Detection. The graph partitioning algorithm in which we integrate our work uses a community detection algorithm as a preprocessing step [27]. Then, in the coarsening phase contractions are restricted to the detected communities. The community detection algorithm minimizes the *modularity* objective function [54]. This metric prefers a community structure with sparse connections between communities and dense connections within communities compared to a random graph model (see Ref. [8] for more details). The algorithm uses the Louvain method [8, 32], which repeatedly maximizes the *local* modularity by greedily assigning nodes to the neighboring community with the highest increase in modularity.

Since the increase in modularity of a node (also called its *gain*) mostly depends on the total incident edge weight to the target community, we can use a similar technique as before. The basic idea here is that nodes which are only weakly connected to their respective communities are good candidates for peripheral nodes, e.g., if a node provides only a very small increase in modularity compared to others. To implement this, we modify the community detection algorithm: The first phase of the Louvain method is executed as usual, which greedily assigns the nodes to communities. For a given node u, we define $\delta(u)$ as the difference in modularity between placing the node in its current community and placing the node in its own community with size one. As before, we want to consider the value in relation to the node weight. Therefore, we define $r_\delta(u) := \frac{\delta(u)}{c(u)}$ as the ratio of modularity difference relative to its node weight.

Using this, we use a similar threshold as before, with average $\mu_\delta(G) := \frac{1}{\omega(E)} \sum_{u \in V} \omega(I(u)) r_\delta(u)^{-1}$ and standard deviation $\sigma_\delta(G) := \frac{1}{\omega(E)} \sum_{u \in V} \omega(I(u))(r_\delta(u)^{-1} - \mu_\delta(G))^2$. We iterate in parallel over all nodes, separating a node if $r_\delta(u)^{-1} \geq \mu_\delta(G) + d \cdot \sigma_\delta(G)$ for a factor d. To separate nodes, we can place them in their own community and afterwards separate all communities with size one. More formally, we define a maximum weight $\tilde{c}$ and, during the coarsening phase, separate a node u if its community has size one and $c(u) \leq \tilde{c}$. This means that we also separate small communities which are contracted into a single node during coarsening – which is probably a good idea, as small communities have only sparse connections to the remainder of the graph.

[5] Note that we use the inverse of $r(u)$ so that peripheral nodes become outliers with high value, which is easier to detect with basic methods than outliers towards zero.

There is also a number of variants that can be applied here. We can calculate the average ratio and standard deviation for each community separately instead of using the values defined on the complete graph. Further, instead of directly separating nodes that are below the threshold, we can allow a small number of contractions between adjacent peripheral nodes before they are separated. This might be useful as we want to minimize the amount of edges between peripheral nodes. Approaches to implement this could be contracting all such edges, or applying a community detection step which is restricted to the subgraph of peripheral nodes, allowing contractions within the detected communities.

Detection based on a Density Clustering Formulation. Generally, it is desirable that there are very few edges between peripheral nodes while the core has high density. This can be used to directly define an objective function for the detection of peripheral nodes: For a given graph $G = (V, E)$ we define the density of G as $D(G) := \frac{\omega(E)}{c(V)}$. The goal is then to minimize $\varphi(P) := \frac{D(G[P])}{D(G[V \setminus P])}$ where P are the peripheral nodes. Note that φ can be minimized by choosing P as an independent set.

However, choosing P as a independent set is not really useful in practice: As already mentioned earlier, there are usually a few edges between nodes that should be considered peripheral, even though the majority of the edge weight is in the core of the graph. Fortunately, we can circumvent this problem by using an initial set of nodes that is provided by another method, e.g., the previously explained statistical approach. Then, we use an algorithm for the clustering that is only allowed to add nodes but not to remove nodes. The result is that the clustering converges towards a local minimum of φ without requiring P to be an independent set. There are also possible variants for the definition of the objective function. For example, we can further encourage maximizing the size of P by using $\varphi'(P) := \frac{c(V \setminus P)}{c(P)} \varphi(P)$.

For the clustering, we sort the nodes in increasing order of their incident weight ratio r. Afterwards, we iterate over all nodes and add a node to P if this improves φ.

5.2. Coarsening

In traditional multilevel partitioning, the coarsening phase has multiple purposes. It should successively reduce the size of the input graph such that we can afford to use expensive initial partitioning algorithms on the coarsened graph. For this, the coarsest graph should be structurally similar to the input graph such that the initial partition can provide a good approximation of the optimal solution. Further, on coarser levels the refinement algorithm can explore the search space efficiently while improving the solution in detail at finer levels.

The situation is somewhat similar for star partitioning. After separating the peripheral nodes, we apply traditional coarsening algorithms to the core of the graph. However, it is also desirable to apply some kind of coarsening to the peripheral nodes for similar reasons: The peripheral nodes need to be integrated into both initial partitioning and refinement. While there are different possibilities for the exact strategy, most approaches require a coarsening of the peripheral nodes to ensure both an acceptable running time and more efficient search steps during initial partitioning and refinement.

However, traditional coarsening approaches based on clustering strongly-connected neighbors are not applicable to the peripheral nodes, as these are not or only via a few edges connected to other peripheral nodes. Instead, we apply *two-hop coarsening*: During a coarsening step, we do not require that nodes are adjacent. Instead, we cluster nodes with a similar neighborhood. This approach is already known to be useful for graphs with highly-skewed node degree distribution [47], which indicates that it is a good choice for coarsening peripheral nodes.

To preserve the structure of the peripheral nodes during two-hop coarsening, we prefer contractions between nodes with a large number of common neighbors. Further, as seen in Section 4, the ratio r of incident edge weight to node weight can be important when the peripheral nodes are assigned since the knapsack algorithm prioritizes nodes with high r ratios. For this reason, it is preferable to contract nodes with similar ratio r in addition to a similar neighborhood. In the following, we present a few techniques to implement this overall idea. While some of these techniques are already known from Ref. [47], to the best of our knowledge both the similarity matching and the node distance approach are novel in the context of a coarsening algorithm.

Degree One Nodes. Degree one nodes (which are connected to exactly one core node) are handled as follows: We contract nodes with degree one and the same neighbor. However, as mentioned we want to prefer contracting nodes with similar ratio of incident weight to node weight. To achieve this, we first collect all degree one nodes adjacent to a given core node and sort them by r. Then we cluster the nodes according to this order, grouping at most four nodes in the same cluster. We implement this by iterating in parallel over all core nodes.

Twins. We call nodes u and v *twins* if their neighbor sets are identical, i.e., $N(u) = N(v)$. Clearly, twins are good candidates for contractions. The detection of twins can be implemented in a similar fashion to the InrSrt algorithm of Aykanat et al. [5, 17] for detecting identical sets. We calculate *fingerprints* of the neighborhoods by applying a hash function to the neighborhood set – peripheral nodes with identical fingerprints are candidates for twins. Then, the nodes are sorted by their fingerprint and ratio r and we use pairwise comparisons to determine contraction partners. This is parallelized by distributing the peripheral nodes to different threads based on their fingerprint.

Similarity Matching. Nodes might have very similar neighborhoods without being twins. In the following, we present an approach that contracts nodes with similar neighborhoods that are not twins. To provide a formal metric for the similarity of the neighborhoods of two nodes, we can use the *Jaccard index* which measures the similarity of two sets. Given sets M and N, it is defined as $J(M, N) \coloneqq \frac{|M \cap N|}{|M \cup N|}$. Additionally, we want to include edge weights, which can be done by using the weighted Jaccard index for two peripheral nodes u and v:

$$J^*(u, v) \coloneqq \frac{\sum_{w \in N(u) \cap N(v)} \min\{\omega(u, w), \omega(v, w)\}}{\sum_{w \in N(u) \cup N(v)} \max\{\omega(u, w), \omega(v, w)\}}$$

Note that $J^*(u, v) \in [0, 1]$. Unfortunately, it is non-trivial to efficiently determine node pairs with a high weighted Jaccard index. As a first step, we therefore ignore the edge weights and try to group nodes based on their neighborhood sets. For this, we use a randomized approach based on locality-sensitive hashing. The idea of locality-sensitive hashing is to hash the nodes in such a way that the probability for nodes with a similar neighborhood to have equal hash values is high, while the probability is low for nodes with dissimilar neighborhoods. For the Jaccard index, the *min-hash* family of hash functions is known to be locality-sensitive [9] (see [49] for an introduction to min-hashing and locality-sensitive hashing): Given a hash function h and a set of nodes $U \subseteq V$, the min-hash is defined as $\overline{h}(U) \coloneqq \min_{u \in U} h(u)$. Using this, we choose a constant number b of hash functions $h_1, \dots, h_b$ uniformly at random. Our similarity-based coarsening algorithm then computes the hash function $h_S(u) \coloneqq (\overline{h_1}(N(u)), \dots, \overline{h_b}(N(u)))$ for each peripheral node $u \in P$. Given two nodes u and v, the probability that the hashes are equal is $\mathbb{P}(h_S(u) = h_S(v)) = J(N(u), N(v))^b$ due to the uniform distribution [49]. Nodes with similar

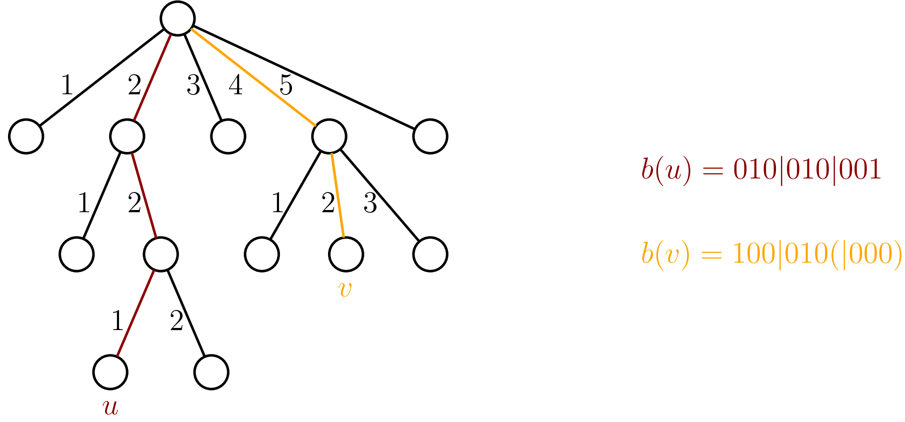


Figure 7: Example of encoding paths in the spanning tree by assigning an index to each child.

neighborhood are thus more likely to have an equal hash value and increasing the number b of min-hash functions decreases the probability of a hash collision exponentially.

To find contraction candidates, we then use the same approach as for twins with the fingerprint of a node being the value of the hash function h_S. In addition, we want to include the edge weights for determining the similarity. We therefore compute the weighted Jaccard index $J^*(u,v)$ for a constant number of nodes with the same fingerprint. A second node v is chosen as a match if the similarity $J^*(u,v)$ is above a predefined threshold. Note that we can only use pairwise contractions here (i.e., we compute a matching instead of a clustering) since the similarity value is not transitive. Therefore, we use multiple passes such that nodes that stay unmatched in the first pass can still be matched later. As before, we achieve parallelism by distributing the nodes to different threads based on their fingerprints.

Degree Two Nodes. There is one additional case where we use a specialized technique: nodes with degree two. Let $P_2 := \{u \in P \mid d(u) = 2\}$ be the set of peripheral nodes with degree two. For a given node $u \in P_2$ let $N(u) = \{n_1, n_2\} \subseteq C$ be the neighborhood set of u such that $\omega(u,n_1) \leq \omega(u,n_2)$. We define $n_{min}(u) := n_1$ as the neighbor of u where the incident edge has lower weight and $n_{max}(u) := n_2$ as the neighbor of u where the incident edge has higher weight. Further, for a core node $c \in C$ we define $P_2^{max}(c) := \{u \in P_2 \mid n_{max}(u) = c\}$ as the set of adjacent peripheral nodes with degree two and high incident edge weight.

For a given core node c, we want to find contractions within $P_2^{max}(c)$ – which means that $n_{max}(u) = n_{max}(v) = c$ for contraction candidates u and v (which is desirable since it implies $J^*(u,v) \geq \frac{1}{2}$). In addition, it could be useful to consider $n_{min}(u)$ and $n_{min}(v)$: For example, the contraction is preferable if $n_{min}(u)$ and $n_{min}(v)$ are adjacent and strongly connected compared to the case where $n_{min}(u)$ and $n_{min}(v)$ are not adjacent. To generalize this idea, we define a distance metric on the core nodes as follows: Let us assume that C is connected. We construct a *maximal spanning tree* $T = (C, E_T)$ on the subgraph induced by C (i.e., T is a tree with nodes C such that $\omega(E_T)$ is maximized) and define the distance $d(u,v)$ of $u, v \in C$ as the length of the (unweighted) path from u to v in T. Note that for constructing T, we can use a modified minimum spanning tree algorithm, e.g., we can use the Jarník-Prim algorithm with a max heap instead of a min heap for choosing the next edge.

The goal is to find contraction partners u and v in $P_2^{max}(c)$ where $d(n_{min}(u), n_{min}(v))$ is small. Implementing this efficiently is non-trivial: Comparing each pair in $P_2^{max}(c)$ is quadratic in $|P_2^{max}(c)|$, processing T (e.g., with a breadth-first search) for each $c \in C$ is quadratic in $|C|$. Instead, we calculate for each node $c \in C$ an encoding $b(c)$ of its position within T as a bitstring,

Stage	Applied techniques	Max. difference of ratio r
1	D1, Twins	1.4
2	D1, D2, Twins	1.8
3	D1, D2, Twins, Sim	2.1
4	D1, D2, Twins, Sim*	2.6
5	D1, D2, DX, Twins, Sim*	4.1
6	Any	∞

Table 2: Summary of the stages used for two-hop coarsening. Used techniques: degree one nodes (D1), degree two nodes (D2), twin matching (Twins), similarity matching (Sim), similarity matching with reduced threshold (Sim*), cluster nodes in $P^{max}(c)$ with degree ≥ 3 (DX), cluster any nodes in $P^{max}(c)$ (Any).

as exemplified in Figure 7.[6] We then sort $P_2^{max}(c)$ based on $b(n_{min}(u))$ for $u \in P_2^{max}(c)$. Nodes that are adjacent in T are likely to also be adjacent in this order. We compare each node to a constant number of predecessors and successors, contracting a pair if the distance in T is below a predefined threshold.

Here, we use a sequential implementation for the construction of the spanning tree, while iterating in parallel over the core nodes for the second step. Note that the impact of this technique is hard to measure and probably rather small. It likely works better if the core has very small size.[7]

Nodes with a Common Neighbor. If the previously described techniques do not provide a sufficient size reduction, we use a more coarse-grained approach. For a core node c, we define similar to before $P^{max}(c) := \{u \in P \mid \omega(u,c) = \max_{v \in N(u)} \omega(u,v)\}$ as the set of adjacent peripheral nodes with high incident weight. First, we contract any nodes within $P^{max}(c)$ with degree larger than two but we lift even the degree restriction if necessary. As before, the implementation uses a parallel iteration over the core nodes.

Putting It All Together. We have presented five techniques for two-hop coarsening that can effectively reduce the number of peripheral nodes and also try to preserve the structure of the original graph. However, it is preferable to first try approaches that are likely to preserve the structure to a higher degree, i.e., degree one nodes and twins. Only if these are not sufficient to reach the desired size reduction, we apply the remaining techniques.

To provide a reasonable progression from more fine-grained to more coarse-grained techniques, we use multiple stages for the two-hop coarsening algorithm. We execute the stages one after the other and each stage runs a predefined set of coarsening techniques with a maximum acceptable difference of the incident edge weight to node weight ratio r of contracted nodes. If we reach the target size for the current coarsening level, the algorithm terminates. Otherwise, we run the next stage. A summary of the stages is given in Table 2.

Implementation Details. Having established this two-hop coarsening technique, there are some higher-level concerns that need to be addressed with regards to integrating it into the

[6] To calculate an encoding that is guaranteed to fit in a single machine word, we restrict both the maximum degree and the maximum depth of T during construction. Then, we index the children of each node and construct the bitstring as the concatenation of these indices. However, it is likely that more efficient encodings are possible.

[7] Our experiments indicate that it finds an amount of contractions comparable to the other techniques if the core is already coarsened. But it is unclear whether this has a quality benefit compared to simpler approaches.

multilevel paradigm. First, the two-hop coarsening is defined on the core C of the graph, but it is not specified whether C is the core of the input graph or a coarsened approximation of the core. Using the coarsened representation can actually be preferable, as this reduces the node degrees of peripheral nodes and thus makes our two-hop coarsening techniques more effective.

Our implementation supports two different coarsening strategies: In variant (i), we first construct the complete contraction hierarchy for C using traditional coarsening techniques. Then, the coarsest level of the core is used as basis for the two-hop coarsening. We apply our two-hop coarsening algorithm and afterwards contract the peripheral nodes. This is repeated until a size appropriate for initial partitioning is reached, using a predefined size reduction factor for each step. Note that each step might include multiple different stages from Table 2. This strategy is efficient for constructing a coarsened graph that can be used in the initial partitioning. However, it is not useful for refinement: Refinement algorithms operate on each level of the contraction hierarchy, thus if the peripheral nodes are to be included we need for each level an according "level" of the peripheral nodes with comparable size. Therefore, strategy (ii) constructs a contraction hierarchy of peripheral nodes with the same number of levels as the hierarchy of the core. For this, we first reduce the size of the core by applying traditional coarsening techniques to C and subsequently, we run our two-hop coarsening algorithm on the peripheral nodes. Here, multiple two-hop coarsening steps could be required as the two-hop coarsening provides less size reduction per step than traditional coarsening. Afterwards, we contract the graph and proceed to the next level.[8] The result allows to construct a unified contraction hierarchy including both the core and the peripheral nodes. Then, we apply the traditional refinement techniques used by Mt-KaHyPar during uncontraction.

For both strategies, it is necessary to determine a target number of peripheral nodes for the two-hop coarsening. Our implementation uses the same predefined target size for the coarsest level of the core as in traditional coarsening (i.e, $160k$ as used by Mt-KaHyPar).[9] The target size for the peripheral nodes is then $p|C|$ where p is an input parameter and $|C|$ is the size of the coarsened core. This ensures that $|P| \in \mathcal{O}(|C|)$ which works well with our two-hop coarsening techniques. In case of strategy (ii), we extrapolate the target size to the contraction levels such that each level uses the same reduction factor.

5.3. Initial Partitioning

If the core is contracted to a sufficiently small size, we can assume the coarsened graph is similar to a star graph. Naturally, we would like to use our fixed core partitioning algorithm to compute a solution which provides a constant approximation guarantee. Unfortunately, the size of the coarsened graph is still too large to directly apply our proposed brute-force method (at least $160k$ nodes). We could further reduce the size of the coarsened graph, but it has already been shown that this negatively affects the solution quality [62, 63]. Instead, we modify the initial partitioning algorithm implemented in Mt-KaHyPar to make use of our theoretical results from Section 4.

The initial partitioning phase of Mt-KaHyPar uses multilevel recursive bipartitioning [27, 63]. Recursive bipartitioning first computes a bipartition of the input graph and then continues this process on both blocks recursively until the graph is partitioned into the desired number of blocks. This is done in multilevel fashion using a portfolio of different bipartitioning techniques to compute an initial solution. The coarsening algorithm contracts the graph down to a target

[8] While the traditional coarsening and the two-hop coarsening are applied in alternating order *conceptually*, note that our actual implementation first constructs the complete hierarchy for the core and afterwards coarsens the peripheral nodes, using the according level of the core as input for each step.

[9] It might be interesting for future work to explore whether using a smaller target size for $|C|$ makes sense.

size of 320 nodes, while the initial solution is refined using label propagation and FM refinement during uncoarsening.

Our goal is to combine this with fixed core partitioning to achieve improved quality for star-like graphs. To achieve this, we first partition the core nodes using traditional initial partitioning and then assign the peripheral nodes, using Algorithm 1 for fixed core partitioning. This technique works well if the total cost of cutting the peripheral nodes is negligible compared to the edge cut of the core. However, as we will see in the experimental evaluation, for many instances we do not achieve good results without also considering the peripheral nodes during the partitioning of the core. Thus we need a technique to include the peripheral nodes.

Partitioning of the Core. To compute an initial (k-way) partition of the core, we use a modified version of the multilevel recursive bipartitioning algorithm employed by Mt-KaHyPar. We include the peripheral nodes within the recursive bipartitioning by using the techniques for coarsening and refinement recursively, i.e., two-hop coarsening is applied until the graph is sufficiently small for bipartitioning. Note that it might make sense to use a separate parameter p' instead of p for the target size of peripheral nodes, i.e., the target size is $p'|C|$ (with $|C| = 320$). Then, we apply the portfolio of bipartitioning algorithms [27] on the coarsened graph that includes both the core and the peripheral nodes. Afterwards, we apply refinement that is aware of peripheral nodes which we discuss in the next section in more detail.

Computing the k-way Partition. After the core is partitioned, our initial partitioning algorithm discards the assignment of the peripheral nodes and uses Algorithm 1 to compute a new assignment. This substantially improves the quality, as shown in the experimental evaluation (see Figure 16). This is because – in contrast to the heuristic partitioning of the core – the fixed core partitioning algorithm considers all k blocks at once and provides a constant approximation guarantee.

Note that two-hop coarsening destroys some of the structure of the peripheral nodes and thus may worsen the achievable quality of the initial partition. However, Algorithm 1 can handle a large number of peripheral nodes efficiently. Therefore, we evaluate two different strategies that applies the fixed core partitioning algorithm either to peripheral nodes of the smallest graph or the input graph. If used on the input graph, we perform an additional two-hop coarsening pass in which we restrict contractions to nodes of the same block (similar to a V-cycle [67]). This hierarchy then replaces the hierarchy of our first two-hop coarsening pass.

Basically, there is a trade-off between quality and running time: We can either use coarsened peripheral nodes for the fixed core partitioning or apply two-hop coarsening twice to achieve better quality, as we will see in the experimental evaluation. A possibility for reducing the running time could be to apply the fixed core partitioning algorithm to the peripheral nodes of a partially coarsened graph (i.e., a level somewhere between the input graph and the smallest graph) – we leave this for future work.

Tracking the Effects of Peripheral Nodes. The previously described approach for computing an initial partition of the core directly includes the peripheral nodes into recursive bipartitioning. However, there is also an alternative approach: We can keep the peripheral nodes separated from the core and and modify the algorithms used for computing an initial bipartition such that they explicitly consider the effect of the peripheral nodes on the cut. The bipartitioning algorithms used in Mt-KaHyPar include greedy graph growing, graph growing via label propagation, an alternating breadth-first search and random assignment [27, 30]. The greedy graph growing and label propagation algorithms compute a *gain value* for moving a node u to

a particular block. Here, the gain value is the incident edge weight between u and the corresponding block, i.e., the gain for assigning u to block V_i is $\sum_{v \in N(u) \cap V_i} \omega(u,v)$. In each step, greedy graph growing then assigns the node with the highest gain to the corresponding block. Label propagation works in rounds, where one round moves each node to the block with higher gain. The partition is initialized with a small number of nodes that are chosen at random. Then, label propagation rounds are repeatedly applied until the partition converges.

Algorithm 4: Label Propagation Round with Tracking of Peripheral Nodes

Input: Graph $G = (C \cup P, E, c, \omega)$, maximum allowed block weight L_{max}, partial partition $\{C_1, C_2\}$, tracking data structure T

```
1  foreach u ∈ V in random order do
2      b_max ←⊥                                          // target block for u
3      δ_max ← 0
4      foreach i ∈ {1, 2} do
5          δ ← Σ_{v∈N(u)∩C_i} ω(u, v)
6          if i ≠ getCurrentBlock(u) then
7              c_old ← T.getPeripheralNodesCut()
8              T.updateAssignment(u, i)
9              c_new ← T.getPeripheralNodesCut()       // get cut with u assigned to new block
10             T.updateAssignment(u, getCurrentBlock(u))   // restore previous state
11             δ ← δ + c_old − c_new                    // update gain value
12         if δ > δ_max then
13             b_max ← i
14     if b_max ≠ getCurrentBlock(u) and c(C_{b_max}) + c(u) ≤ L_max then
15         moveToBlock(u, b_max)
16         T.updateAssignment(u, b_max)
```

Our idea is to apply these algorithms only to the core nodes, but in a modified version. Algorithm 4 shows how this works for a label propagation round: For each step, we consider an assignment of the peripheral nodes to the blocks of the partition that is based on the core nodes which are already assigned. This is represented with a data structure that stores the current assignment (denoted by T). When the gain is calculated (Line 5), we additionally calculate how moving a node would affect the assignment of the peripheral nodes and thus the cut, using the result to update the gain value of the according move (Lines 6-11). For example, this might reduce the gain value for adding a core node to a block that already contains many core nodes, because less peripheral nodes can be assigned to this block afterwards. To implement this, the data structure needs to support both querying the current cut with regards to the peripheral nodes and updating the assignment of a core node. After the gains are calculated, the node is assigned to the block with the highest gain and we update T accordingly (Lines 14-16).

However, the running time overhead to compute a new assignment of all peripheral nodes for every step of the algorithm would be problematic in practice. Instead, we propose to implement the data structure using incremental updates as follows:[10] Given a partial partition $\Phi = \{C_1, C_2\}$ of C (note that some nodes might not be assigned yet), we define a corresponding partial partition $\{P_1, P_2\}$ of P by assigning $p \in P$ to P_1 if $\sum_{c \in C_1} \omega(p,c) \geq \frac{\omega(I(p))}{2}$, to P_2 if $\sum_{c \in C_2} \omega(p,c) \geq \frac{\omega(I(p))}{2}$ and to none of the blocks otherwise. Both P_1 and P_2 are represented as a

[10] For the sake of clarity, the described variant is simplified and stores the data differently than in our actual implementation.

balanced binary tree. For P_1, we use the ratio $r_1(p) \coloneqq \frac{1}{c(p)} \left(\sum_{c \in C_1} \omega(p,c) - \sum_{c \in C_2} \omega(p,c)\right)$ as key. The key $r_2(p)$ used for P_2 is defined analogous. For a given node p, we store the node weight as well as $\sum_{c \in C_1} \omega(p,c)$ and $\sum_{c \in C_2} \omega(p,c)$ in the binary tree. In addition, we store the sum of each of these values for the sub-tree with root p. To query an actual assignment (and corresponding cut) of peripheral nodes that respects the balance constraint for the maximum allowed block weight L_{max}, we select for the first block a subset $\overline{P_1}$ of P_1 with $c(C_1)+c(\overline{P_1}) \leq L_{max}$ by greedily adding the node with the highest value of r_1 until L_{max} is reached. $\overline{P_2}$ is constructed analogous (note that either $\overline{P_1} = P_1$ or $\overline{P_2} = P_2$). The resulting assignment is $\Phi_P = \{\overline{P_1} \cup P_2 \setminus \overline{P_2}, \overline{P_2} \cup P_1 \setminus \overline{P_1}\}$ and the cut of Φ_P is consequently $\sum_{p \in \overline{P_1} \cup P_2 \setminus \overline{P_2}} \sum_{c \in C_2} \omega(p,c) + \sum_{p \in \overline{P_2} \cup P_1 \setminus \overline{P_1}} \sum_{c \in C_1} \omega(p,c)$. To compute this, we search in the tree representing P_1 for the maximum position i where the total weight of the nodes left from i does not exceed $L_{max} - c(C_1)$: The set of these nodes is $\overline{P_1}$. Using the stored node weight of the sub-trees this works in $\mathcal{O}(\log |P|)$ time. To calculate the cut, we then sum the stored incident edge weights of the sub-trees on the path to i in $\mathcal{O}(\log |P|)$ time. Note that this assignment of peripheral nodes corresponds to using Algorithm 1, but with assigning the nodes greedily for each block instead of using a knapsack algorithm with an approximation guarantee. When the assignment of a core node c is updated, we need to update P_1 and P_2. For this, we iterate over the peripheral nodes that are adjacent to c and update them, which might involve removing the node from the binary tree and inserting it at a new position in P_1 or P_2. This works in $\mathcal{O}(|N(c) \cap P| \log |P|)$ time.

Note that this technique could also be used for refinement, i.e., we apply the traditional refinement algorithms only to the core of the graph but adjust the involved gains using the peripheral nodes and the presented data structure. Unfortunately, the running time overhead of this method is much more significant during refinement than during initial partitioning due to the larger size of the graph. Also, generalizing this technique to work in the k-way case is more complicated than in the two-way case. For these reasons, we implemented this technique only for the two-way refinement that is used in the recursive bipartitioning but not for k-way refinement.

5.4. An Efficient Data Structure for Separated Nodes

To enable processing both the core of the graph and the separated nodes in an efficient way, we use a specialized data structure for representing the peripheral nodes. We want to note that we could alternatively modify the existing graph data structure of Mt-KaHyPar to internally separate both types of nodes, which might be beneficial for performance reasons. However, using a separate data structure provides more flexibility for implementing and evaluating our different strategies.

Given a graph $G = (V, E)$ and separated nodes P, our data structure represents both the subgraph $G[P]$ of separated nodes and all edges between the core C and P, i.e., $E_{CP} \coloneqq \{\{c, p\} \in E \mid c \in C \land p \in P\}$ (while we use the established data structures of Mt-KaHyPar for $G[C]$). We use an adjacency array to represent the subgraph containing E_{CP}. Internal edges in $G[P]$ are represented as an edge list (i.e., an unsorted array containing all edges), since they are only accessed during reinsertion of the peripheral nodes in initial partitioning and refinement. To integrate this data structure into the shared-memory partitioning algorithm Mt-KaHyPar, the data structure needs to support the following operations (i) Insertion of a set of new nodes and edges to $G[P]$ and E_{CP}, (ii) updating the edges in E_{CP} with regards to a set of contractions of C and (iii) applying a set of contractions to P. Recall that for some of our peripheral nodes detection strategies the peripheral nodes are known before coarsening, while one strategy detects them during coarsening. For the latter strategy we need to support operation (i), while (ii) and (iii) are required for two-hop coarsening.

After identifying a set of peripheral nodes P, we need to update E_{CP} and the subgraph $G[P]$. To implement (i), we first compute the new IDs of the added nodes with a parallel prefix sum over an array of size $|V|$ that contains a one at every position u where u is a peripheral node. To obtain the position of the first incident edge of a node u within the adjacency array representing E_{CP}, we compute a parallel prefix sum over an array of size $|P|$ that contains at position u (using the new ID) the degree of u within $G[C]$, i.e., $|N(u) \cap C|$. Then, we resize the adjacency array accordingly and fill its entries by iterating over the peripheral nodes in parallel, using the previously computed prefix sum. To update the edge list with the internal edges of $G[P]$, we iterate over P in parallel and add edges that are incident to another peripheral node to the edge list (each threads accumulates the edges locally and then copies the result into the edge list).

After contracting the core, we have to (ii) update E_{CP} ($G[P]$ is not affected). For this, we iterate over all edges in E_{CP} in parallel and update the node IDs of the core nodes to their corresponding representative in the coarser representation of the core. However, E_{CP} now might contain duplicate edges. Thus, we apply a de-duplication step that accumulates the weights of all identical edges at one representative and removes all others. This is done by iterating over the peripheral nodes in parallel and sorting their adjacent nodes. The sorted edges then are de-duplicated by iterating over the sorted order and aggregating the weight of each identical edge at their first occurrence. Afterwards, we update the node degrees with a parallel prefix sum and shrink the adjacency array accordingly.

After computing a set of two-hop contractions between peripheral nodes, we (iii) need to apply these contraction to $G[P]$ and also update E_{CP} accordingly. To remap the node IDs, we compute a parallel prefix sum over an array of size $|P|$ that contains a one for every representative of a cluster. Then we iterate in parallel over the peripheral nodes and accumulate the node weights and node degrees with atomic instructions. The updated nodes are copied into a new instance of the adjacency array and we compute the position of the first incident edge for each node with a parallel prefix sum over the node degrees. Then the edges are copied to the new instance according to the computed positions. As this might result in duplicate edges, we apply the same edge de-duplication step to E_{CP} as in (ii). To de-duplicate the internal edges of $G[P]$ in the edge-list we use a parallel sort on the edges, iterate over the sorted order and aggregate the weight of identical edges at their first occurrence.

Further, the data structure supports extraction of a subset P_1 of P, which is required for recursive bipartitioning. Similar to before, we compute the new node IDs with a parallel prefix sum over an array that contains a one for each node in P_1. In addition, this involves extracting a subset of the core, which can be implemented in a similar fashion to contracting the core (but does not require de-duplication of edges).

5.5. Generalization to Hypergraphs

While this thesis focuses on graph partitioning, it might be interesting for future work to generalize the developed techniques for hypergraphs. Therefore, we provide a short overview of the associated challenges.

The techniques for detecting peripheral nodes are mostly based on the incident edge weight, i.e., nodes with low incident weight are considered peripheral. This could be applied in the same way to hypergraph. However, large hyperedges should probably be rated differently so that the total contribution of the hyperedge to the weight is similar to that of a smaller hyperedge. For a given node u, this could be achieved by using $\sum_{e \in I(u)} \frac{\omega(e)}{|e|}$ or $\sum_{e \in I(u)} \frac{\omega(e)}{|e|-1}$ instead of directly summing the weights (the latter formula corresponds to the heavy-edge rating function that is used for the coarsening of hypergraphs [42, 63]).

The techniques for two-hop coarsening from Section 5.2 are only partially applicable for hypergraphs: The generalization of similarity matching is rather straightforward, even though the same considerations regarding the weight of hyperedges as before are required. While twin matching could be applied, it is probably less effective in the case of hypergraphs: Neighborhoods will likely not be exactly equal as soon as large hyperedges are involved. A possible approach to improve this is ignoring hyperedges above a certain size threshold. The technique for degree two nodes is unlikely to be effective for hypergraphs, as it requires that a peripheral node is adjacent to exactly two core nodes. Finally, clustering degree one nodes is very useful for graphs but harder to apply for hypergraphs, as it requires that a given peripheral node is adjacent to only a single core node. A possible solution could be to handle a node as if it has degree one if the majority of its incident edge weight is connected to a single core node.

With regards to initial partitioning, we use a fixed core partitioning algorithm to compute an assignment of the peripheral nodes with a constant approximation guarantee. As we already assume that the peripheral nodes are an independent set, it seems reasonable to use the same requirement for a generalization to hypergraphs: Each hyperedge may be connected to at most one peripheral node. With this assumption, we can reduce the problem of assigning peripheral nodes for hypergraphs to fixed core partitioning. In the case of the $(\lambda - 1)$-metric, a hyperedge e that is incident to a peripheral hypernode p can be modeled with an edge from p to every block of the core that is incident to e. With the cut-net metric, a hyperedge can be removed if it is already cut by the partition of the core and corresponds to a single edge otherwise. The former case corresponds to cut-net splitting and the latter to cut-net removal, which are techniques known from recursive bipartitioning of hypergraphs [13, 62]. In conclusion, fixed core partitioning algorithms should be applicable for hypergraphs in the same way as for graphs.

Regarding bipartitioning and refinement, the approach of reinserting the peripheral nodes is also applicable for hypergraphs. However, tracking the effect of peripheral nodes would be more complicated than in the case of graphs and would require a generalization of the involved data structures.

Benchmark Set	SNAP	DAC	Walshaw	Recomp	Random	Other
Set A	6	2	4	2	5	6
Set B	29	4	2	7	18	20
Total	31	6	6	9	23	24
Star-like	6	0	0	9	8	0

Table 3: The number of graphs included for the different graph types and the number of star-like graphs for each type.

6. Experimental Results

In this section, we evaluate the different configurations and parameter choices of our star partitioning algorithm. First, we identify a set of well-performing configurations and analyze their effectiveness with two different approaches: We compare the configuration with the best overall performance to the state-of-the-art partitioning algorithm Mt-KaHyPar. In addition, we present an *algorithm portfolio* that contains both our new configurations and the baseline algorithm of Mt-KaHyPar and is capable of providing even better quality. To evaluate the portfolio, we generalized the effectiveness tests presented by Ahkremtsev et al. [2] to portfolio algorithms.

6.1. Setup and Methodology

We integrate our algorithms into the Multi-Threaded Kalsruhe Hypergraph Partitioning framework Mt-KaHyPar [27, 30]. Mt-KaHyPar was originally developed for partitioning hypergraphs, but implements optimized data structures for graph partitioning.

Mt-KaHyPar provides multiple partitioning configurations: Mt-KaHyPar-D (-**D**efault) uses multilevel partitioning with a logarithmic number of levels [27]. Mt-KaHyPar-Q (-**Q**uality) achieves better quality at the cost of longer running times using the n-Level approach, which is the most extreme version of the multilevel scheme: Here, only a single node is contracted on each level, thereby allowing more opportunities for refinement [28]. In addition, Mt-KaHyPar extends these configurations with flow-based refinement techniques to improve the quality even further [25] (Mt-KaHyPar-D-F and Mt-KaHyPar-Q-F, -**F**lows). Further, Mt-KaHyPar-SDet (-**S**peed **Det**erministic) is a multilevel configuration with deterministic results [24]. We refer to the dissertation of Heuer [30] for a detailed description of the configurations.

Methodology. For the evaluation, we use parameters $\varepsilon = 0.03$ and $k \in \{2, 4, 8, 16, 32\}$ with a time limit of one hour where not stated otherwise. For each instance (graph and number of blocks k) we perform ten repetitions using different random seeds. We aggregate the edge cuts and the running times of the different runs using the arithmetic mean. To aggregate running times over multiple instances we use the geometric mean. If a run exceeded the time limit, we use the time limit in the aggregate and mark it with ✗ in the plot. If all runs for a given instance exceeded the time limit or could not produce a feasible partition for other reasons, the instance is similarly marked with ✗ in the plot.

Instances. For our evaluation we use instances from different application areas as well as a set of graphs that are generated from random models. An overview is given in Table 3. We include instances from the Walshaw Benchmark for Graph Partitioning [65], the DAC 2012 Routability-Driven Benchmark Suite (DAC) [66], the **S**tanford **N**etwork **A**nalysis **P**roject (SNAP) [48]

and instances that are derived from scientific collaboration networks [55]. From a benchmark set for finite element computations we include graphs that are generated from dynamic two-dimensional numerical simulations [51] and from the 2018 Static Graph Challenge [59] we use protein k-mer graphs where nodes represent segments of amino acids. Further, we use graphs from the following random models: Random geometric graphs (rggX), where nodes are random points on the unit square with connection based on the Euclidean distance [33]. Similarly, Delaunay graphs (delaunayX) represent a Delaunay triangulation of random points in the unit square [33]. Random hyperbolic graphs (rhgX) are another variant of random geometric graphs generated by placing random points within a disk in the hyperbolic plane [22]. The Kronecker graphs (kronX) are derived from an R-MAT generator that works by sampling from a perturbed Kronecker product [7]. Based on a similar generator, the R-MAT graphs (rmatX) are created using the PaRMAT generator as described in [43]. Finally, Erdös-Rényi graphs (erX) are generated by assigning a uniform probability for inserting the corresponding edge to each pair of nodes. In this case, the probability is chosen such that the expected number of edges is logarithmic in the number of nodes. In addition, we include a set of graphs that are generated by a text compression tool based on the recompression technique [36] (recompX). The texts for the graphs are from the Pizza&Chili corpus [57]. Note that while all other benchmark instances are unweighted, the recompX graphs have weighted edges.

The benchmark set contains 99 instances in total. These instances include 23 star-like graphs which we identified in preliminary experiments. On these instances, we expect significant improvements by applying our star partitioning techniques. The remaining graphs are chosen as a representative sample from the application domains and random graph models described above. We use a subset consisting of 25 relatively small graphs for parameter tuning experiments, denoted by benchmark set A. In this set, 7 out of the 25 instances have a star-like structure. Afterwards, we evaluate our final configurations on benchmark set B, which consists of 80 graphs of varying sizes (from a hundred thousand edges up to 1.2 billion edges for the Twitter graph) and includes 19 star-like graphs. The two benchmark sets are mostly disjoint except for six instances – we reuse the star-like social graphs for benchmark set B because only few such instances are available.

Table 3 provides an overview over the composition of the benchmark sets with regards to the different graph types. A complete list of the instances in benchmark set A and benchmark set B is given in Appendix B.

System. The experiments are performed on a machine that uses an AMD Epyc 7702P processor with 64 cores, a clock frequency of 1.50 GHz and 256 MiB L3-Cache. The machine runs Ubuntu 20.04.2 LTS Linux and has 996 GB main memory. We run up to 8 processes in parallel (provided the main memory is not exceeded) and use 8 threads per process. This setup causes an overall increase in the running time measurements. However, this is reasonable since the same setting is used for all algorithms and performance engineering is not the goal of this work. Our implementation is written in C++ and compiled with g++-9.4 using the flags `-O3 -mtune=native` and `-march=native`.

Performance Profiles. We use performance profile plots [18] to compare the solution quality of different algorithms. In the plot, each algorithm is represented by a performance curve. Let $\mathcal{A}$ be the set of all algorithms we want to compare, $\mathcal{I}$ the set of instances, and $q_A(I)$ the quality of algorithm $A \in \mathcal{A}$ on instances $I \in \mathcal{I}$. For each algorithm A, we plot the fraction of instances (y-axis) for which $q_A(I) \leq \tau \cdot \text{Best}(I)$, where τ is on the x-axis and $\text{Best}(I)$ is the best solution produced by any algorithm. For example, if the value of A on the y-axis is 0.6 for $\tau = 1.2$,

Name	Applicable values	Default
Peripheral nodes detection strategy	`local`, `stats`, `comm`, `cluster`	-
	Parameters for comparing adjacent nodes (`local`)	
Max. ratio difference s	$[1, \infty)$	3
Separated incident edge weight fraction t	$[0, 1]$	1
	Parameters for statistical threshold and community detection (`stats`, `comm`)	
Standard deviation factor d	$\mathbb{R}$	2
Contractions of p. nodes (only `comm`)	`none`, `all`, `sub_communities`	`none`
	Parameters for density based clustering (`cluster`)	
Objective function	φ, φ'	φ
Std. dev. factor d for initial solution	$\mathbb{R}$	4
	Shared parameters	
Coarsening target size p	$[1, \infty)$	2
Coarsening target size p' for IP	$[1, \infty)$	2
Strategy for fixed core partitioning after IP	`none`, `coarse`, `input`	`input`
Fixed core partitioning algorithm	`greedy`, `approximate`	`approximate`
Initial partitioning strategy	`reinsert`, `tracking`, `core`	`reinsert`

Table 4: Overview of available strategies and parameters. A discussion of the strategies is given in Section 5. Note: IP is a shorthand for initial partitioning.

then the cut of the partitions computed by this algorithm is at most 20% ($\tau = 1.2$) worse than the best solution found on 60% ($y = 0.6$) of the instances. For $\tau = 1$, the y-value indicates the percentage of instances for which an algorithm $A \in \mathcal{A}$ performs best. Note that we separate the x-axis into three areas with different scale: The first area display instances which are close (within 10%) to the quality of the best result. In the second area, instances are shown where the quality is within 2 times of the best result. The third area uses a logarithmic scale to display the remaining instances. Any remaining instances are either timeouts or the algorithm could not find a feasible partition for other reasons. These infeasible instances are marked with a ✗-tick. Note that these plots relate the quality of an algorithm to the best solution and thus do not permit a full ranking of three or more algorithms.

6.2. Parameter Tuning

To evaluate the strategies developed in Section 5 and choose configurations with effective parameters, we perform parameter tuning experiments on benchmark set A. The primary goal of this study is to find a configuration computing partitions with the highest possible quality on star-like graphs without negatively affecting the solution quality on the remaining instances. However, this turned out to be difficult as we will see throughout this evaluation. Therefore, we propose two configurations: one that works well for all instances (referred to as *balanced* configuration) and one that achieves the highest possible quality for star-like graphs (referred to as *optimized* configuration). An overview of available strategies and parameters is given in Table 4. The provided default values are based on preliminary experiments. In the following experiments, we evaluate different choices for one or two parameters, while all others are set to their default values.

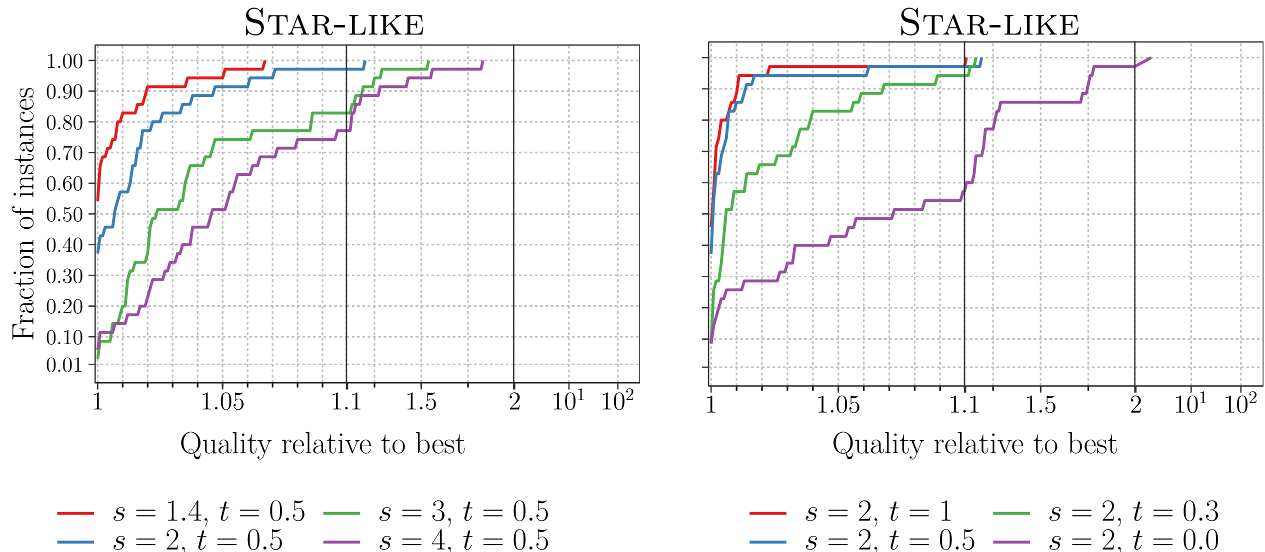


Figure 8: Comparison of parameters for peripheral nodes detection based on comparing adjacent nodes (`local`).

6.2.1. Detection of Peripheral Nodes

Dynamic Detection in the Coarsening Phase. The `local` strategy separates peripheral nodes in the coarsening phase. It compares the ratio $r(u) := \frac{\omega(I(u))}{c(u)}$ of adjacent nodes and considers a node u peripheral if $s \cdot r(u) \leq r(v)$ for all its neighbors v and a tuning parameter s. Also, u can be adjacent to nodes that were separated on a previous level of the multilevel hierarchy. We separate u only if $\omega(I'(u)) \leq t \cdot \omega(I(u))$, where $I'(u)$ is the set of incident edges restricted to already separated nodes and t is a second parameter. Figure 8 provides an overview over the effect of both parameters on star-like instances. As it can be seen, smaller values for s and larger values for t lead to better results on star-like instances. Using $s = 2$ instead of $s = 3$ improves the quality by 1.5% in the median while $t \in \{0.5, 1\}$ provides significantly better results than smaller values of t. However, this tends to decrease the quality for instances that are not star-like.

Figure 9 compares promising candidates for a balanced and an optimized configuration. The best results for all instances are achieved with $s \in \{3, 4\}$ and $t = 0.3$. The difference between $s = 3$ and $s = 4$ is not significant on all instances, but we can see an improvement by 0.8% in the median on star-like graphs when we use $s = 3$ instead of $s = 4$. Therefore, we use the values $(s = 3, t = 0.3)$ for the Balanced_{local} configuration.

The candidates for an optimized configuration are $s \in \{1.3, 1.6\}$ and $t \in \{0.5, 1\}$. All tested configurations produce similar solutions on star-like graphs, as shown in Figure 9 (left). However, $(s = 1.3, t = 0.5)$ and $(s = 1.6, t = 1)$ produce on 12% of all instances solutions that are more than a factor of two worse than the best solution, as it can be seen in Figure 9 (right). We therefore choose $(s = 1.6, t = 0.5)$ for the Optimized_{local} configuration.

Outlier Detection of Statistical Properties. The `stats` strategy considers the distribution of the ratio r over the complete graph and detects peripheral nodes as outliers with small value. The threshold for outliers is $\mu + d \cdot \sigma$, where μ is the average and σ the standard deviation of the r values, and d a tuning parameter. Figure 10 compares different values for d. We see in the plots that small values for d lead to better results on star-like instances but worse results on all instances. The configurations with $d \in \{2, 3, 4\}$ achieve the best quality on all instances.

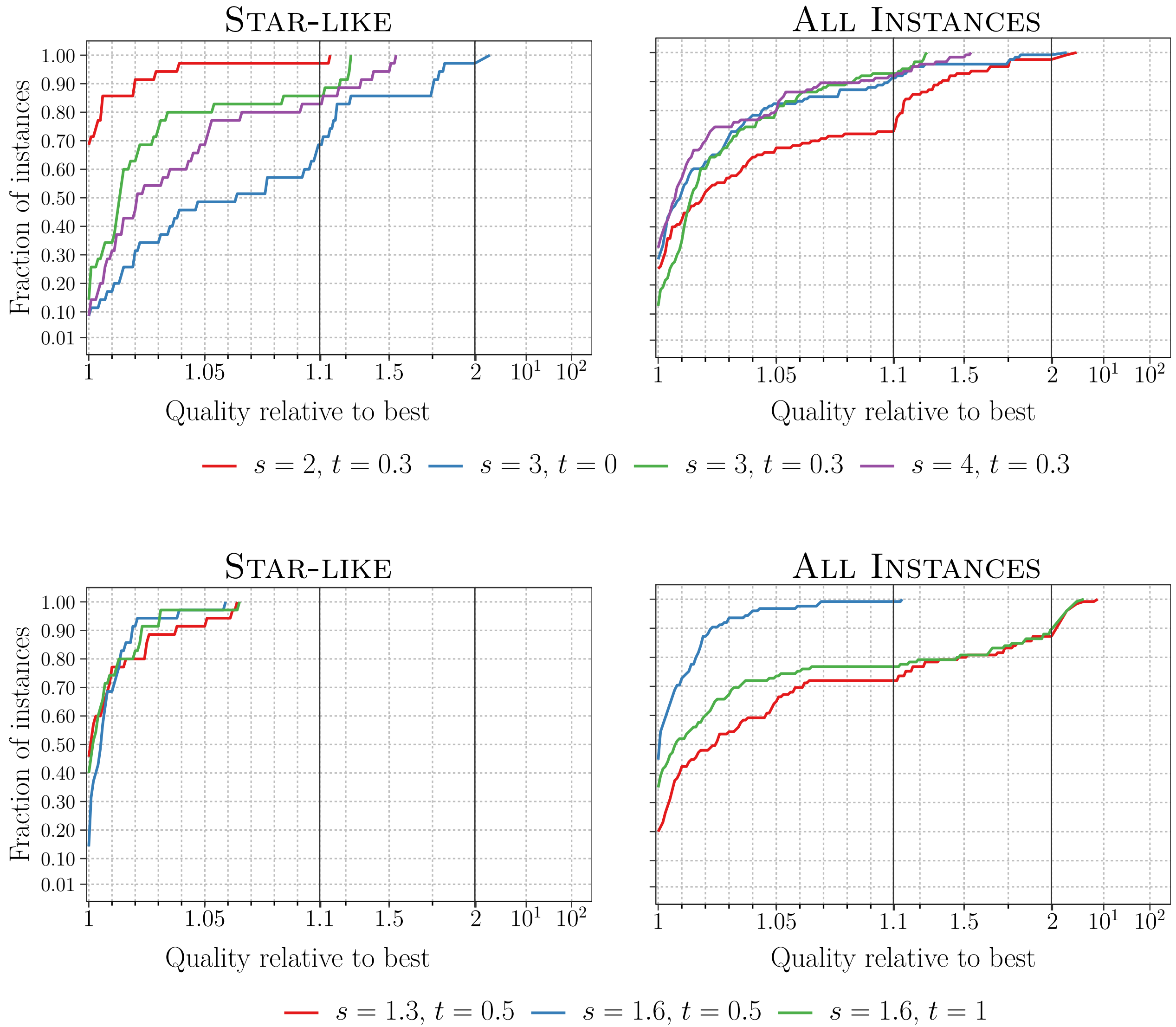


Figure 9: Comparison of parameters for peripheral nodes detection based on comparing adjacent nodes (`local`). We evaluate balanced configurations (top) and optimized configurations (bottom).

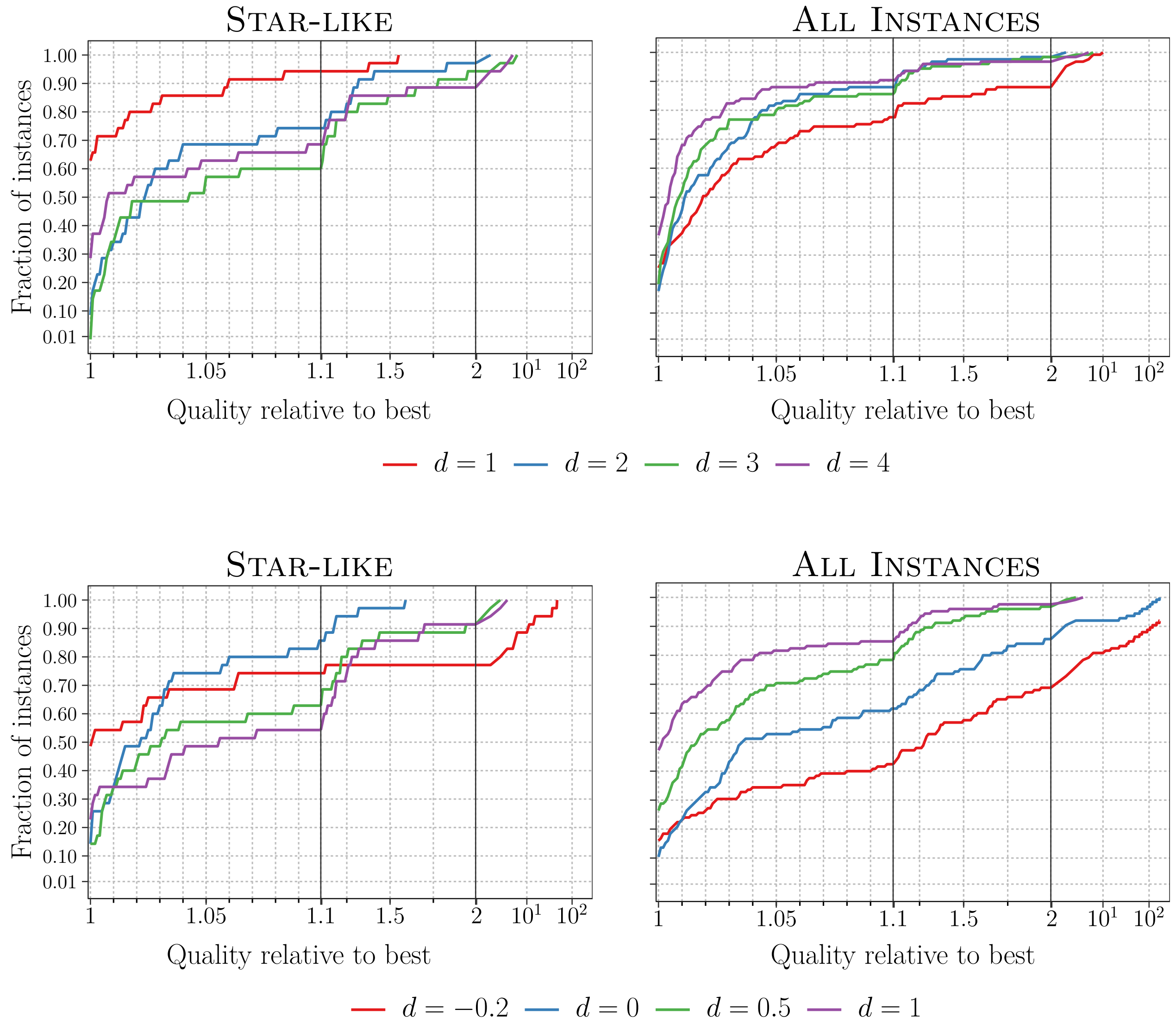


Figure 10: Comparison of parameters for peripheral nodes detection based on a statistical threshold (`stats`). We evaluate balanced configurations (top) and optimized configurations (bottom).

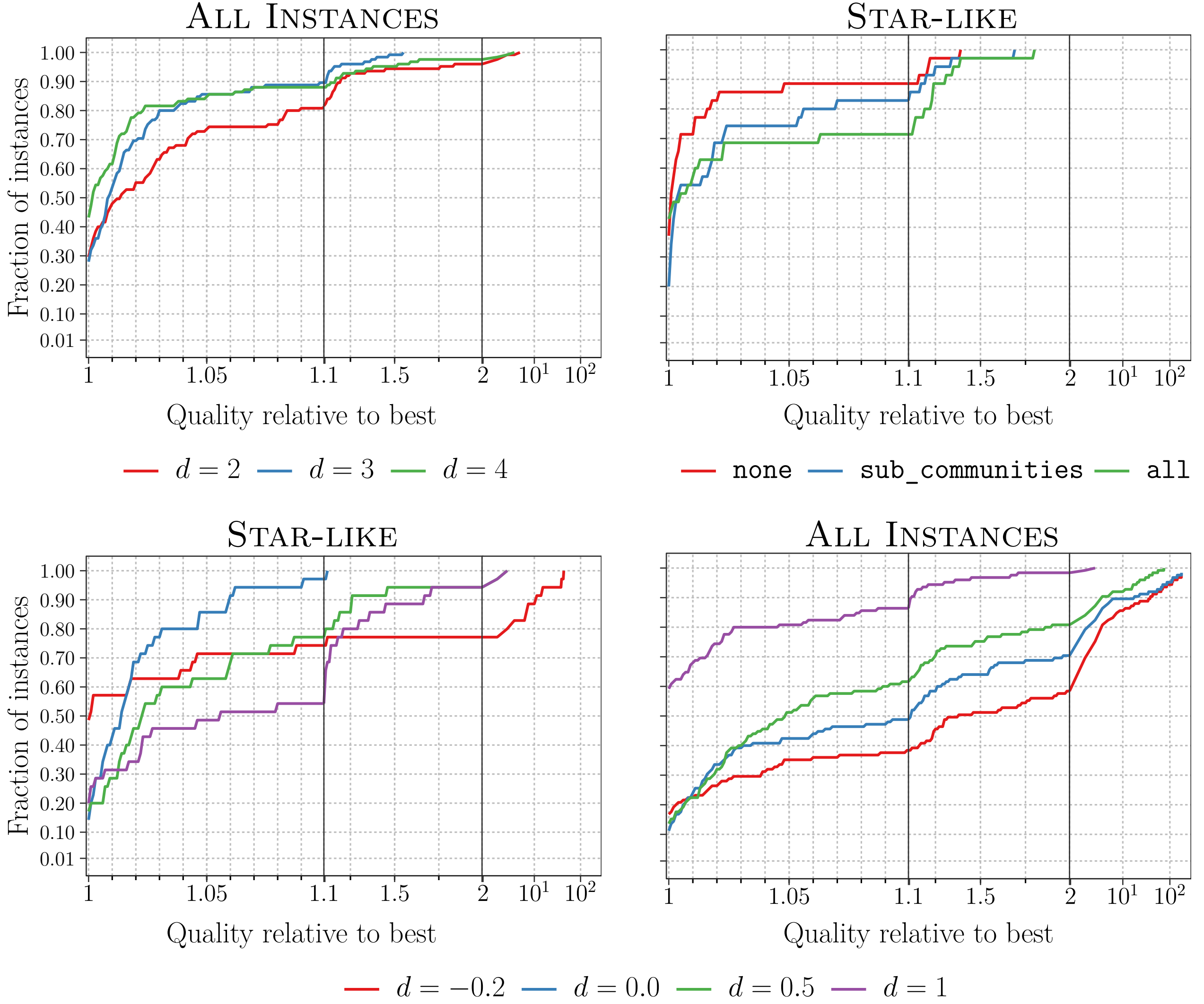


Figure 11: Comparison of parameters for peripheral nodes detection based on a community detection (`comm`). We evaluate balanced configurations (top-left) and different contraction strategies for edges of peripheral nodes with $d = 3$ (top-right), as well as optimized configurations (bottom).

The differences are small, with a deviation of less than 2% in the median quality compared to the best solution. We choose $d = 4$ for the Balanced$_{stats}$ configuration since it finds the best solution for 40% of instances and also has slightly better running time on average (0.48s compared to 0.5s for $d = 3$ and 0.54s for $d = 2$).

Further, Figure 10 (left) shows that the quality for star-like graphs continues to improve down to surprisingly small values for d. The best quality is achieved for $d = 0$ (note: $d = 0$ means that anything below average is considered an outlier). While $d = -0.2$ finds the best partition for more instances than the other configurations (50% of instances), the results are also two times worse than the best solution for ≈ 25% of the instances. Therefore, we choose $d = 0$ for the Optimized$_{stats}$ configuration.

Community Detection. The `comm` strategy is based on community detection. It uses a similar statistical threshold as the `stats` strategy, but the distribution is based on the modularity gain δ instead of r. As before, we use $\mu + d \cdot \sigma$ with a parameter d as threshold. Figure 11 shows

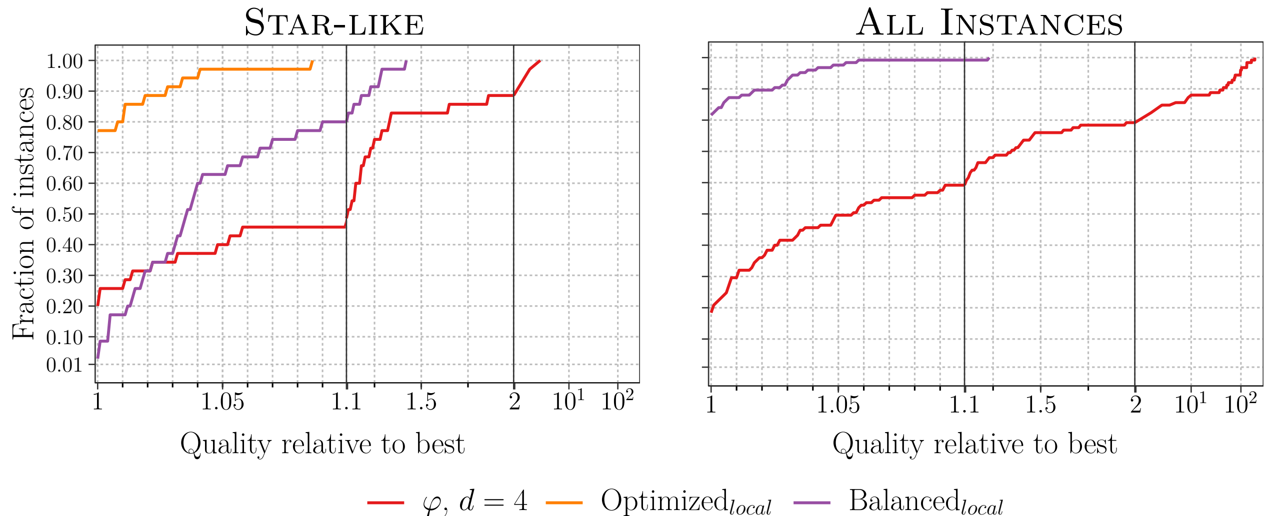


Figure 12: Comparison of the `cluster` strategy to the `local` strategy.

that small values for d provide better results for star-like and worse results for other instances, similar to the `stats` strategy. On all instances, $d = 3$ and $d = 4$ achieve the best quality and both configurations compute similar solutions. For a few instances, the solution found with $d = 4$ is two times worse than the best solution, while the configurations have similar running times. We therefore use $d = 3$ for the Balanced$_{comm}$ configuration.

On star-like instances, $d = 0$ achieves the best quality: The results are within 10% of the best solution on all instances. Similar to the `stats` strategy, $d = -0.2$ finds the best partition for more instances than $d = 0$ but is two times worse than the best solution for more than 20% of instances. Thus, we choose $d = 0$ for the Optimized$_{comm}$ configuration.

Further, Figure 11 (top-right) compares different strategies for contracting peripheral nodes: The `none` strategy separates the nodes without applying contractions. With `sub_communities`, community detection is applied to the peripheral nodes and the resulting communities are contracted. The `all` strategy contracts all edges between peripheral nodes before separating them. However, we see in the plot that both the `sub_communities` and `all` strategies worsen the quality for star-like instances. The results for the remaining instances are nearly identical to the `none` strategy. Therefore, we use the `none` strategy.

Density Clustering. The last peripheral nodes detection strategy (`cluster`) uses the density-based clustering algorithm. The strategy takes a partition of the node set into peripheral and core nodes computed via the `stats` strategy as input and then improves it by optimizing our proposed objective function. The parameter d of the `stats` strategy is used for calculating the input partition. For the clustering, we tested two variants of our objective function (φ and φ', see Section 5.1). However, Figure 12 shows that the results of this strategy are significantly inferior to the `local` strategy, on star-like instances as well as on the overall benchmark set. This holds with both objective functions and $d \in \{2, 4\}$ and also in comparison to the `stats` and `comm` strategies. Therefore, we do not use this strategy for our final configurations.

Comparison of Resulting Configurations. We compare the resulting configurations in Figure 13. Regarding the optimized configurations, the best quality on star-like instances is achieved by Optimized$_{local}$. The results produced by Optimized$_{local}$ are within 9% of the best

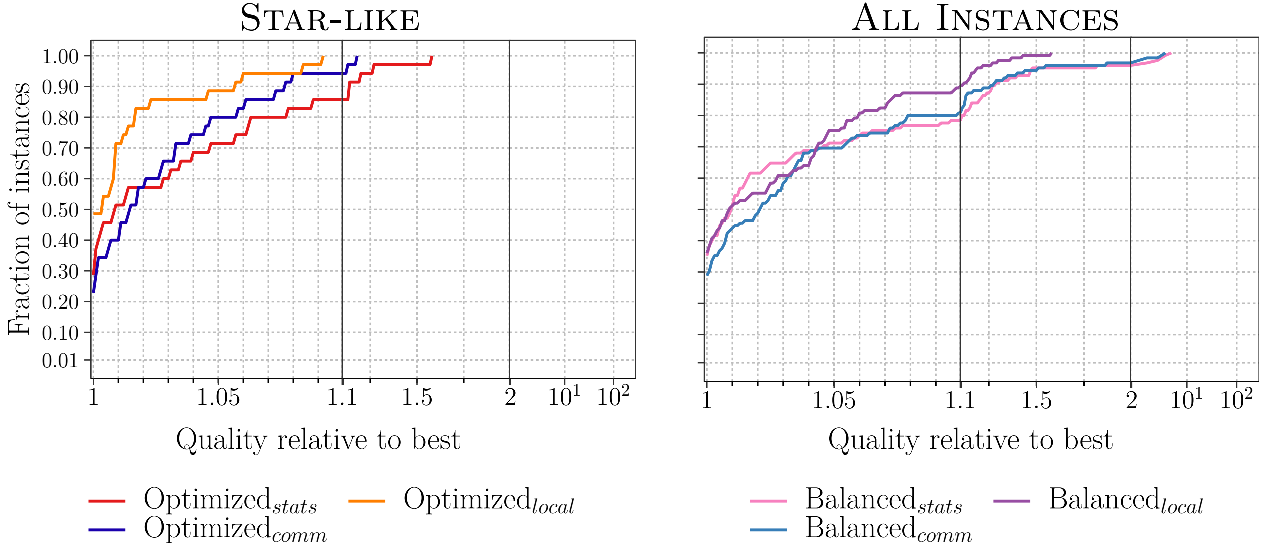


Figure 13: Comparison of optimized configurations on star-like instances (left) and balanced configurations on all instances (right).

solution for all instances. The Optimized_{comm} and Optimized_{stats} configurations compute comparable solutions. However, the results of Optimized_{comm} are within 20% of the best solution for all instances while Optimized_{stats} computes partitions with up to 50% worse cut than the best solution. On all instances, Balanced_{local} provides better results than the two other configurations, but only by a small margin (1% improvement in the median compared to Balanced_{comm}). Note that Balanced_{local} significantly outperforms the other balanced configurations on star-like instances, finding the best solution for 80% of instances. If we compare Balanced_{comm} and Balanced_{stats}, it is not obvious from the plot which produces better results and both also have similar running times (0.52s vs 0.48s). In summary, these results indicate that the `local` strategy works best for star partitioning.

6.2.2. Coarsening and Initial Partitioning

In the following, we consider the parameters and strategies that are applicable independent from the strategy for detecting peripheral nodes. While the detection of peripheral nodes tends to be a trade-off between quality on star-like instances and quality on the remaining instances, the shared parameters effect all instances similarly. The effects of the following parameters are similar for all peripheral node detection strategies. We therefore evaluate them using the Optimized_{local} strategy. However, we observed that the effects tend to be less pronounced for balanced strategies.

Two-hop Coarsening Target Size. The parameters p and p' determine the target size for two-hop coarsening, which is $p|C|$ peripheral nodes where C is the coarsest approximation of the core. For each bipartitioning step within the initial partitioning, we use p' instead of p. Figure 14 shows that the impact on the quality is minor. Higher values such as $p \geq 4$ lead to slightly worse results. The quality achieved with $p = 1$ and $p = 2$ is almost identical. However, $p = 1$ increases the running time by 9% on average. The reason might be that more coarsening passes are required to reach the target size. In addition, we tested choosing p and p' differently, but this did not yield any improvement. Therefore, we use $p = p' = 2$.

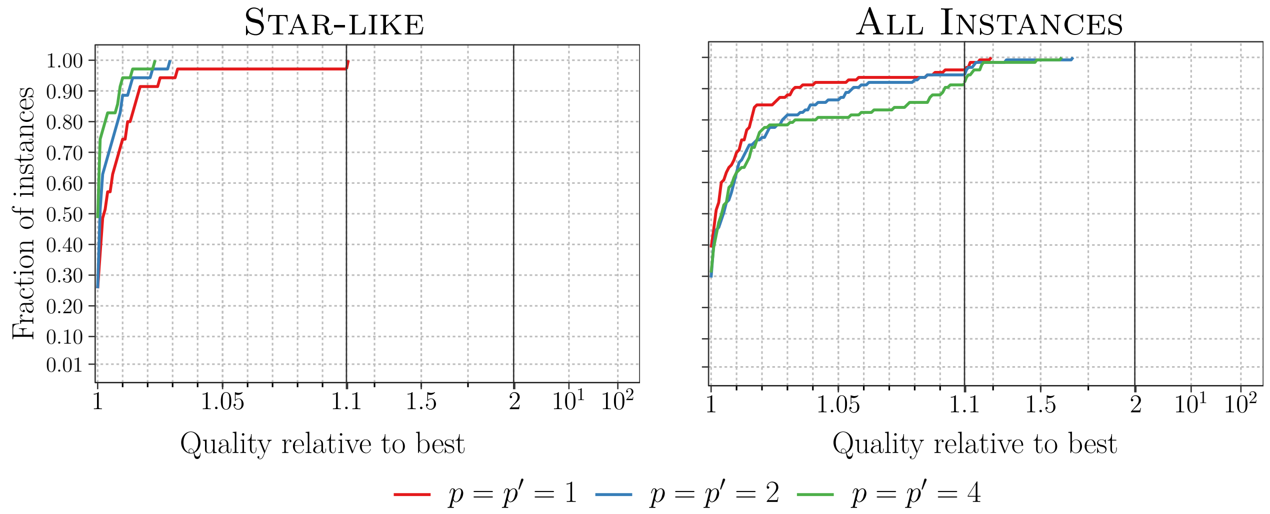


Figure 14: Comparison of different values for the two-hop coarsening target size (based on Optimized$_{local}$).

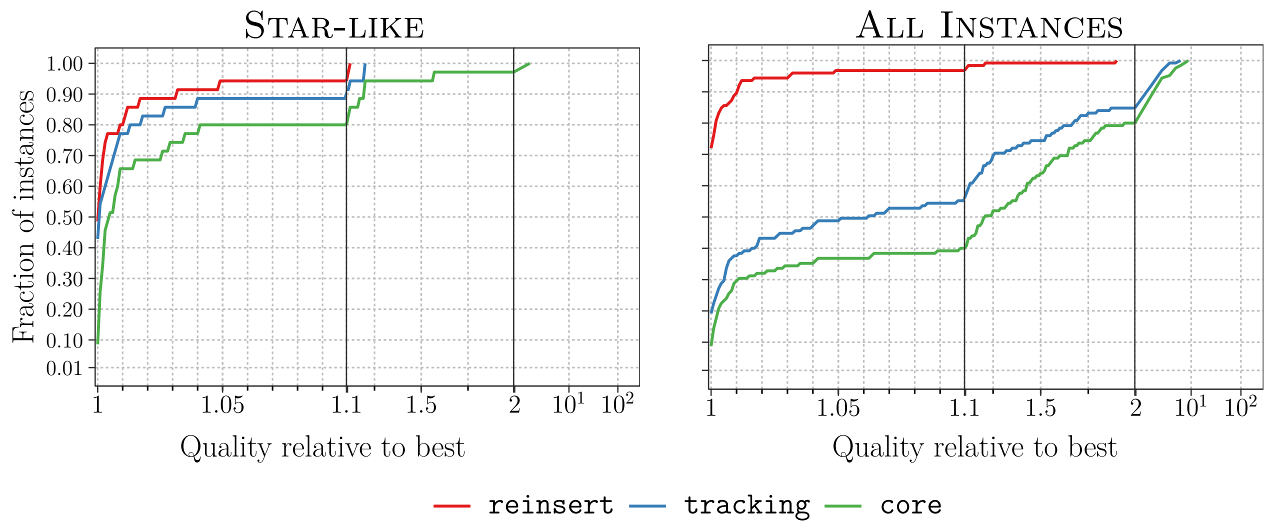


Figure 15: Evaluation of the `reinsert`, `tracking` and `core` strategies for the initial partitioning of the core (based on Optimized$_{local}$).

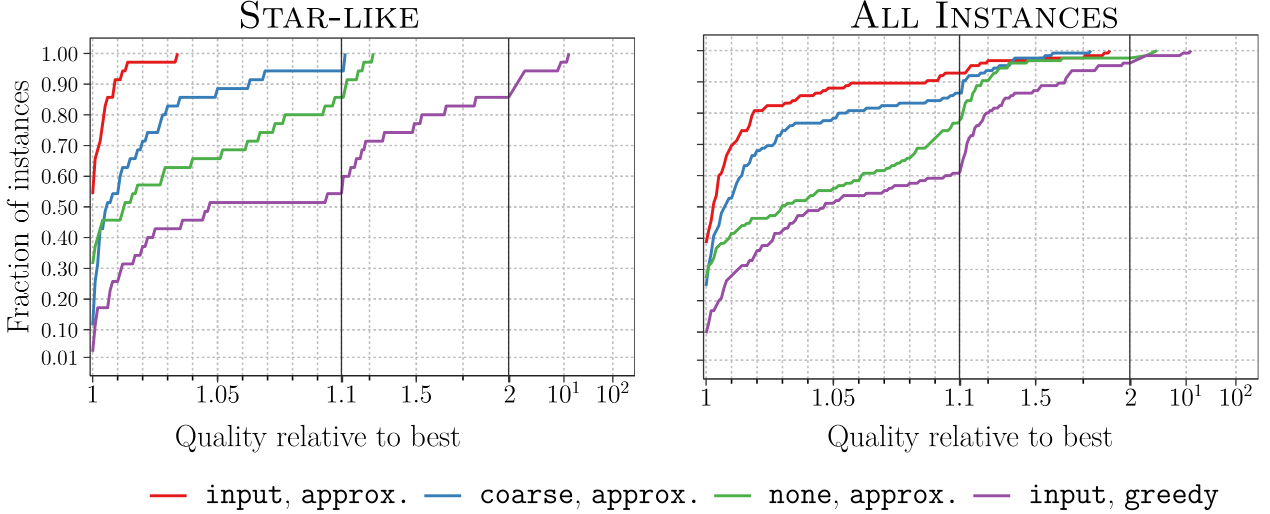


Figure 16: Comparison of different strategies for fixed core partitioning after initial partitioning (based on Optimized$_{local}$).

Initial Partitioning. For the initial partitioning of the core, we evaluate the following strategies: Applying traditional initial partitioning techniques only to the core (`core`), applying traditional techniques to the graph that includes both the core and the peripheral nodes (`reinsert`) and modifying the algorithms used for traditional bipartitioning to track the effect of the peripheral nodes on the cut (`tracking`). The results are presented in Figure 15. Clearly, the `reinsert` strategy provides the best results, finding the best partition for 72% of the instances. On the star-like instances, it is within 11% of the best solution for all instances. In comparison, the quality of the `tracking` and `core` strategies on all instances is worse than the best solution by 5% and 20% in the median, respectively.[11] The inferior results of the `core` strategy indicate that it is important to consider the peripheral nodes during the partitioning of the core. In case of the `tracking` strategy, a possible reason for the worse results in comparison to `reinsert` could be that `tracking` is not aware of edges between peripheral nodes. Also, we observed that the difference between `reinsert` and `tracking` is significantly smaller for balanced configurations.

Fixed Core Partitioning. To compute a high quality assignment of the peripheral nodes after initial partitioning, we use our fixed core partitioning algorithm (see Section 5.3). For the evaluation of this technique, we implement different strategies: The `input` strategy applies the fixed core partitioning to the peripheral nodes of the input graph and afterwards applies two-hop coarsening that respects the resulting assignment. In contrast, the `coarse` strategy first applies two-hop coarsening and uses the coarsest level of peripheral nodes for fixed core partitioning. Additionally, we implemented the `none` variant which does not apply fixed core partitioning and instead uses the result of traditional initial partitioning.

Figure 16 provides a comparison of these strategies. Clearly, the best results are achieved with `input`, which is within 4% of the best solution for all star-like instances. The `coarse` strategy has 0.5% worse quality than the best solution in the median and 10% worse quality than the best solution for 6% of the star-like instances. Also, we observed that the results of the `coarse`

[11] In addition, the current implementation of the `tracking` strategy increases the required running time for initial partitioning by a factor of five in the geometric mean.

strategy improve slightly when larger values for the two-hop coarsening target size p are used.[12] If we do not apply fixed core partitioning, we further decrease the quality. On all instances, the results for the `none` strategy are 3% worse than the best solution in the median. Overall, these results suggest that the fixed core partitioning step is one of the most important components for a high-quality star partitioning algorithm.

The described configurations all use Algorithm 1 for computing an assignment of peripheral nodes with a constant approximation guarantee. To quantify the effect of the fixed core partitioning algorithm on the solution quality, we compare our proposed approximation algorithm to a simple greedy algorithm that sorts the peripheral nodes in decreasing order by the $\frac{\omega(I(U))}{c(u)}$ ratios and assigns each node to the block to which the weight of all incident edges is maximized. If we use the greedy algorithm, we can see that it produces significantly worse partitions than Algorithm 1. For $\approx 50\%$ of the star-like instances, the cut is at least 10% worse than for the best result. This shows that the fixed core partitioning algorithm significantly affects the solution quality. Thus it might be interesting for future work to develop a fixed core partitioning algorithm that further improves upon Algorithm 1.

Final Configurations. Based on the results of our parameter tuning experiments, we include the Optimized$_{local}$, Optimized$_{comm}$, Balanced$_{local}$ and Balanced$_{comm}$ configurations for the comparison to Mt-KaHyPar. We use $p = p' = 2$ as parameter for the two-hop target size and we use the `reinsert` strategy for the initial partitioning of the core. To compute an assignment of the peripheral nodes after initial partitioning, we apply our fixed core partitioning algorithm with the peripheral nodes of the input graph.

6.3. Evaluation of Final Configurations

We evaluate both the quality and the running time of our final configurations on benchmark set B. Our implementation shares many algorithmic components with Mt-KaHyPar-D. We use the coarsening algorithm of Mt-KaHyPar-D to reduce the size of the core, while integrating the two-hop clustering to handle peripheral nodes. Moreover, we extend the initial partitioning phase of Mt-KaHyPar-D with a fixed core partitioning algorithm. During refinement, we use the same local search algorithms as Mt-KaHyPar-D for both core nodes and peripheral nodes (i.e., label propagation and FM local search). Thus, we compare our algorithm to Mt-KaHyPar-D as the most similar configuration.[13]. We do not include an explicit comparison to n-level partitioning algorithms since multilevel partitioners can produce solutions with comparable solution quality [30]. With regards to flow-based refinement, the current implementation is not capable of handling large star-like graphs efficiently [30, see pp. 149-150]. Therefore, we leave this for future work.

Additionally, we include the degree-based partitioning technique that we used to identify star-like instances as a baseline implementation. This technique sorts the nodes by degree, placing the first $\frac{|V|}{k}$ nodes in the first block, the second $\frac{|V|}{k}$ nodes in the second block and so on. Afterwards, we perform one V-cycle [67] on the partition as a refinement step. A V-cycle consists of one coarsening round where contractions are restricted to nodes within the same block of the input partition. Afterwards, the input partition is used as initial partition which

[12] Note that the `coarse` strategy in principle allows for a reduced running time in comparison to the `input` strategy, because only one two-hop coarsening pass is required. However, the architecture of our current implementation does not allow to take advantage of this.

[13] The version of Mt-KaHyPar used for our experiments has the commit hash `f21a419` while the implementation of our star partitioning algorithm has the hash `aa348c3`. Mt-KaHyPar is publicly available from `https://github.com/kahypar/mt-kahypar`

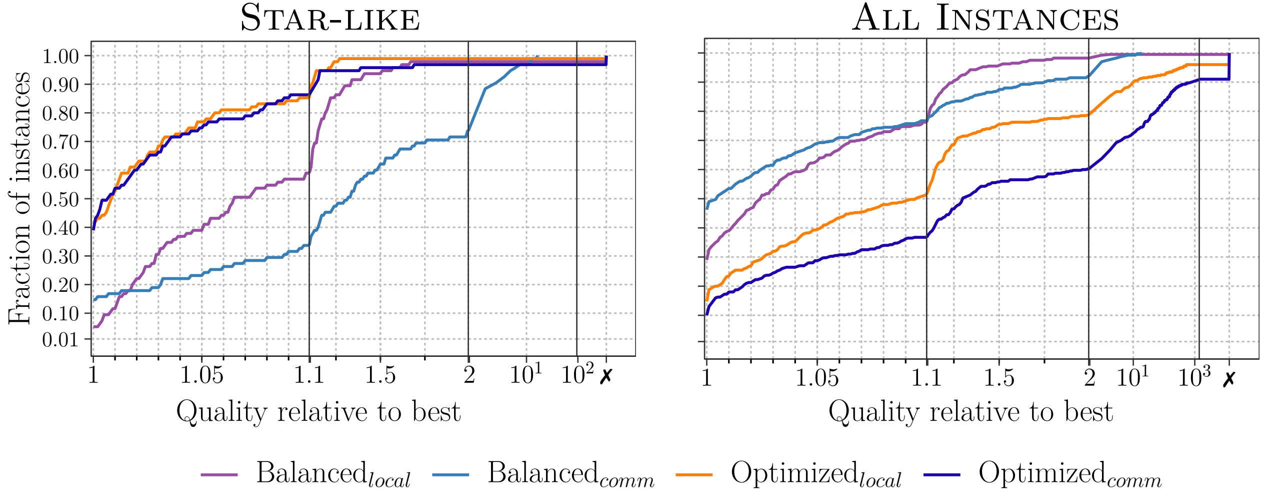


Figure 17: Performance profiles comparing the results of the different configurations both on star-like instances (left) and the complete benchmark set B (right).

induces the same cut and balance as on the input graph. This allows to improve the input partition in the uncoarsening phase.

Moreover, we observed that the star-partitioning techniques negatively impact the solution quality on mesh networks. We therefore include a configuration that detects whether or not the instance is a mesh graph and disables the star partitioning techniques if this is the case. Mesh graph detection is already implemented in Mt-KaHyPar and is based on the average μ and standard deviation σ of the node degrees of the graph. We assume a graph is a mesh network if $\sigma \leq \frac{\mu}{2}$, i.e., the graph has a uniform degree distribution [30].

Quality. Figure 17 provides a comparison of our final configurations. The Optimized$_{local}$ and Optimized$_{comm}$ configurations achieve the best results on star-like instances. They compute comparable solutions, both finding the best partition for 40% of instances. However, Optimized$_{local}$ has significantly better results on all instances, while the quality of Optimized$_{comm}$ is worse than the best solution by at least a factor of two for 40% of instances. Both balanced configurations achieve similar quality on the complete benchmark set. While Balanced$_{comm}$ finds the best solution for 40% of instances, for 10% of instances it also computes partitions which are worse than the best solution by a factor of two. Notably, Balanced$_{local}$ has better quality than Balanced$_{comm}$ on the star-like instances by 13% in the median. This indicates that Balanced$_{comm}$ provides only a small improvement on star-like instances in comparison to traditional techniques. Note that all configurations except Balanced$_{comm}$ reached the time limit for a few instances.

The results of comparing our configurations to Mt-KaHyPar as well as the degree-based approach are summarized in Figure 18. Notably, our configurations compute significantly better solutions than Mt-KaHyPar on the star-like instances. The Optimized$_{local}$ configuration achieves better quality than Mt-KaHyPar by a factor of 1.1 for 80% and better quality by a factor of 2 for 40% of the instances. For some of the recompX graphs (e.g., `recomp_english1GB_5` with $k \geq 4$) the quality is better by a factor of 10 or more. The degree-based approach also performs surprisingly well on star-like instances, with a quality that is only 5% worse than the best solution in the median. If we look at the Balanced$_{local}$ configuration, we see that adding mesh graph detection further improves the quality on the complete benchmark set, as shown in Figure 18 (top right). This is expected as mesh graphs do not have star-like properties. If

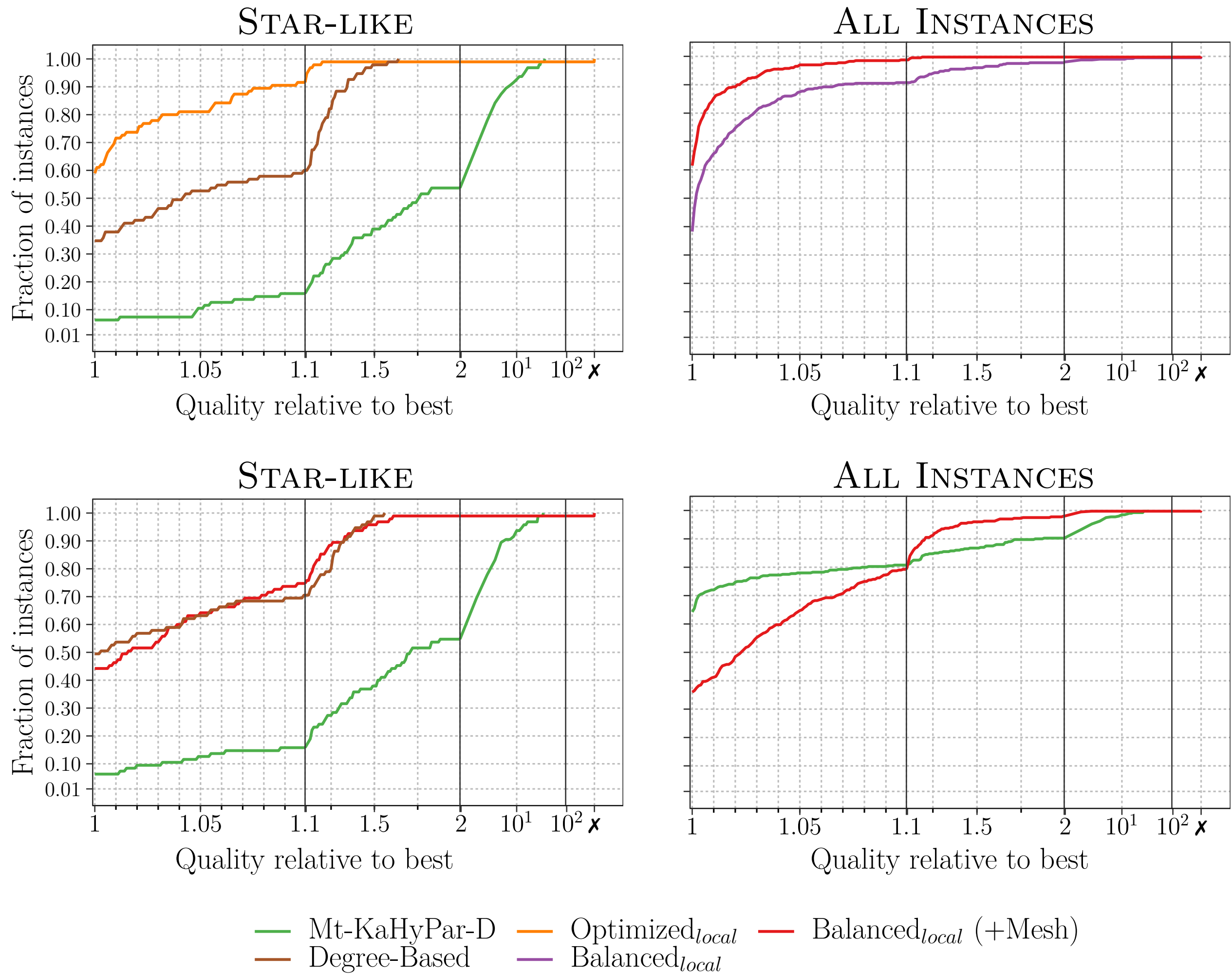


Figure 18: Comparison to Mt-KaHyPar. We evaluate the effect of adding mesh graph detection to the Balanced$_{local}$ configuration (+Mesh, top right), compare both `local` configurations to Mt-KaHyPar and the degree-based approach on the star-like instances (left) and compare the Balanced$_{local}$ (+Mesh) configuration to Mt-KaHyPar on all instances (bottom right).

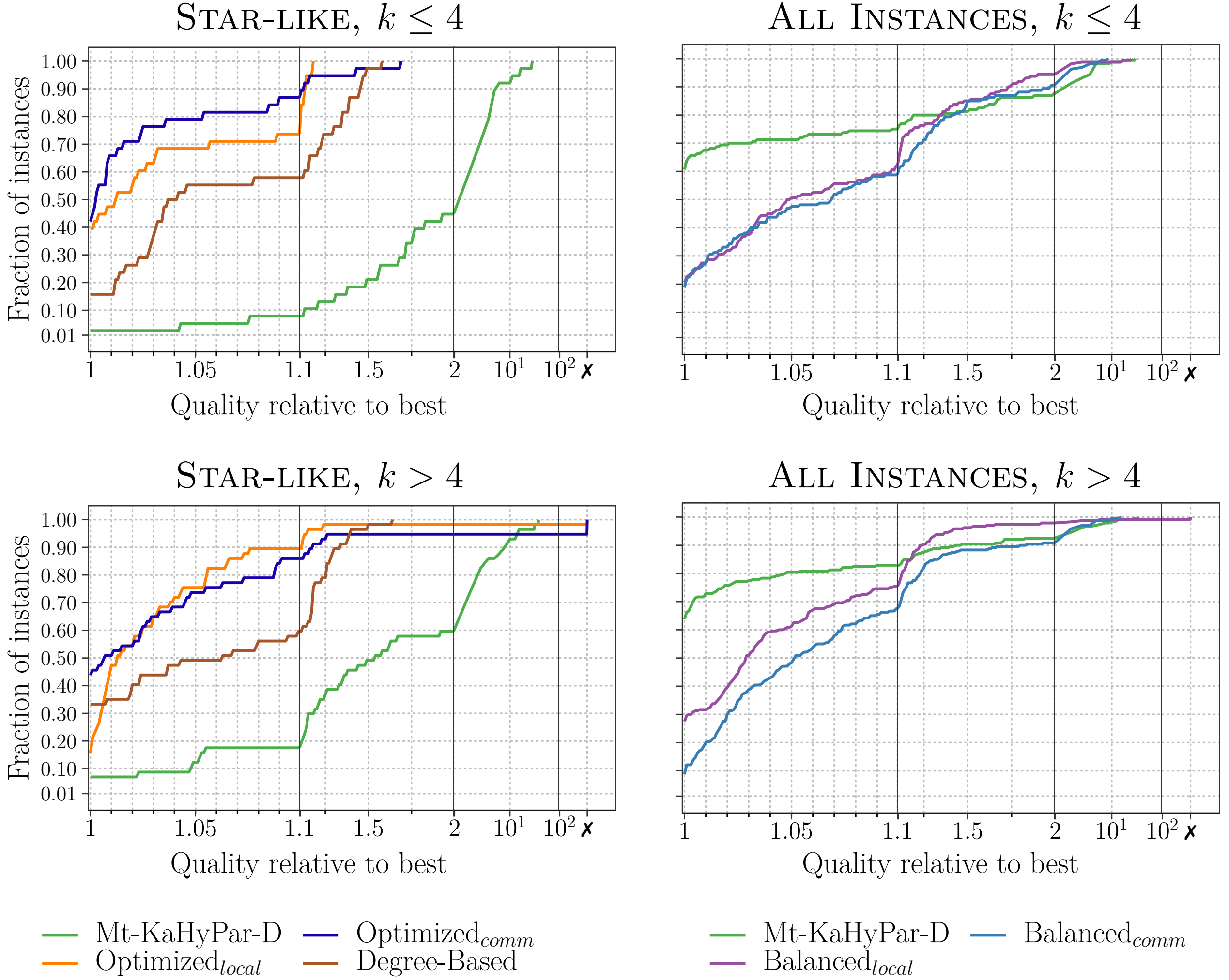


Figure 19: Comparison to Mt-KaHyPar and the degree-based technique for $k \in \{2, 4\}$ (top) and for $k \in \{8, 16, 32\}$ (bottom).

we compare the configuration that includes mesh graph detection to Mt-KaHyPar, it finds the best solution only for 40% of instances but unlike Mt-KaHyPar it is within a factor of 2 of the best solution for almost all instances. On star-like instances, the Balanced$_{local}$ (+Mesh) configuration has better quality than Mt-KaHyPar with an improvement of 65% in the median. However, in this case the degree-based approach has competitive performance. In summary, the Balanced$_{local}$ (+Mesh) configuration provides good quality on all instances, but only the Optimized$_{local}$ configuration is capable of outperforming the degree-based approach for star-like instances.

Additionally, we evaluate how different values for k influence the solution quality in Figure 19. Overall, the results are similar for $k \leq 4$ and $k > 4$. However, the difference on star-like instances is more pronounced for $k \leq 4$. The results of Mt-KaHyPar are worse than the best solution by a factor of 2 in the median while for $k > 4$, the median difference is only a factor of 1.5. In addition, for $k \leq 4$ Optimized$_{comm}$ computes slightly better results than Optimized$_{local}$. On all instances, the Balanced$_{local}$ and Balanced$_{comm}$ configurations have very similar performance for $k \leq 4$. For $k > 4$, Balanced$_{local}$ provides better results by 1.5% in the median. This indicates that the `comm` strategy works better for small values of k.

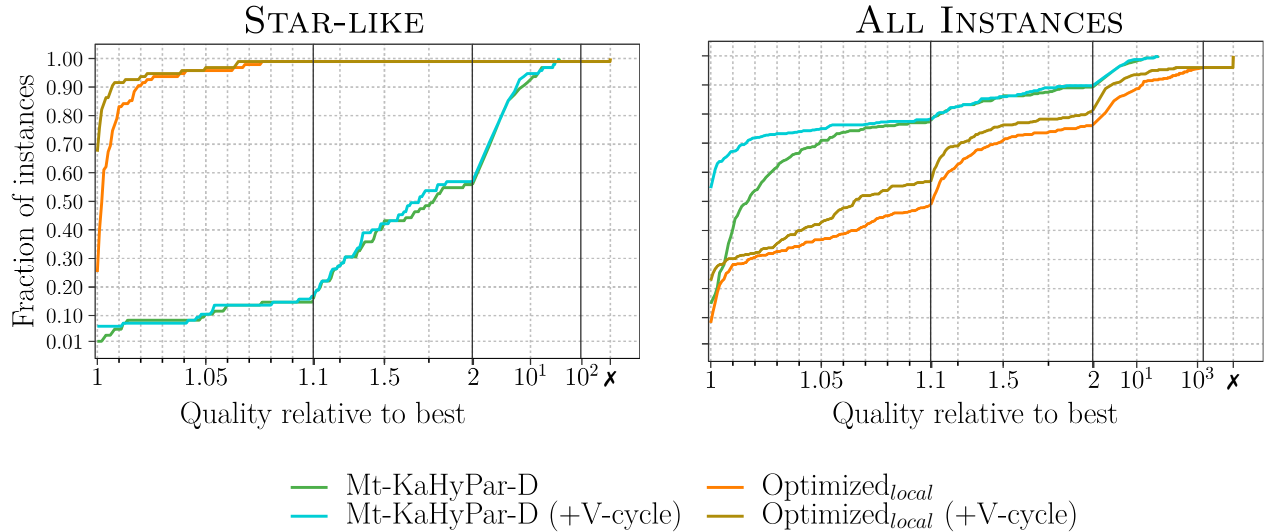


Figure 20: Performance profiles evaluating the effect of using one V-Cycle for Mt-KaHyPar-D and Optimized$_{local}$.

V-cycles. V-cycles are a common technique to further improve the quality of a partition [67], as already described. We evaluate the effect of V-cycles in Figure 20. As it can be seen, adding one V-cycle provides a substantial quality improvement on all instances. Mt-KaHyPar-D with one V-cycle finds the best solution for more than half of instances. In case of the Optimized$_{local}$ strategy, adding one V-cycle reduces the number of outliers with very bad quality (but the results are still clearly inferior to Mt-KaHyPar-D). However, on star-like instances adding a V-cycle does not provide a significant improvement for Mt-KaHyPar-D. For Optimized$_{local}$ a small improvement is achieved, but the quality difference compared to the variant without a V-cycle is less than 1% on most instances. This indicates that adding V-cycles is not an effective strategy for star-like graphs.

Running Time. We compare the running time of the algorithms in Figure 21. As it can be seen, the star partitioning techniques have longer running times then Mt-KaHyPar. The balanced configurations are slower than Mt-KaHyPar by a factor of two, while Optimized$_{local}$ is slower by a factor of four and Optimized$_{comm}$ is slower by a factor of ten on average.[14] The reason for this is that in our current implementation, we made sacrifices on performance in favor of exploring the large design space of the star partitioning problem. However, this is likely not inherent to star partitioning. With an improved understanding of the problem domain and a more specialized implementation it should be possible to achieve a similar running time to Mt-KaHyPar-D – we leave the engineering and optimization of such an implementation for future work.

6.4. Evaluation of Algorithm Portfolios

Effectiveness Tests. Merely comparing the solution quality of two algorithms $\mathcal{A}$ and $\mathcal{B}$ is often not sufficient for a comprehensive evaluation. If algorithm $\mathcal{A}$ produces better partitions but is significantly slower than algorithm $\mathcal{B}$, it may have an unfair advantage due to its longer

[14] The reason that Optimized$_{comm}$ is significantly slower than Optimized$_{local}$ might be that it detects more nodes as peripheral and it does this as a preprocessing step, thereby placing more load on the data structure that represents peripheral nodes.

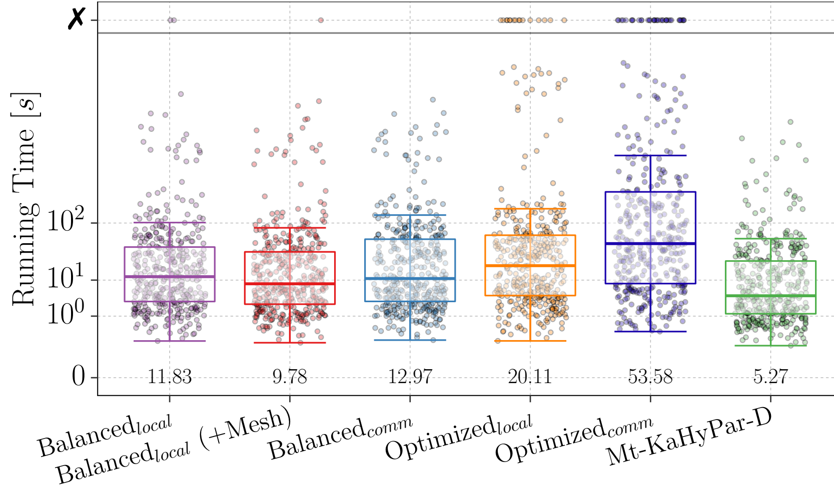


Figure 21: Comparing the running time of each algorithm on benchmark set B. The number under each boxplot denotes the geometric mean of the running time for the corresponding algorithm. Note, ✗ marks instances that exceeded the time limit.

running time. Effectiveness tests [2] allow to compare two algorithms with different running time by performing additional repetitions with the faster algorithm. The situation is similar if we want to evaluate a *portfolio* that consists of multiple algorithms – which is of particular interest in our case, as the best results might be achieved by combining an algorithm that is specialized for star-like instances with a traditional multilevel partitioner. Therefore, we introduce a modified approach to effectiveness tests that allows to compare a single algorithm with a portfolio of algorithms.

Definition 6.1. *A portfolio $\mathcal{P} = \{(\mathcal{A}_1, p_1), \dots, (\mathcal{A}_n, p_n)\}$ is a set of algorithms together with a probability that the algorithm is selected for a given sample, subject to $p_i \geq 0$ for $i \leq n$ and $p_1 + \dots + p_n = 1$.*

Suppose we want to compare a portfolio $\mathcal{P} = \{(\mathcal{A}_1, p_1), \dots, (\mathcal{A}_n, p_n)\}$ with an algorithm $\mathcal{B}$ on instance I. We generate a *virtual instance* as follows: For a given time limit T, we sample runs of $\mathcal{B}$ until their accumulated running time exceeds T. Let $t^1_\mathcal{B}, \dots, t^l_\mathcal{B}$ denote the running times. We accept the last sample with probability $\frac{1}{T}(T - \sum_{i=1}^{l-1} t^i_\mathcal{B})$, which ensures that the expected total running time of the samples equals T. We then plot the best solution of the included samples. The solution quality for the portfolio $\mathcal{P}$ is determined similarly, except that for a given sample we determine the used algorithm at random.To minimize the variance of the results, we generate ten virtual instances per instance.

The time limit T is determined by sampling ten repetitions of algorithm $\mathcal{B}$ with running times $t^1_\mathcal{B}, \dots, t^{10}_\mathcal{B}$ and ten repetitions of each algorithm in $\mathcal{P}$.[15] Let $t^{max}_\mathcal{P} := \max_{i \leq n, j \leq 10} t^j_{\mathcal{A}_i}$ be the maximum time required by an algorithm in the portfolio, where $t^j_{\mathcal{A}_i}$ denotes the running time of the j'th repetition of algorithm $\mathcal{A}_i$ (we exclude infeasible repetitions). Then, the time limit is defined as $T := \min\{\max\{\sum_{i=1}^{5} t^i_\mathcal{B}, 2t^{max}_\mathcal{P}\}, \sum_{i=1}^{10} t^i_\mathcal{B}\}$. The purpose of this definition is to allow for enough running time such that a portfolio solution becomes worthwhile.

Results. In the following, we use effectiveness tests to evaluate algorithm portfolios with the goal of achieving good quality on all instances. We use portfolios consisting of Mt-KaHyPar-D

[15] Since we executed ten runs per algorithm, this means we use all of the repetitions for determining T but randomize their order.

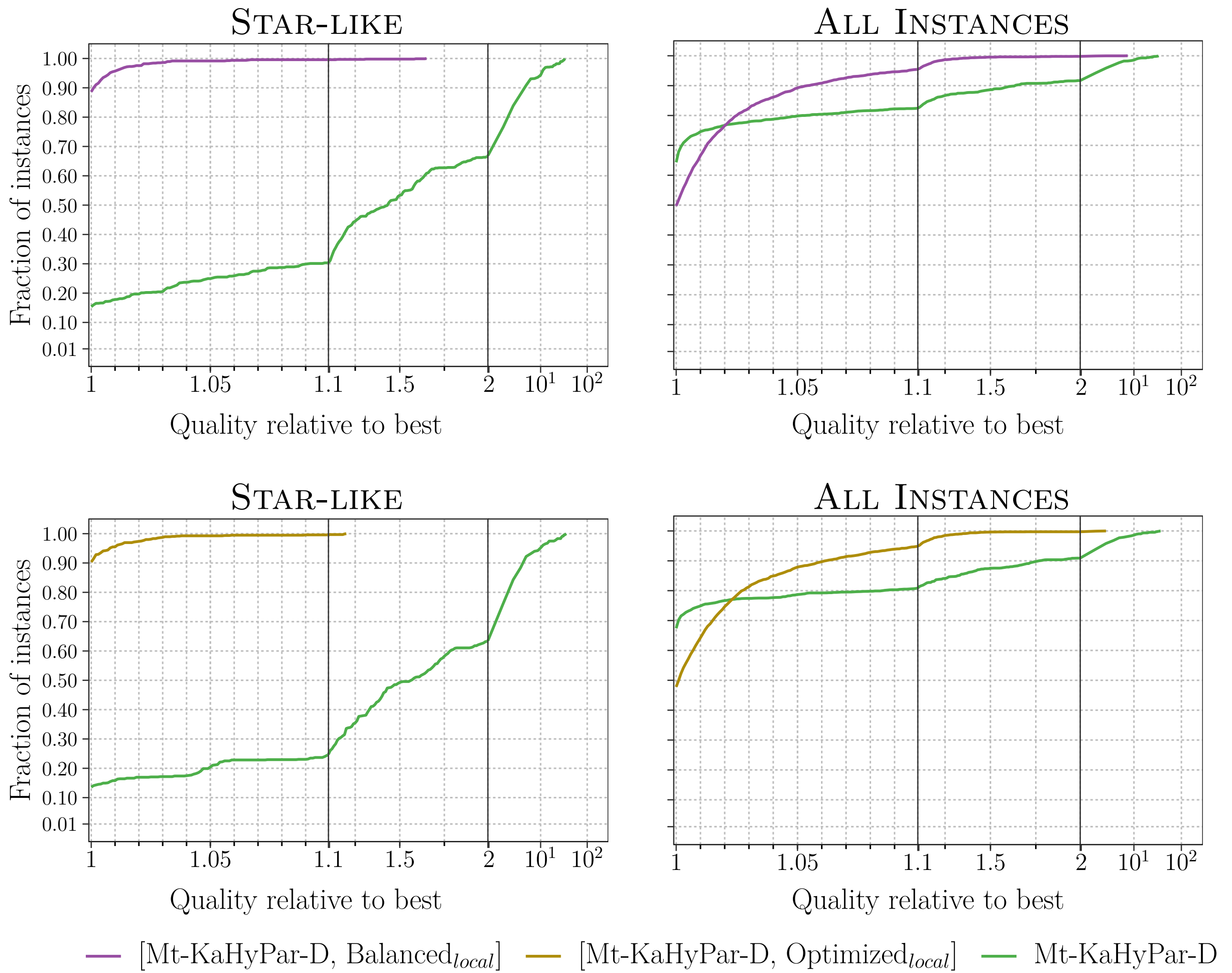


Figure 22: Evaluation of algorithm portfolios. We compare portfolios that use the Balanced$_{local}$ (top) and Optimized$_{local}$ (bottom) configurations to Mt-KaHyPar.

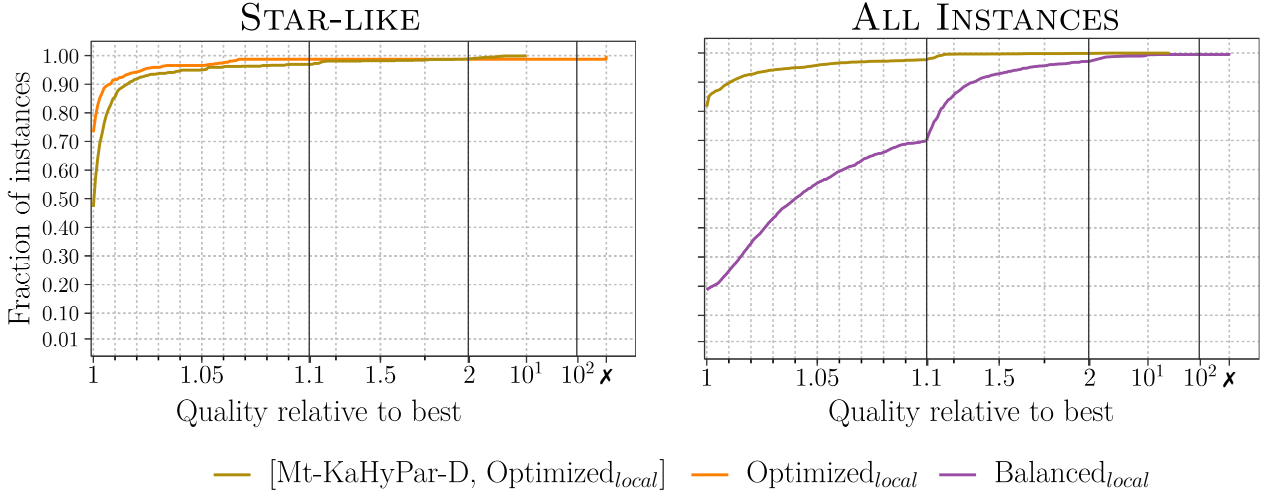


Figure 23: Comparing the [Mt-KaHyPar-D, Optimized$_{local}$] portfolio to the Optimized$_{local}$ configuration on star-like instances (left) and to the Balanced$_{local}$ configuration on all instances (right).

and one other algorithm with a probability of $\frac{1}{2}$ for both algorithms. Additionally, we always run Mt-KaHyPar-D for the first sample, since this strategy provides better quality than randomizing all samples.[16]

Figure 22 shows the results for portfolios consisting of Mt-KaHyPar-D and one of our final configurations. Note that Mt-KaHyPar-D always runs first. On the star-like instances, both portfolios clearly outperform Mt-KaHyPar: The [Mt-KaHyPar-D, Balanced$_{local}$] portfolio achieves better quality than Mt-KaHyPar by a factor of 1.4 in the median, while the difference for [Mt-KaHyPar-D, Optimized$_{local}$] is a factor of 1.55 in the median. On all instances, Mt-KaHyPar finds the best solution for a larger fraction of instances than the portfolios ($\approx 20\%$ more of the instances). However, the portfolios provide significantly better results for the high percentiles. For 10% of instances, the portfolios using the Balanced$_{local}$ configuration and the Optimized$_{local}$ configuration find better solutions by a factor of 1.53 and 1.63, respectively. The difference is even larger for higher percentiles, up to a factor of 10 or more. This indicates that the [Mt-KaHyPar-D, Optimized$_{local}$] portfolio is a good choice for achieving high quality on all instances.

Further, we compare the effectiveness of the [Mt-KaHyPar-D, Optimized$_{local}$] portfolio to our other configurations in Figure 23. The plots show that the results on star-like instances are comparable to the Optimized$_{local}$ configuration, finding the best solution only on 48% instead of 73% of instances. However, the result of the portfolio is within 2% of the best solution on more than 90% of instances – thus the overall difference is minor. For a few instances, the result is significantly worse, most likely because the portfolio only chose Mt-KaHyPar in these cases. On all instances, the portfolio clearly outperforms the Balanced$_{local}$ configuration: It finds the best solution on 80% of instances and the quality is better than the quality of Balanced$_{local}$ by 4% in the median. Overall, from our presented algorithms the [Mt-KaHyPar-D, Optimized$_{local}$] portfolio seems to be the best choice for achieving a high quality both on star-like graphs and on other instances.

[16] We then adjust the probabilities for the current virtual instance to compensate the advantage given to Mt-KaHyPar-D. Let $\mathcal{A}_1$ be the algorithm that runs first and let $\delta_{\mathcal{A}_1} := \min\{t^1_{\mathcal{A}_1}/T, p(\mathcal{A}_1)\}$. Then, the adjusted probabilities are defined as $\tilde{p}_1 := \frac{1}{1-\delta_{\mathcal{A}_1}}(p(\mathcal{A}_1) - \delta_{\mathcal{A}_1})$ and $\tilde{p}_i := \frac{1}{1-\delta_{\mathcal{A}_1}}p(\mathcal{A}_i)$ for $i > 1$.

7. Conclusion

Star-like graphs are characterized by their high number of peripheral nodes with low degree connected to a small and dense core. The properties of these instances allow us to use algorithms specifically tailored for star-like instances to compute partitions with significantly smaller cut than traditional multilevel algorithms. In this thesis, we developed the components that are necessary to achieve this in practice. From a theoretical point of view, we constructed an approximation algorithm where the running time is fixed-parameter tractable with regards to the size of the core of the graph. The underlying idea is to reduce the problem to the fixed core partitioning problem, for which we can compute a constant factor approximation in polynomial time based on algorithms for the min-knapsack problem. Using a min-knapsack algorithm with given approximation ratio R (a $\frac{3}{2}$-approximation can be computed in quadratic running time [15] and an FPTAS exists [35]), our algorithm achieves an approximation ratio of $R + 1$ – which is an important result as there exists no constant factor approximation for general graph partitioning [10].

However, the running time of our approximation algorithm is exponential in the size of the core, which would be infeasible in practice. Therefore, we explored techniques for modifying the multilevel paradigm such that it allows to exploit the properties of star-like graphs. We found that in traditional multilevel partitioning, peripheral nodes tend to get contracted onto core nodes in the coarsening phase. Due to the increased node weight, this forces initial partitioning to separate the core in multiple blocks. This induces a large cut, which can not repaired during uncoarsening. In our implementation, we solve this by using the following components: First, a strategy to detect peripheral nodes in the coarsening phase. Testing multiple strategies, we achieved the best results using the `local` strategy that is based on comparing the ratio of the incident edge weight of a node relative to its node weight between adjacent nodes. After separating the peripheral nodes, we apply two-hop coarsening on the peripheral nodes. As peripheral nodes are often not connected, we presented techniques to find contraction partners based on the similarity of their neighborhoods. More precisely, this includes clustering of nodes with identical neighborhood sets, or if this does not sufficiently reduce the size, we group nodes with similar neighborhoods using the min-hashing technique. In the initial partitioning phase, our theoretical results are put into practice. We showed that our approximation algorithm can be used for assigning the peripheral nodes, which significantly improves the quality of the resulting initial partition.

Our experimental results demonstrate that current state-of-the-art multilevel partitioners are not capable of finding high-quality solutions for star-like graphs – for these instances, an astonishingly simple technique based on sorting the nodes by their degree provides significantly better results. The best quality is achieved by our Optimized$_{local}$ configuration, which we tuned to achieve the highest possible quality on star-like instances. Compared to Mt-KaHyPar, it improves the quality by a factor of two for 40% of the star-like instances and up to a factor of more than ten for some of the instances. However, this comes at the cost of worse quality for the remaining instances. Thus, we also investigated balanced configurations that provide good overall quality. To further improve the quality, we proposed to use an algorithm portfolio: On our complete benchmark set, the portfolio consisting of Mt-KaHyPar and the Optimized$_{local}$ configuration computes better results than any single algorithm by a significant margin.

7.1. Future Work

We introduced the fixed core partitioning problem in Section 4 and showed in our experimental evaluation that we can substantially improve the initial partitions on star-like graphs using

an approximation algorithm for fixed core partitioning. Thus, a more thorough analysis of this problem is both of theoretical interest and might lead to improved partitions in practice. For this, future research could develop an algorithm with a better approximation guarantee, possibly based on a generalization of our proposed two-slot algorithm. Note that the achievable approximation ratio is bound by the approximation ratio of the used algorithm for the min-knapsack problem. Alternatively, heuristic techniques could be used to improve fixed core partitioning algorithms on real-world instances. In both cases, it would be insightful to integrate the improved algorithm into the initial partitioning and provide an evaluation of the effect on the quality for star-like graphs.

Since we adapted the multilevel scheme in a novel way, it is natural that there are multiple areas for further research with regards to engineering a high-performing algorithm. Our experimental evaluation shows that the detection of peripheral nodes is of central importance for the quality of a star partitioning algorithm. While we explored and evaluated multiple detection strategies, the design space is large and allows for the development of more alternative strategies. Further, some of the parameters used for two-hop coarsening could be investigated in more detail. Specifically, exploring different target sizes for the coarsening of the peripheral nodes could be beneficial. With regards to initial partitioning, we developed two approaches for the application of the fixed core partitioning algorithm: either using the peripheral nodes of the input graph or of the smallest graph. This involves a trade-off between quality and running time (since the first approach requires a second round of two-hop coarsening) and thus raises the question whether it is possible to improve (or avoid) this trade-off. Another area of interest that is not covered in this thesis is the development of refinement algorithms that are specialized for star-like graphs. Peripheral nodes tend to have low impact on the overall cut in comparison to their weight. Therefore, it could be beneficial to develop a refinement algorithm that is capable of exploiting this. One approach could be developing a refinement algorithm that can move (heavy) high-degree nodes to another block, e.g., by swapping them with a large set of peripheral nodes.

Further, our current implementation focuses on the exploration of multiple possible design choices and thus is not optimized with regards to running time. Consequently, future research should focus on reducing the running time overheads, while providing the same solution quality on star-like instances.

This thesis explores how to improve the quality on star-like graphs. However, it is likely that there are also hypergraph instances with similar properties where star partitioning techniques are applicable. Therefore, the notion of star-like graphs should be extended to hypergraphs and the prevalence of such hypergraph instances in practice should be investigated. Then, the techniques developed in this work should be generalized to hypergraphs. Section 5.5 provides a starting point for this.

We have seen that algorithms specialized for star-like graphs involve a trade-off, decreasing the quality for some of the other instances. To overcome this, we proposed to use an algorithm portfolio including both a traditional multilevel algorithm and a specialized star partitioning algorithm. However, there is another approach that could provide even better results. For a given instance, we could use a preprocessing step to detect whether it is beneficial or not to use a star partitioning algorithm. Thereby, we would avoid the running time overhead of executing both algorithms and still achieve the same quality, provided the detection method is precise enough. The detection could be based on statistical properties of the graph such as the node degree distribution, or it could try to find peripheral nodes and consider, e.g., the density of the subgraph of peripheral nodes and the density of the core. Here, it might also be possible to reach a high accuracy by using machine learning techniques for the detection.

References

[1] Amine Abou-Rjeili and George Karypis. "Multilevel Algorithms for Partitioning Power-law Graphs". In: *20th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2006. DOI: 10.1109/IPDPS.2006.1639360.

[2] Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. "High-Quality Shared-Memory Graph Partitioning". In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 31.11 (2020), pp. 2710–2722. DOI: 10.1109/TPDS.2020.3001645.

[3] Yaroslav Akhremtsev et al. "Engineering a direct k-way Hypergraph Partitioning Algorithm". In: *19th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2017, pp. 28–42. DOI: 10.1137/1.9781611974768.3.

[4] Maram Assi and Ramzi A. Haraty. "A Survey of the Knapsack Problem". In: *International Arab Conference on Information Technology (ACIT)*. IEEE, 2018, pp. 1–6. DOI: 10.1109/ACIT.2018.8672677.

[5] Cevdet Aykanat, Berkant Barla Cambazoglu, and Bora Uçar. "Multi-level direct k-way Hypergraph Partitioning with Multiple Constraints and Fixed Vertices". In: *Journal of Parallel and Distributed Computing* 68.5 (2008), pp. 609–625. DOI: 10.1016/j.jpdc.2007.09.006.

[6] Cevdet Aykanat et al. "Adaptive Decomposition and Remapping Algorithms for Object-space-parallel Direct Volume Rendering of Unstructured Grids". In: *Journal of Parallel and Distributed Computing* 67.1 (2007), pp. 77–99. DOI: 10.1016/j.jpdc.2006.05.005.

[7] David A. Bader et al. "Benchmarking for Graph Clustering and Partitioning". In: *Encyclopedia of Social Network Analysis and Mining*. 2014, pp. 73–82. DOI: 10.1007/978-1-4614-6170-8_23.

[8] Vincent D. Blondel et al. "Fast Unfolding of Communities in Large Networks". In: *Journal of Statistical Mechanics: Theory and Experiment* 10 (2008). DOI: 10.1088/1742-5468/2008/10/P10008.

[9] Andrei Z. Broder. "On the Resemblance and Containment of Documents". In: *Compression and Complexity of Sequences (SEQUENCES)*. IEEE, 1997, pp. 21–29. DOI: 10.1109/SEQUEN.1997.666900.

[10] Thang Nguyen Bui and Curt Jones. "Finding Good Approximate Vertex and Edge Partitions is NP-Hard". In: *Information Processing Letters* 42.3 (1992), pp. 153–159. DOI: 10.1016/0020-0190(92)90140-Q.

[11] Rainer E. Burkard and Eranda Çela. "Linear Assignment Problems and Extensions". In: *Handbook of Combinatorial Optimization*. Springer, 1999, pp. 75–149. DOI: 10.1007/978-1-4757-3023-4_2.

[12] Ümit V. Çatalyürek and Cevdet Aykanat. "Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication". In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 10.7 (1999), pp. 673–693. DOI: 10.1109/71.780863.

[13] Ümit V. Çatalyürek and Cevdet Aykanat. "Patoh (Partitioning Tool for Hypergraphs)". In: *Encyclopedia of Parallel Computing*. Springer, 2011.

[14] Ümit V. Çatalyürek et al. "More Recent Advances in (Hyper)Graph Partitioning". In: *Computing Research Repository (CoRR)* abs/2205.13202 (2022). DOI: 10.48550/arXiv.2205.13202. arXiv: 2205.13202.

[15] János Csirik. "Heuristics for the 0-1 Min-Knapsack Problem". In: *Acta Cybernetica* 10.1-2 (1991), pp. 15–20.

[16] George B. Dantzig. "Discrete-Variable Extremum Problems". In: *Operations Research* 5.2 (1957), pp. 266–277. DOI: 10.1287/opre.5.2.266.

[17] Mehmet Deveci, Kamer Kaya, and Ümit V. Çatalyürek. "Hypergraph Sparsification and Its Application to Partitioning". In: *42nd International Conference on Parallel Processing (ICPP)*. IEEE Computer Society, 2013, pp. 200–209. DOI: 10.1109/ICPP.2013.29.

[18] Elizabeth D. Dolan and Jorge J. Moré. "Benchmarking Optimization Software with Performance Profiles". In: *Mathematical Programming* 91.2 (2002), pp. 201–213. DOI: 10.1007/s101070100263.

[19] Charles M. Fiduccia and Robert M. Mattheyses. "A Linear-Time Heuristic for Improving Network Partitions". In: *19th Design Automation Conference (DAC)*. ACM/IEEE, 1982, pp. 175–181. DOI: 10.1145/800263.809204.

[20] ICM Flinsenberg. "Graph Partitioning for Route Planning in Car Navigation Systems". In: *11th International Association of Institutes of Navigation World Congress (IAIN)*. Eindhoven University of Technology, 2003.

[21] Michael L. Fredman and Robert Endre Tarjan. "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms". In: *Journal of the ACM* 34.3 (1987), pp. 596–615. DOI: 10.1145/28869.28874.

[22] Daniel Funke et al. "Communication-free Massively Distributed Graph Generation". In: *Journal of Parallel and Distributed Computing* 131 (2019), pp. 200–217. DOI: 10.1016/j.jpdc.2019.03.011.

[23] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[24] Lars Gottesbüren and Michael Hamann. "Deterministic Parallel Hypergraph Partitioning". In: *28th International European Conference on Parallel and Distributed Computing (Euro-Par)*. Vol. 13440. Lecture Notes in Computer Science. Springer, 2022, pp. 301–316. DOI: 10.1007/978-3-031-12597-3_19.

[25] Lars Gottesbüren, Tobias Heuer, and Peter Sanders. "Parallel Flow-Based Hypergraph Partitioning". In: *29th European Symposium on Algorithms (ESA)*. Vol. 233. LIPIcs. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 5:1–5:21. DOI: 10.4230/LIPIcs.SEA.2022.5.

[26] Lars Gottesbüren et al. "Deep Multilevel Graph Partitioning". In: *29th European Symposium on Algorithms (ESA)*. Vol. 204. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 48:1–48:17. DOI: 10.4230/LIPIcs.ESA.2021.48.

[27] Lars Gottesbüren et al. "Scalable Shared-Memory Hypergraph Partitioning". In: *23rd Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2021, pp. 16–30. DOI: 10.1137/1.9781611976472.2.

[28] Lars Gottesbüren et al. "Shared-Memory n-level Hypergraph Partitioning". In: *24th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, Jan. 2022, pp. 131–144. DOI: 10.1137/1.9781611977042.11.

[29] Michael Hamann and Ben Strasser. "Graph Bisection with Pareto Optimization". In: *ACM Journal of Experimental Algorithmics* 23 (2018). DOI: 10.1145/3173045.

[30] Tobias Heuer. "Scalable High-Quality Graph and Hypergraph Partitioning". PhD thesis. Karlsruhe Institute of Technology, Germany, 2022.

[31] Tobias Heuer, Nikolai Maas, and Sebastian Schlag. "Multilevel Hypergraph Partitioning with Vertex Weights Revisited". In: *19th International Symposium on Experimental Algorithms (SEA)*. Vol. 190. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 8:1–8:20. DOI: 10.4230/LIPIcs.SEA.2021.8.

[32] Tobias Heuer and Sebastian Schlag. "Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure". In: *16th International Symposium on Experimental Algorithms (SEA)*. Vol. 75. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 21:1–21:19. DOI: 10.4230/LIPIcs.SEA.2017.21.

[33] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. "Engineering a Scalable High Quality Graph Partitioner". In: *24th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470485.

[34] Ellis Horowitz and Sartaj Sahni. "Computing Partitions with Applications to the Knapsack Problem". In: *Journal of the ACM* 21.2 (1974), pp. 277–292. DOI: 10.1145/321812.321823.

[35] Mohammad Tauhidul Islam. "Approximation Algorithms for Minimum Knapsack Problem". Master Thesis. University of Lethbridge, 2009.

[36] Artur Jez. "Faster Fully Compressed Pattern Matching by Recompression". In: *ACM Trans. Algorithms* 11.3 (2015), 20:1–20:43. DOI: 10.1145/2631920.

[37] Andrew B. Kahng et al. *LSI Physical Design - From Graph Partitioning to Timing Closure*. Vol. 312. Springer, 2011. DOI: 10.1007/978-3-030-96415-3.

[38] George Karypis and Vipin Kumar. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*. 1997.

[39] George Karypis and Vipin Kumar. "Multilevel k-way Hypergraph Partitioning". In: *VLSI Design* 11.3 (2000), pp. 285–300. DOI: 10.1155/2000/19436.

[40] George Karypis and Vipin Kumar. "Multilevel k-way Partitioning Scheme for Irregular Graphs". In: *Journal of Parallel and Distributed Computing* 48.1 (1998), pp. 96–129. DOI: 10.1006/jpdc.1997.1404.

[41] George Karypis and Vipin Kumar. "Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs". In: *ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, 1996, p. 35. DOI: 10.1109/SC.1996.32.

[42] George Karypis et al. "Multilevel Hypergraph Partitioning: Applications in VLSI Domain". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 7.1 (1999), pp. 69–79. DOI: 10.1109/92.748202.

[43] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. "Scalable SIMD-Efficient Graph Processing on GPUs". In: *International Conference on Parallel Architectures and Compilation (PACT)*. IEEE Computer Society, 2015, pp. 39–50. DOI: 10.1109/PACT.2015.15.

[44] Richard E. Korf. "From Approximate to Optimal Solutions: A Case Study of Number Partitioning". In: *14th International Joint Conference on Artificial Intelligence (IJCAI)*. IJCAI, 1995, pp. 266–272.

[45] Richard E. Korf. "Multi-Way Number Partitioning". In: *21st International Joint Conference on Artificial Intelligence (IJCAI)*. IJCAI, 2009, pp. 538–543.

[46] Harold W. Kuhn. "The Hungarian Method for the Assignment Problem". In: *Naval Research Logistics Quarterly* 2.1-2 (1955), pp. 83–97. DOI: 10.1002/nav.3800020109.

[47] Dominique LaSalle et al. “Improving Graph Partitioning for Modern Graphs and Architectures”. In: *5th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. ACM, 2015, 14:1–14:4. DOI: 10.1145/2833179.2833188.

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

[49] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. “Finding Similar Items”. In: *Mining of Massive Datasets* (2014), pp. 68–122.

[50] Nikolai Maas. “Multilevel Hypergraph Partitioning with Vertex Weights Revisited”. Bachelor Thesis. Karlsruhe Institute of Technology, Germany, 2020.

[51] Oliver Marquardt and Stefan Schamberger. “Open Benchmarks for Load Balancing Heuristics in Parallel Adaptive Finite Element Computations”. In: *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. CSREA Press, 2005, pp. 685–691.

[52] Henning Meyerhenke, Peter Sanders, and Christian Schulz. “Parallel Graph Partitioning for Complex Networks”. In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 28.9 (2017), pp. 2625–2638. DOI: 10.1109/TPDS.2017.2671868.

[53] Henning Meyerhenke, Peter Sanders, and Christian Schulz. “Partitioning Complex Networks via Size-Constrained Clustering”. In: *13th International Symposium on Experimental Algorithms (SEA)*. Vol. 8504. Lecture Notes in Computer Science. Springer, 2014, pp. 351–363. DOI: 10.1007/978-3-319-07959-2_30.

[54] Mark E. Newman. “Modularity and Community Structure in Networks”. In: *National Academy of Sciences* 103.23 (2006), pp. 8577–8582. DOI: 10.1073/pnas.0601602103.

[55] Mark E. Newman. “The Structure of Scientific Collaboration Networks”. In: *National Academy of Sciences* 98.2 (2001), pp. 404–409. DOI: 10.1073/pnas.98.2.404.

[56] David Pisinger and Paolo Toth. “Knapsack Problems”. In: *Handbook of Combinatorial Optimization* (1998), pp. 299–428.

[57] *Pizza&Chili Corpus (Compressed Indexes and their Testbeds)*. http://pizzachili.dcc.uchile.cl/index.html. Accessed: 2022-11-30.

[58] Jakob Puchinger, Günther R. Raidl, and Ulrich Pferschy. “The Multidimensional Knapsack Problem: Structure and Algorithms”. In: *INFORMS Journal on Computing* 22.2 (2010), pp. 250–265. DOI: 10.1287/ijoc.1090.0344.

[59] Siddharth Samsi et al. “Static Graph Challenge: Subgraph Isomorphism”. In: *IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1–6. DOI: 10.1109/HPEC.2017.8091039.

[60] Peter Sanders and Christian Schulz. “Engineering Multilevel Graph Partitioning Algorithms”. In: *19th European Symposium on Algorithms (ESA)*. Vol. 6942. Lecture Notes in Computer Science. Springer, 2011, pp. 469–480. DOI: 10.1007/978-3-642-23719-5_40.

[61] Peter Sanders and Christian Schulz. “High Quality Graph Partitioning”. In: *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop*. Vol. 588. Contemporary Mathematics. American Mathematical Society, 2012, pp. 1–18.

[62] Sebastian Schlag. “High-Quality Hypergraph Partitioning”. PhD thesis. Karlsruhe Institute of Technology, Germany, 2020.

[63] Sebastian Schlag et al. “k-way Hypergraph Partitioning via n-Level Recursive Bisection”. In: *18th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2016, pp. 53–67. DOI: 10.1137/1.9781611974317.5.

[64] Prabhakant Sinha and Andris A. Zoltners. “The Multiple-Choice Knapsack Problem”. In: *Operations Research* 27.3 (1979), pp. 503–515. DOI: 10.1287/opre.27.3.503.

[65] Alan J. Soper, Chris Walshaw, and Mark Cross. “A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph-Partitioning”. In: *Journal of Global Optimization* 29.2 (2004), pp. 225–241. DOI: 10.1023/B:JOGO.0000042115.44455.f3.

[66] Natarajan Viswanathan et al. “The DAC 2012 Routability-Driven Placement Contest and Benchmark Suite”. In: *49th Design Automation Conference (DAC)*. ACM, 2012, pp. 774–782. DOI: 10.1145/2228360.2228500.

[67] Chris Walshaw. “Multilevel Refinement for Combinatorial Optimisation Problems”. In: *Operations Research* 131.1-4 (2004), pp. 325–372. DOI: 10.1023/B:ANOR.0000039525.80601.15.

[68] Christophe Wilbaut, Said Hanafi, and Said Salhi. “A Survey of Effective Heuristics and their Application to a Variety of Knapsack Problems”. In: *IMA Journal of Management Mathematics* 19.3 (2008), pp. 227–244. DOI: 10.1093/imaman/dpn004.

[69] Min Zhou et al. “Controlling Unstructured Mesh Partitions for Massively Parallel Simulations”. In: *SIAM Journal on Scientific Computing* 32.6 (2010), pp. 3201–3227. DOI: 10.1137/090777323.

A. Visualization of a Star-like Graph

Most star-like instances in our benchmark set are too large to be visualized with common techniques. However, we provide a visualization of `wiki-Vote` (the smallest instance), both for the input graph and coarsened approximations.

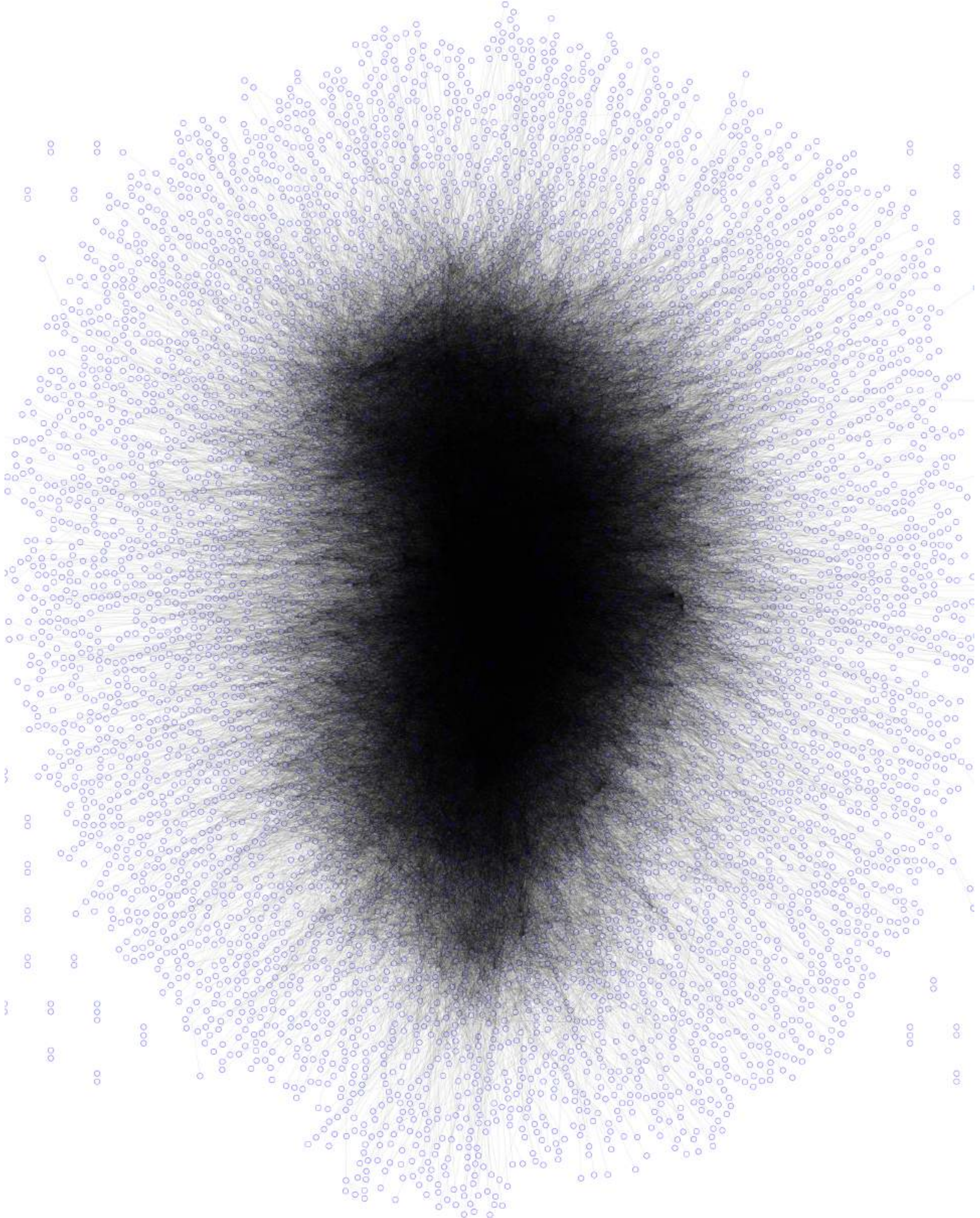


Figure 24: Visualization of the `wiki-Vote` instance.

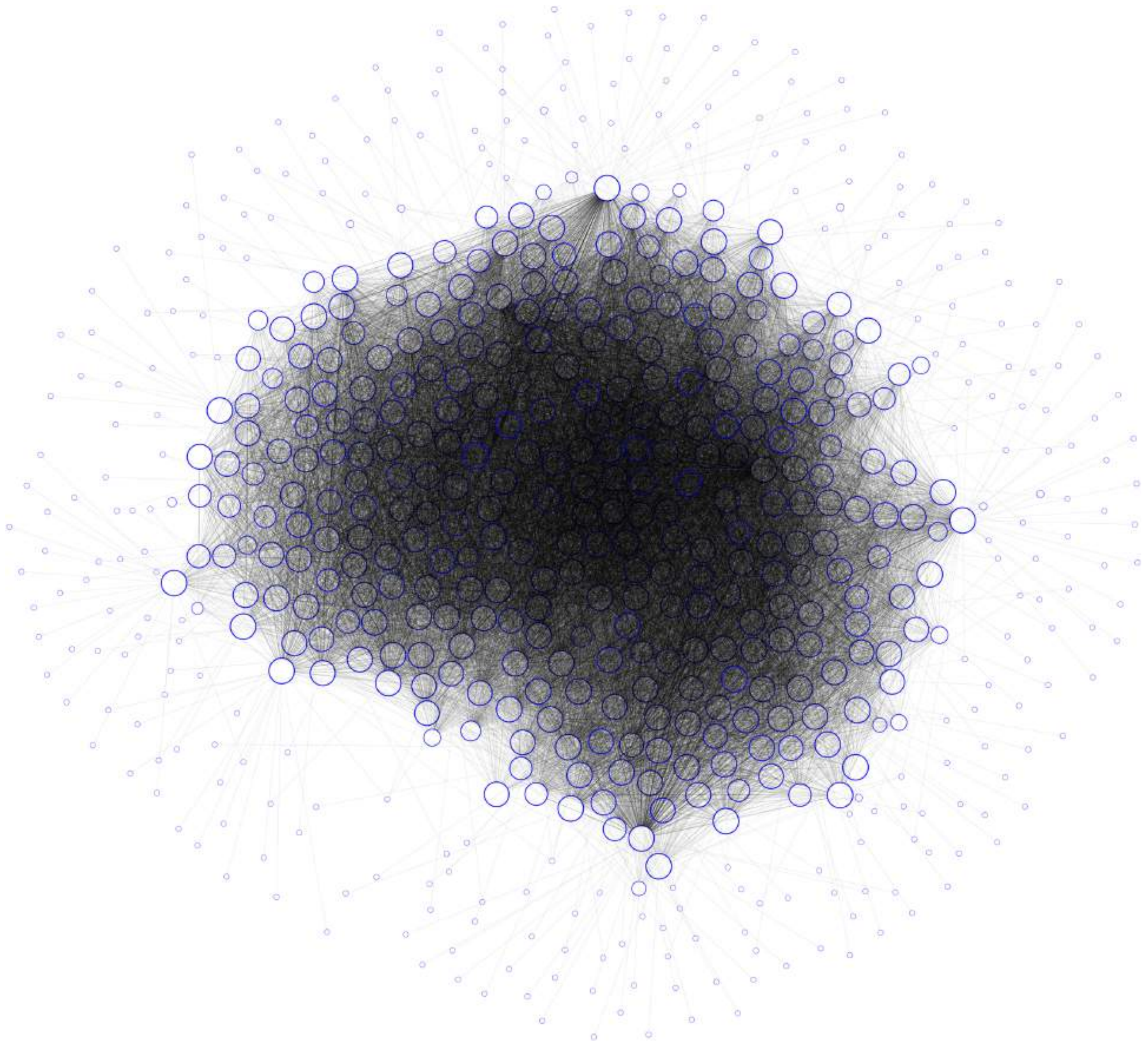


Figure 25: Visualization of the coarsest approximation of `wiki-Vote`, computed with the traditional coarsening techniques used by Mt-KaHyPar.

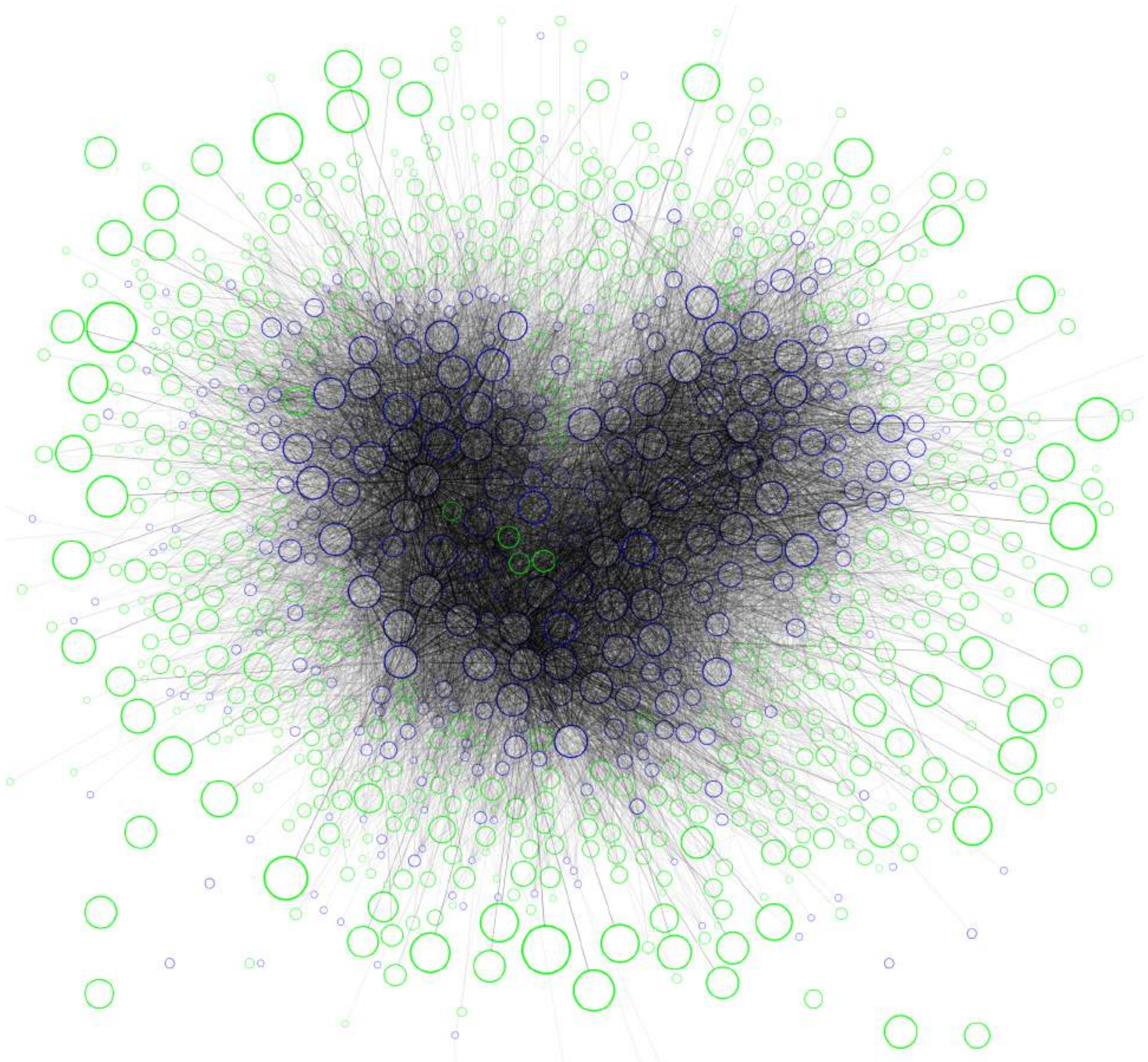


Figure 26: Visualization of the coarsest approximation of `wiki-Vote`, computed with our star partitioning algorithm. That is, traditional coarsening was applied to the core (blue) and two-hop coarsening to the peripheral nodes (green).

B. Detailed Composition of Benchmark Sets

In the following, we list the exact instances used for benchmark set A and benchmark set B including their sizes, type and whether they are considered star-like with regards to the experimental evaluation. Note that all instances are unweighted, except for the recompX graphs that have edge weights.

Graph	Nodes	Edges	Instance Type	Star-like
add32.graph	4960	9462	Walshaw	no
amazon0302.graph	262K	900K	SNAP	no
astro-ph.graph	17K	121K	Scientific	no
bcsstk29.graph	14K	303K	Walshaw	no
belgium.osm.graph	1441K	1550K	Other	no
coAuthorsCiteseer.graph	227K	814K	Scientific	no
cond-mat.graph	17K	48K	Scientific	no
delaunay_n14.graph	16K	49K	delaunayX	no
email-Enron.graph	37K	184K	SNAP	no
fe_sphere.graph	16K	49K	Walshaw	no
hugetrace-00010.graph	12M	18M	Simulation	no
kron_g500-simple-logn16.graph	66K	2456K	kronX	yes
luxembourg.osm.graph	115K	120K	Other	no
recomp_dna1GB_3.graph	2174	66K	recompX	yes
recomp_english1GB_3.graph	12K	316K	recompX	yes
rgg_n_2_15_s0.graph	33K	160K	rggX	no
rhg10.graph	992	3491	rhgX	no
rmat_n16_m22.graph	65K	4194K	rmatX	yes
soc-Epinions1.graph	76K	406K	SNAP	yes
soc-Slashdot0811.graph	77K	469K	SNAP	yes
superblue14.graph	1251K	2049K	DAC	no
superblue19.graph	1034K	1714K	DAC	no
web-Stanford.graph	282K	1993K	SNAP	no
wiki-Vote.graph	7115	101K	SNAP	yes
wing.graph	62K	122K	Walshaw	no

Table 5: Instances of benchmark set A.

Graph	Nodes	Edges	Instance Type	Star-like
amazon-2008.graph	735K	3523K	SNAP	no
amazon.graph	401K	2350K	SNAP	no
arabic-2005.graph	23M	554M	SNAP	no
asia.osm.graph	12M	13M	Other	no
bcsstk32.graph	45K	985K	Walshaw	no
bcsstk33.graph	8738	292K	Walshaw	no
channel.graph	4802K	43M	Simulation	no
citationCiteseer.graph	268K	1157K	Scientific	no
cnr-2000.graph	326K	2739K	SNAP	no
coAuthorsCiteseer.graph	227K	814K	Scientific	no
coAuthorsDBLP.graph	299K	978K	Scientific	no
com-dblp.ungraph.graph	426K	1050K	SNAP	no
com-LiveJournal.graph	3998K	35M	SNAP	no
com-lj.ungraph.graph	4037K	35M	SNAP	no
com-orkut.graph	3072K	117M	SNAP	no
coPapersDBLP.graph	540K	15M	Scientific	no
delaunay_n20.graph	1049K	3146K	delaunayX	no
delaunay_n22.graph	4194K	13M	delaunayX	no
delaunay_n24.graph	17M	50M	delaunayX	no
email-Enron.graph	37K	184K	SNAP	no
enwiki-2013.graph	4207K	92M	SNAP	no
enwiki-2018.graph	5617K	117M	SNAP	no
er-fact1.5-scale20.graph	1049K	11M	erX	no
er-fact1.5-scale21.graph	2097K	23M	erX	no
er-fact1.5-scale22.graph	4194K	48M	erX	no
eswiki-2013.graph	973K	21M	SNAP	yes
eu-2005.graph	863K	16M	SNAP	no
europe.osm.graph	51M	54M	Other	no
germany.osm.graph	12M	12M	Other	no
hollywood-2011.graph	2181K	114M	SNAP	no
hugebubbles-00010.graph	19M	29M	Simulation	no
hugetrace-00010.graph	12M	18M	Simulation	no
hugetric-00010.graph	6593K	9886K	Simulation	no
in-2004.graph	1383K	14M	SNAP	no
indochina-2004.graph	7415K	151M	SNAP	no
kmer_P1a.graph	139M	148M	kmerX	no
kmer_U1a.graph	65M	66M	kmerX	no
kmer_V2a.graph	54M	57M	kmerX	no
kron_g500-simple-logn18.graph	262K	11M	kronX	yes
kron_g500-simple-logn20.graph	1049K	45M	kronX	yes
kron_g500-simple-logn21.graph	2097K	91M	kronX	yes
ljournal-2008.graph	5363K	50M	SNAP	no
netherlands.osm.graph	2217K	2441K	Other	no
nlpkkt120.graph	3542K	47M	Other	no
nlpkkt200.graph	16M	216M	Other	no
packing.graph	2146K	17M	Simulation	no

Table 6: Instances of benchmark set B.

Graph	Nodes	Edges	Instance Type	Star-like
recomp_dna1GB_9.graph	3233K	25M	recompX	yes
recomp_english1GB_5.graph	110K	2552K	recompX	yes
recomp_english1GB_7.graph	802K	13M	recompX	yes
recomp_proteins1GB_7.graph	2826K	44M	recompX	yes
recomp_proteins1GB_9.graph	15M	74M	recompX	yes
recomp_sources1GB_7.graph	899K	7272K	recompX	yes
recomp_sources1GB_9.graph	2792K	12M	recompX	yes
rgg_n_2_20_s0.graph	1049K	6892K	rggX	no
rgg_n_2_22_s0.graph	4194K	30M	rggX	no
rgg_n_2_24_s0.graph	17M	133M	rggX	no
rhg16.graph	62K	259K	rhgX	no
rhg18.graph	242K	987K	rhgX	no
rhg.graph	10M	200M	rhgX	no
rmat_n16_m23.graph	65K	8389K	rmatX	yes
rmat_n16_m24.graph	66K	17M	rmatX	yes
rmat_n25_m28.graph	27M	268M	rmatX	yes
roadNet-CA.graph	1971K	2767K	Other	no
roadNet-PA.graph	1091K	1542K	Other	no
soc-Epinions1.graph	76K	406K	SNAP	yes
soc-Slashdot0811.graph	77K	469K	SNAP	yes
soc-Slashdot0902.graph	82K	504K	SNAP	yes
superblue12.graph	2585K	4774K	DAC	no
superblue16.graph	1396K	2280K	DAC	no
superblue3.graph	1816K	3109K	DAC	no
superblue7.graph	2701K	4931K	DAC	no
twitter-2010.graph	42M	1203M	SNAP	yes
uk-2002.graph	19M	262M	SNAP	no
venturiLevel3.graph	4027K	8054K	SNAP	no
webbase-2001.graph	118M	855M	SNAP	no
web-Google.graph	876K	4322K	SNAP	no
web-NotreDame.graph	326K	1090K	SNAP	no
wiki-Talk.graph	2394K	4660K	SNAP	no
wiki-Vote.graph	7115	101K	SNAP	yes
youtube.graph	1135K	2988K	SNAP	no

Table 7: Instances of benchmark set B (continued).