Karlsruher Institut für Technologie

Bachelor Thesis

Engineering Multi-Constraint Graph Partitioning Algorithms with Unconstrained Refinement

Konstantin von Pückler

Date: 3. Juli 2024

Supervisors: Prof. Dr. Peter Sanders
Dr. Lars Gottesbüren
Nikolai Maas
Daniel Seemaier

Institute of Theoretical Informatics, Algorithm Engineering
Department of Informatics

Karlsruhe Institute of Technology

Abstract

The m-constraint graph partitioning problem divides the vertices of a graph $G = (V, E)$ in k parts so that m balancing constraints are fulfilled and an objective is minimized. A common objective is the edge cut, which is the number of edges having pins in different partitions. So far, the Mt-KaHyPar graph partitioning framework allows graph partitioning only for single-constraint graphs. In this thesis, we extend Mt-KaHyPar to multi-constraint graph partitioning. Multi-constraint graph partitioning has various applications, for instance it is useful to parallelize several graph algorithms such as the page rank algorithm [1]. Contrary to prior research on multi-constraint graph partitioning our algorithm is based on unconstrained refinement [11]. The idea of unconstrained refinement is to allow violation of the balance constraints and apply a rebalancing algorithm afterwards, which ensures that all constraints are fulfilled while partitioning quality is not decreased too much. We then compare our results to the state-of-the-art multi-constraint graph partitioning framework METIS [8] and show that our algorithm outperforms METIS on a wide range of instances.

Das m-Restriktionsgraphpartitionierungsproblem teilt die Knoten eines Graphen $G = (V, E)$ in k Blocks so auf, dass m Ausgleichsbeschränkungen erfüllt sind und ein Zielwert minimiert wird. Ein gängiges Ziel ist der Kantenschnitt, der die Anzahl der Kanten darstellt, deren Endpunkte in verschiedenen Partitionen liegen. Bisher erlaubt das Mt-KaHyPar-Graphpartitionierungs-Framework die Graphpartitionierung nur für Einzelrestriktionsgraphen. In dieser Arbeit erweitern wir Mt-KaHyPar auf die Multi-Restriktions-Graphpartitionierung. Multi-Restriktions-Graphpartitionierung hat verschiedene Anwendungen, zum Beispiel ist sie nützlich, um mehrere Graphalgorithmen wie den PageRank-Algorithmus [1] zu parallelisieren. Im Gegensatz zu früheren Forschungen zur Multi-Restriktions-Graphpartitionierung basiert unser Algorithmus auf unbeschränkter Verfeinerung [11]. Die Idee der unbeschränkten Verfeinerung besteht darin, die Verletzung der Ausgleichsbeschränkungen zuzulassen und anschließend einen Ausgleichsalgorithmus anzuwenden, der sicherstellt, dass alle Beschränkungen erfüllt sind, während versucht wird, die Partitionierungsqualität nicht zu sehr zu verschlechtern. Schlussendlich vergleichen wir unsere Ergebnisse mit dem Multi-Restriktions-Graphpartitionierungs-Framework METIS [8] und zeigen, dass unser Algorithmus METIS auf einer Vielzahl von Instanzen übertrifft.

Acknowledgments

Ich danke meiner Familie.

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

Karlsruhe, den 03.07.2024

Contents

1 Introduction

Graph partitioning is the problem of dividing a graph into k roughly balanced blocks such that a given objective is minimized. This problem has many real-world applications.
For instance, in the design of very large-scale integration (VLSI) systems, graph partitioning can help reduce the connections between circuits. Another important application is the parallelization of graph algorithms by partitioning the graph into several blocks, which can then be processed in parallel by multiple worker nodes. It is desirable to have equally loaded blocks, as the performance of parallel execution depends on the performance of the slowest worker. It is also desirable to have largely independent blocks, as this allows the workers to perform their calculations relatively independently, thereby reducing communication overhead.
One example of such a parallelizable graph algorithm is the PageRank algorithm. PageRank is used in modern search engines to estimate the importance of web pages based on the number and quality of links to other pages. Experiments show that when parallelizing the PageRank algorithm, the workload of a worker node depends not only on the number of nodes assigned to it but also on the number of edges incident to those nodes.
Therefore, to balance the workload between worker nodes our partition must satisfy two balance constraints, the number of vertices and the number of edges. This problem is referred to as multi-constraint graph partitioning.

1.1 Contribution

So far, research on multi-constraint graph partitioning has focused on constrained refinement. However, on single-constraint graphs unconstrained refinement outperforms constrained refinement due to its ability to overcome local minima. In this thesis we evaluate whether unconstrained refinement can outperform constrained refinement also on multi-constraint graphs. Unconstrained refinement is based on temporarily allowing imbalance for refinement and rebalancing the partition afterwards. Since the load balancing problem becomes much more difficult with multiple constraints, the main challenge of performing unconstrained refinement on multi-constraint graphs is to find a powerful rebalancer.
In this thesis, we extend the Mt-KaHyPar framework to allow for the partitioning of multi-constraint graphs. We propose an algorithm based on unconstrained refinement and rebalancing. While our refinement algorithm is inspired by the unconstrained refinement algorithm of METIS [8], we focus on designing a rebalancing algorithm which allows to

rebalance partitions without worsening partitioning quality too much. For instances which are hard to balance, we then propose further techniques to reach a balanced solution.
We evaluated our algorithm on a large benchmark set and observed a large improvement in solution quality compared to the state-of-the-art multi-constraint graph partitioner METIS [8].

1.2 Structure of Thesis

In Chapter 2, we present general definitions and notations that are used throughout the rest of this thesis and give a more formal introduction to multi-constraint graph partitioning and imbalance metrics. Subsequently, we give an overview over existing research upon unconstrained refinement and multi-constraint graph partitioning in Chapter 3. In Chapter 4, our rebalancing algorithms are described before we give an experimental evaluation of them in Chapter 5.

2 Fundamentals

In this chapter we introduce general definitions and notations that are used in this thesis. Moreover, the multi-constraint graph partitioning problem is introduced formally.
If not stated differently, arrays are 1-indexed. When iterating over a range of numbers, we will often use the notation of $[x] \coloneqq \{1, .., x\}$.

2.1 Graph Theory

An undirected m-constraint *graph* $G = (V, E, c, \omega)$ consists of a finite set V and a binary relation on V, $E \subset \{\{u, v\} \mid u, v \in V,\ u \neq v\}$. The elements of V are called *nodes* or *vertices* and the elements of E are called *edges*. The node weight function $c : V \to \mathbb{R}^m_{\geq 0}$ with $c(V) = 1^m$ assigns m weights to each node while the edge weight function $\omega : E \to \mathbb{N}_0$ assigns a weight to every edge. The d-th node weight of a node v is denoted as $c(v)[d]$. We call it the weight of v in *dimension* d. We call a m-constraint graph with $m > 1$ a *multi-constraint* graph.
A *k-way partition* $\Pi = \{V_1, .., V_k\}$ of a graph G is a partition of the node set V into k disjoint blocks. A 2-way partition is also called *bisection*. The block V_i of a node v is denoted as $\Pi(v)$. For a node v and a set of nodes W we define $\omega(v, W) \coloneqq \sum_{\substack{\{v,w\} \in E \\ w \in W}} \omega(\{v, w\})$. We call $\omega(v, \Pi(v))$ the *internal edge weight* and $\omega(v, V \setminus \Pi(v))$ the *external edge weight* of v. The *boundary* is the set of all nodes with positive external edge weight. We call changing $\Pi = \{V_1, .., V_k\}$ to $\Pi' = \{V_1, .., V_i \setminus \{v\}, .., V_j \cup v, .., V_k\}$ a *move* of v from V_i to V_j. A move is denoted as m, (v, V_i, V_j) or (Π, Π').

2.2 Multi-Constraint Graph Partitioning

The *multi-constraint balanced graph partitioning problem* is to find a k-way partition Π of a graph G that minimizes an objective function defined on the edges while each block $V' \in \Pi$ satisfies the balance constraints $c(V')[d] \leq \left\lceil \frac{1+\epsilon}{k} \right\rceil$ for each $d \in [m]$ for some imbalance ratio $\epsilon \in (0, 1)$. We call $c_{max} = \left\lceil \frac{1+\epsilon}{k} \right\rceil$ the max-part weight.
As the objective function to minimize, we will focus on the *edge cut*. The *edge cut*, also denoted as cut, is the total weight of all edges running between different blocks, i.e.

$cut(\Pi) := \sum_{\substack{\{u,v\}\in E \\ \Pi(u)\neq\Pi(v)}} c(\{u,v\})$. The *gain* of a move m is the improvement in edge cut it induces. We denote it by $gain(m, \Pi)$.
The violation of the balance constraints is called *imbalance*. A moves *balance gain* quantifies the improvement in imbalance it induces. Since it is not clear how to measure a partitions imbalance, there are several possible definitions of balance gain. The algorithm proposed in this thesis uses the *L1-norm balance gain*. The *L1-norm imbalance* describes the total exceed of balance constraints across all blocks. It is defined as $ib_{L1}(\{V_1, ..., V_k\}) := \sum_{i\in[k]} \sum_{d\in[m]} \max\{0, c(V_i)[d] - c_{max}\}$. A moves L1-norm balance gain is the improvement in L1-imbalance it induces.
Moreover, we introduce some common metrics used to describe an m-dimensional weight w. For a weight w we define its *highest weight* as $\max(w) := \max_{d\in[m]}\{w[d]\}$, its *lowest weight* as $\min(w) := \min_{d\in[m]}\{w[d]\}$, its 1-norm weight as $||w||_1 := \sum_{d\in[m]} w[d]$ and its *average* as $avg(w) := \frac{1}{k} \cdot \sum_{d\in[m]} w[d]$. We then define the *vertical imbalance* of a weight w as $ib_{vert}(w) := \frac{1}{k} \cdot \sum_{d\in[m]} (w[d] - avg(w))^2$. The vertical imbalance measures how much the weights of w differ from each other. We denote $|w|$ as the standard norm of a vector.
With these definitions, we can define another metric to measure the imbalance of a partition. For vectors v_1 and v_2 we consider $v_1 < v_2$ if $(v_1[1], .., v_1[i]) = (v_2[1], .., v_2[i])$ and $v_1[i+1] < v_2[i+1]$. The *max-norm imbalance* ib_{max} of a partition is a tuple (x, y) where x and y each are vectors of size k. x contains the highest weight of each block ordered by size, e.g. $x := (\max(c(V_{i_1})), ..., \max(c(V_{i_k})))$ with $\max(c(V_{i_1})) \geq, ..., \geq \max(c(V_{i_k}))$. y contains the norm of each blockweight ordered by size, e.g. $y := (|c(V_{j_1})|, ..., |c(V_{j_k})|)$ with $|c(V_{j_1})| \geq, ..., \geq |c(V_{j_k})|$. For max-norm imbalances (x, y), (x', y'), we consider $(x, y) < (x', y')$ if $x < x'$ or $x = x'$ and $y < y'$.
Note that in contrast to L1-norm imbalance, max-norm imbalance does not ignore blockweights which do not violate balance constraints.
A moves *max-norm balance gain* is then defined as the improvement in max-norm imbalance it induces.

3 Related Work

The single-constraint graph partitioning problem has been studied extensively. Most research is based on the multilevel graph partitioning paradigm and local search heuristics. We introduce these schemes in this chapter and describe how they are applied in the single-constraint graph partitioning algorithms of the Mt-KaHyPar framework. Subsequently, we give an overview of existing research upon the multi-constraint graph partitioning problem and related problems.

3.1 Single-Constraint Graph Partitioning

Before we regard the multi-constraint graph partitioning problem, we will give an overview of existing research upon single-constraint graph partitioning, as single-constraint graph partitioning is a special case of multi-constraint graph partitioning.

3.1.1 Multilevel Graph Partitioning

Most graph partitioning softwares, including Mt-KaHyPar, PaToH and METIS, use a multilevel graph partitioning scheme described by Hendrickson and Leland in 1995 [7]. The scheme partitions a graph in three phases: Coarsening, initial partitioning and unconarsening.

Coarsening. The Coarsening phase produces a hierarchy of smaller graphs $G_1, .., G_N$, where $G_1 = G$ and G_{i+1} is obtained from G_i by a set of *contractions*. In each step a *clustering* is computed. A clustering assigns each vertex to a cluster. Subsequently, the vertices of a cluster are *contracted*. Contracting a cluster $C = \{v_1, ..., v_n\}$ means replacing $\{v_1, ..., v_n\}$ through a single new vertex v' with $c(v') = \sum_{1 \leq i \leq n} c(v_i)$. Edges $\{v, w\}$ with $v, w \in C$ are deleted, edges $\{v, w\}$ with $v \in C, w \notin C$ are replaced by an edge $\{v', w\}$. When this procedure produces parallel edges $e_1, ..., e_n$, they are removed and replaced by a single edge e' with $\omega(e') = \sum_{1 \leq i \leq n} \omega(e_i)$.
The coarsening process is aborted when the total number of nodes falls below a certain threshold. G_1 is called the *finest* graph, G_N is called the *coarsest* graph.
The main challenge of coarsening algorithms is to compute clusterings which lead to a

graph G_N for which a high-quality partitioning can be found in the *initial partitioning*-phase.

Initial Partitioning. The initial partitioning computes a k-way partitioning for the coarsest graph. Since the coarsest graph is reasonably small, the initial partitioning cares more about quality than about runtime. Most initial partitioning algorithms use a *recursive bi-partitioning*-scheme. The graph is recursively separated into two equally loaded parts until k partitions are reached.

Uncoarsening. Finally, starting from the initial partition of the coarsest graph G_N, the graph is successively projected onto the next finer level. After each projection step, a local search algorithm is performed in order to improve the objective. In that way, the local search goes from a more global view to a more local way, which helps to avoid local minima in partitioning quality.

3.1.2 Local Search

At each uncoarsening level a local search is performed. Generally, we differentiate between two local search schemes: *Constrained* local search tries to improve the quality of a partition without allowing a temporary increase in imbalance. *Unconstrained* local search allows temporary imbalance in order to achieve higher quality improvements. For both constrained and unconstrained local search, algorithms optimizing partitioning quality are called *refinement* algorithms. Algorithms reducing the imbalance are called *rebalancing* algorithms. Rebalancing algorithms are only performed if the partition is imbalanced. Constrained local search typically successively performs several rounds of refinement and several rounds of rebalancing. Note that rebalancing rounds are only performed if the partitioning was imbalanced previous to local search. Unconstrained local search on the other hand alternates between refinement rounds and rebalancing rounds. In the following, we present algorithms commonly used for refinement.

Fiduccia-Mattheyses Algorithm. The Fiduccia-Mattheyses (FM) algorithm was first proposed by Fiduccia and Mattheyses [4] as a local search algorithm for 2–way partitions. Karypis and Kumar [9] developed a modified version of that algorithm which can be used for k-way partitions. It works as follows:
First, all boundary nodes are inserted into a priority queue with their gain values of their best moves as keys. Next, the algorithm retrieves the node v with the highest gain from the priority queue and tries to move the node to its corresponding block. If the movement would violate a balance constraint, it is skipped. Otherwise, the movement is performed and the gain values of other nodes that are affected by the movement and that still reside

in the priority queue are updated. If neighbors of v become boundary nodes due to the movement, they are inserted into the priority queue (but only if they were not previously moved).After a certain number of nodes were moved without improving the partition, the search is stopped and all movements are reverted until the best partition that occurred during the process is restored.

Label Propagation. Label propagation visits the nodes in random order. When a node is traversed, the block with best gain is determined. If there are more than 1 blocks with best gain, balance is used as tie-break. A node is only moved if the best move improved gain or does not change gain and improves balance.

3.1.3 Mt-KaHyPar

Mt-KaHyPar [6] is a multi-threaded partitioner for single-constraint (hyper-)graphs. Since this thesis focuses on graph partitioning, we describe how Mt-KaHyPar works on graphs. It uses multilevel partitioning in combination with unconstrained local search.
Mt-KaHyPar computes a coarsening hierarchy via parallelized label propagation. The goal is to minimize the remaining edgeweight while asserting that no clustering becomes too heavy. For initial partitioning, Mt-KaHyPar performs several runs of a set of randomized algorithms and takes the best result. In the following, we will discuss the refinement and rebalancing algorithms of Mt-KaHyPar more detailed.
Mt-KaHyPar [11] uses two different refinement algorithms. The first is a parallelized version of the FM algorithm. Differently to classical FM it is unconstrained, i.e. it does not forbid moves worsening imbalance. Instead, a penalty is added to every move m which increases imbalance. The penalty affects both the priority a node has in the priority queue and the block it is moved to. The penalty tries to estimate how much partitioning quality it will cost to rebalance m. Therefore, at the beginning of the FM nodes are per block inserted into exponentially spaced buckets based on the ratio between their internal edge weight and their weight. When FM then considers a move of v to an imbalanced block i, based on these buckets it can be estimated how much cut gain it will cost to remove additional $c(v)$ nodeweight from block i. Note that this approach is only feasible in single-constraint case since in the multi-constraint case the cut worsening it takes to rebalance an overloaded block does highly depend on the distribution of the blocks weights.
The second refinement algorithm is an unconstrained version of parallel label propagation. Constrained label propagation often suffers from local minima since it does not, contrary to FM, allow moves with negative gain. Unconstrained label propagation mitigates this drawback by allowing to overcome local minima via temporary imbalance.
The rebalancing algorithm consists of two phases (insertion and moving). In the insertion phase, all nodes in overloaded blocks are inserted into a concurrent priority queue. During rebalancing, only moves into non-overloaded blocks are permitted. In the moving phase, nodes are extracted from the priority queue and moved to their preferred target block.

Nodes that are not connected to a valid target block are moved to the block with lowest weight. For a node u, its move m with the best gain that does not violate the balance of the target is considered. The priority of u then is defined as $\frac{gain(m,\Pi)}{c(u)}$ if $gain(m,\Pi) < 0$, what is the supposed case and $gain(m,\Pi) \cdot c(u)$, if $gain(m,\Pi) \geq 0$. This priority combines the two goals of progressing quickly to a balanced state and not worsening the edgecut more than necessary. Once a block is no longer overloaded, its nodes are skipped when they are extracted from the priority queue. The algorithm stops as soon as all blocks are balanced.

3.2 Multi-Constraint Graph Partitioning

To our knowledge, so far there have been two approaches explored to solve the multi-constraint graph partitioning problem. While the graph partitioning frameworks PaToH [2] and METIS [8] rely on the multilevel scheme in combination with constrained refinement, Avdiukhin et al. [1] propose an approach based on projected gradient descent. In the following, we will cover existing research on these two approaches. Finally, we give an overview of existing approaches to solve the multi-capacity bin packing problem, as it is closely related to the load-balance problem of multi-constraint graphs.

3.2.1 Multi-Constraint PatoH

PaToH [2] is a multi-threaded (hyper-)graph partitioner which also supports multi-constraint graphs. It uses multilevel partitioning in combination with constrained refinement.
PaToH uses an edge-cut optimized label propagation for coarsening. For initial partitioning, a greedy region growing algorithm is performed several times where each time a starting node is selected randomly. Afterwards, the best result is selected as initial partition. Finally an FM is used for refinement. To handle multiple constraints, balance conditions are simply extended to multiple dimensions. Since PaToH does nothing which is suitable for multi-constraint graphs in particular, we will rather focus on the following multi-constraint graph partitioner called METIS.

3.2.2 Multi-Constraint METIS

METIS [8] is a single-threaded partitioner for multi-constraint graphs which uses multilevel partitioning in combination with constrained refinement.

Coarsening. METIS coarsening computes the clustering by using edge weight reduction as primary criterion and low vertical imbalance of cluster weights as tiebreak criterion. METIS does not consider clusters which contain more than two nodes. The intuition behind using vertical imbalance of cluster weights as tiebreak criterion is that the multi-constraint

load balancing problem becomes easier when nodes are less vertically imbalanced, since in case $c(v)[i] = c(v)[j]$ for all $v \in V, i, j \in [m]$ the multi-constraint load balancing problem equals the single-constraint load balancing problem.

Initial Partitioning. METIS follows the recursive bisection scheme. In order to bisect $G = (V, E)$ in $G_A = (V_A, E_A)$ and $G_B = (V_B, E_B)$, a bisection starts with selecting a vertex v randomly and assigning $V_A = v$ and $V_B = v \setminus \{v\}$. A *region growing*-algorithm then successively moves nodes from V_B to V_A. It holds m priority queues with where a vertex $v \in V_B$ is inserted into the $max_i\{c(v)\}$-th queue, e.g. the j-th queue consists of the nodes from V_B whose weight is maximal in dimension j.The priority of a node v is determined as $\frac{\omega(v, V \setminus \Pi(v)))}{\sqrt{|\{V_i \in \Pi | N(v) \cap V_i \neq \emptyset\}|}} - \omega(v, \Pi(v))$. In each step the region growing algorithm then selects the top vertex from the $max_i\{c(V_B\}$-th queue and moves it to V_A. In case the $max_i\{c(V_B\}$-th queue is empty, the non-empty queue corresponding to the next heavier dimension is selected. The algorithm terminated as soon as G_A is heavier than G_B in at least one dimension. Then a modified 2-way FM is performed. Nodes are successively moved between G_A and G_B, while no node is allowed to be moved more than once. G_A and G_B each hold m queues, where a node $v \in V_x$ is inserted into the $max_i\{c(v)\}$-th queue of G_x similarly to the region growing algorithm. Before each move then the block to move from is determined. If the partition is imbalanced, we compute the highest imbalance across both parts and all dimensions. If $c(V_A)[d]$ has the highest imbalance, then the d-th queue of G_A is selected. In case it is empty, we choose the next greatest imbalanced dimension from G_A, until we find a non-empty queue. If the partitioning is not imbalanced, we iterate over all queues and choose that one with the highest priority node.
The best solution is tracked over the course of the algorithm, finally moves are rolled back to this solution. FM is performed first in rebalancing mode and then in refinement mode. They differ in what they consider the best solution. While rebalancing uses imbalance as main criterion and partitioning quality as tiebreak, refinement allows imbalance to a certain factor and uses partitioning quality as main criterion and imbalance as tiebreak.

Local Search. METIS uses constrained label propagation for refinement and rebalancing. Initially, $c'_{max}[d] := \max\{c_{max}, \max_{V_i \in \Pi}\{c(V_i)[d]\}\}$ is computed for each constraint d. Throughout the local search, METIS only allows moves that do not increase a blocks weight above c'_{max}. Note that this can still allow moves increasing a blockweight above c_{max}. We now describe refinement and rebalancing algorithms of METIS.
Several refinement passes are performed. A refinement pass starts by inserting all nodes with $\omega(v, \Pi(v)) \geq \omega(v, V \setminus \Pi(v))$ into a priority queue. The same approximation as in the initial partitioning algorithm is used as priority. Then subsequently nodes are retrieved from the queue. When a node is retrieved, its best move not violating the c'_{max} constraints is determined with cut gain as primary objective and max-norm imbalance gain as tiebreak. If there exists no move leading to an improvement, no move is performed. After a node was moved, the priorities of its neighbors are updated. Neighbors

with $\omega(v, \Pi(v)) \geq \omega(v, V \setminus \Pi(v))$ which are not yet in the queue are inserted into the queue if they were not moved yet. Then the next node is retrieved. A refinement pass stops if the queue is empty. Refinement stops if a certain number of passes was performed or the last pass did not make any progress.
Rebalancing works analogously to refinement. Instead of nodes with $\omega(v, \Pi(v)) \geq \omega(v, V \setminus \Pi(v))$, all boundary nodes are inserted into the queue. For moves, max-norm imbalance is chosen as primary objective and cut gain as tiebreak. On the coarser graphs, rebalancing is only performed if the partition is very imbalanced. Otherwise, rebalancing is only performed on the finest graph.
For both refinement and rebalancing, METIS uses a slight randomness by inserting nodes into the priority queues in random order. This means nodes with same priority are retrieved in random order.

3.2.3 Projected Gradient Descent(GD)

Avdiukhin et al. [1] propose an approach based on randomized projected gradient descent to solve the multi-constraint graph partitioning problem for graphs with uniform edgeweights. At first, we describe their approach th compute a 2-way partition, subsequently we show, how they generalize to k-way partitioning. They compute a partitioning $\Pi = \{V_1, V_{-1}\}$ by introducing variables $x_v \in \{-1, 1\}$ indicating if vertex v is in V_1 or V_{-1} and solving the following quadratic integer program:

$$Maximize: \frac{1}{2} \cdot \sum_{(v,w) \in E} (x_v \cdot x_w + 1)$$
$$Subject\,to: |\sum_{v \in V} c(v)[d] \cdot x_v| \leq \epsilon \cdot \sum_{v \in V} c(v)[d] \qquad \forall 0 \leq d < m.$$

Maximizing the defined objective corresponds to minimizing the edge-cut. To see that, observe that an edge (v, w) makes a contribution of 1 to the objective if $x_v = x_w$ (and thus $x_v \cdot x_w = 1$) and 0, otherwise. So $\frac{1}{2} \cdot \sum_{(v,w) \in E} x_v \cdot x_w + 1 = |E| - edgecut(\Pi)$.
Note that in case of graphs with non-uniform edgeweights, the objective could easily be adapted $\frac{1}{2} \cdot \sum_{\{v,w\} \in E} \omega(\{v, w\}) \cdot (x) \cdot x_w + 1)$
The integer quadratic program's constraints are equivalent to $-\epsilon \cdot c(V)[d] \leq c(V_1)[d] - c(V_{-1})[d] \leq \epsilon \cdot c(V)[d]$. Adding or subtracting $c(V)[d]$ to both sides and dividing by 2 we have $c(V_i)[d] \leq \frac{1+\epsilon}{2} \cdot c(V)[d]$, which corresponds to the d-th balance constraint.
They solve a relaxed version of this problem by allowing $x_v \in [-1, 1]$, and afterwards assign the vertices to V_1, V_{-1} via randomized rounding with $\mathbb{P}(v \in V_1) = \frac{1+x_v}{2}$ and $\mathbb{P}(v \in V_{-1}) = \frac{1-x_v}{2}$. In expectation, this does not change the objective and all balance constraints are still preserved with high probability.
The algorithm iteratively computes a sequence of vectors $\{x^{(t)}\}$, where $x^{(t)} \in [-1, 1]^{|V|}$. Each iteration consists of a gradient descent step, which optimizes the objective, and a projection step, which projects the found solution onto the space of balanced solutions.

Gradient descent step:
The objective can be expressed as $obj(x) = x^T Ax$ with the graph adjacency matrix A. Since $\nabla^2(obj) = A$, a gradient descent step with stepsize γ corresponds to $y^{(t)} = (\mathbb{I} + \gamma \cdot A) \cdot x^{(t)}$.
Projection step:
The space of feasible solutions K is defined as $K = B_\infty \cap \bigcap_{j=0}^{m-1} S_\epsilon^j$ with $B_\infty = [-1, 1]^{|V|}$ and $S_\epsilon^j = \{x \in \mathbb{R}^{|V|} \,|\, |\sum_{v \in V} c(v)[d] \cdot x_v| \leq \epsilon \cdot \sum_{v \in V} c(v)[d]$. The projection step tries to find $x^{(i+1)} = argmin_{x \in K}\{x - y^{(t)}\}$. Since exact projection is computationally expensive, Avdiukhin et al. use several approximations. For further detail, see [1].

3.2.4 Multi-Capacity Bin Packing

Finally, we introduce a problem which is closely linked to the multi-constraint load balancing problem which is object to our rebalancing algorithm.
The *multi-capacity bin packing problem* can be defined as follows:
A capacity vector $C = (C_1, ..., C_m)$ and n items $(X_1, ..., X_n)$, where each item consists of m weights $c(X_i) = (c(X_i)_1, ..., c(X_i)_m)$, are given. The goal now is to pack $(X_1, ..., X_n)$ into bins $B_1, ..., B_k$ with minimal k, so that $\sum_{X_i \in B_j} c(X_i) \leq C$.
The single-capacity bin packing problem (e.g. m=1) has been studied extensively. We give a short overview of the most common heuristics:

Next Fit. Next-Fit iterates over all items. When selecting an item, next fit tries to insert it into the last bin B_k. If it does not fit, a new bin B_{k+1} is created.

First Fit. First-Fit proceeds similar to next fit, except that when selecting an item X_i, it traverses all $B_1, ..., B_k$ to find the smallest j so that X_i fits into B_j. In case X_j fits in none of the bins, a new bin is created.

Best Fit. Best-Fit selects a further bin selection heuristic to First-Fit. Best-Fit places the next item into the bin in which it leaves the least total empty space.
To all of these heuristics, the order in which the items are traversed is crucial. The heuristics tend to perform better, if items are sorted to a non-increasing sequence.
These single-capacity strategies can easily be extended to the multi-capacity case. To sort the items, different heuristics are reasonable, for instance sorting after the sum of weights or sorting after the sum of the squares of the weights.
Leinberger et al. [10] propose more sophisticated strategies to solve multi-capacity bin packing. Instead of taking an item and trying to fit it into all bins B_i, as Best-Fit does, they

take a bin B_j and iterate over all items. Their goal is to balance B_j, e.g. to make all its weights equally loaded. Therefore, when B_j has weights $(c(B_j)_1, ..., c(B_j)_m)$ with order $(i_1, .., i_m)$, i.e. $c(B_j)_{i_1} \geq ... \geq c(B_j)_{i_m}$, they search for an item X with order $(i_m, .., i_1)$. When there are no items left with this exact order, they search for items in the next nearest order, until they find an order with items left. Since the search costs can be in $O(m!)$, they propose several relaxations for the selection heuristic:

Permutation Pack. The Permutation Pack heuristic with window size w relaxes the order requirement by only demanding that the d largest weights of an element match the required order, i.e. $c(X)_{i_m} \geq ... \geq c(X)_{i_{m-d+1}} \geq c(X)_{i_{m-d}}, ..., c(X)_{i_1}$

Choose Pack. The Choose Pack heuristic with window size is a further relaxation. When searching for an item, it only requires that the set of the items d largest weights equals the set of the bins d smallest weights, i.e. $c(X)_{i_m}, ..., c(X)_{i_{m-d+1}} \geq c(X)_{i_{m-d}}, ..., c(X)_{i_1}$ Leinberger et al. noticed that both Permutation Pack and Choose Pack outperformed classical First Fit heuristics, while a large percentage of the performance gains was already achieved by a small window size ($w \leq 2$). For further detail, see [1].

4 Greedy Unconstrained Local Search for Multi-Constraint Graph Partitioning

In single-constraint graph partitioning, unconstrained local search tends to outperform constrained local search. In this section, we propose an unconstrained local search algorithm 1 for multi-constraint graph partitioning. Similarly to existing single-constraint unconstrained local search algorithms [11], we alternate between refinement rounds, which optimize partitioning quality, and rebalancing rounds, which restore balance. We start with a rebalancing phase. Afterwards, we perform several unconstrained rounds, where an unconstrained round consists of a refinement round followed by a rebalancing round. After each unconstrained round, in case the round either did worsen partition quality while not decreasing the maximal balance constraint violation or it did increase the maximal balance constraint violation, we revert the moves performed by the unconstrained round.
For refinement, we use a modified version of label-propagation based multi-constraint METIS refinement (3.2.2). Unlike METIS, we allow a certain imbalance, i.e. we multiply the maximum allowed block weight by a factor γ. There are several characteristics of the METIS refinement algorithm which can be varied, for example priority for nodes and tie-break for moves. The refinement configurations we tested are described in 5.3.
Existing research on unconstrained local search algorithms focuses on single-constraint graph partitioning. Similarly to existing rebalancing algorithms for unconstrained local search, our rebalancing algorithm tries to satisfy the balance constraints by greedily moving vertices. However, unlike in the single-constraint case we have to deal with *deadlock*-scenarios, where no more greedy moves leading to progress are available while the partition is still imbalanced. We will see an example for a deadlock scenario later on. We will propose several techniques to resolve such deadlocks.
Another common problem of unconstrained refinement even in the single-constraint case is that when performing a move which worsens the balance, we do not know how much quality it will cost to restore balance. Single-constraint refinement algorithms such as implemented in Mt-KaHyPar use penalty-estimation heuristics as described in Section 3.1.3, which probably are not transferable to the multi-constraint case. Thus, we have to deal with the case that our refinement algorithm moves vertices which should not be moved in order to accomplish a quality improvement. Especially when a refinement round was *unsuccessfull*, i.e. it was rolled back, we want to avoid that in the next refinement round the same

Algorithm 1: Unconstrained Local Search

```
Data: partition Π
Π_current ← rebalance(Π)
Π_best ← Π_current
for i = 1 to numIterations do
    ib ← ib_max(Π)
    c ← cut(Π)
    Π_current ← refine(Π_current)
    Π_current ← rebalance(Π_current)
    c_new ← cut(Π)
    ib_new ← ib_max(Π) if ib_new < ib or ib_new = ib and c_new < c then
        Π_best ← Π_current
    else
        Π_current ← Π_best
return Π_best
```

moves are performed again. We do this by locking vertices. Vertex locking in the context of unconstrained refinement was first proposed in the *Jet* algorithm [5]. In Jet, vertices which are moved by the refinement algorithm are locked and cannot be moved by the refinement algorithm in the next round. The rebalancing algorithm is allowed to move locked vertices. We use a slightly modified locking. We lock vertices until the next successfull refinement round, e.g. the refiner is only allowed to move vertices which have been not been moved since the start of the last successfull refinement round.

4.1 Rebalancing Algorithm

Next, we discuss our rebalancing algorithm 2, which greedily chooses moves based on cut-gain and L1-gain.Our rebalancing only considers moves which either improve L1-imbalance or do not change L1-imbalance and improve cut-gain. Our algorithm computes the value of a valid move in the following way:

$$val(m) = \begin{cases} gain_{cut}(m) \cdot gain_{L1}(m) & gain_{cut} \geq 0 \\ \frac{gain_{cut}(m)}{gain_{L1}(m)} & gain_{cut} < 0 \end{cases}$$

This case distinction combines the objectives of progressing to a balanced solution fast and not worsening the cut more than required.
Initially, vertices with valid moves are inserted into a max-priority queue. The priority of a vertex v is defined as the value of the highest-valued valid move of v. When moving a vertex, we perform the move with highest value.

Algorithm 2: Greedy Rebalancer

```
Data: Π
queue ← initializeQueue()
while imbalanced and queue.notEmpty() do
    topnode, priority ← queue.getMax()
    topMove ← bestMove(topnode)
    if bestMove.priority ≥ priority then
        Π ← executeMove(topMove, Π)
        updateNeighbors()
        movesPerformed++
        if movesPerformed = updatePeriod or queue.isEmpty() then
            totalUpdate()
    else
        if hasValidMove(topnode) then
            queue.changeKey(topnode)
        else
            queue.delete(topnode)
return Π
```

After a node was moved, the gain values of its neighbors might have changed. Thus, we need to update the priority queue. The priority of a vertex v could change if $gain_{cut}((v, V_j))$ or $gain_{L1}((v, V_j))$ changes for a $V_j \in \Pi$. We treat those cases separately.
A move of a vertex v can affect cut-gains only for $w \in N(v)$. Since neighborhood sizes are not too large for most vertices, performing neighborhood updates after every move is feasible.
However, after a move of v all other vertices could have changed L1-gains, so updating all of them after every move is not feasible. We observe that vertices whose priority has become worse can be updated lazily. That means, for one of those vertices we just have to perform an update if it is on top of the queue.
For vertices whose priority has improved it is not sufficient to perform lazy updates. Therefore, we periodically update all vertices after a certain number of moves was performed. To find a feasible frequency for periodic updates, we try to estimate the number of moves our rebalancer will need as $exp(\{V_1, ..., V_k\}) = \sum_{i \in [k]} \max_{d \in [m]} \{\frac{\max\{0, c(V_i) - c_{max}\}}{|V_i|}\}$. We then perform periodic updates every $\frac{1}{f} \cdot exp$ moves for a constant $f \geq 1$. Every time the number of performed moves surpasses exp, we adapt exp to $2 \cdot exp$. By that we assure feasible runtime even if our estimation initially was very imprecise.

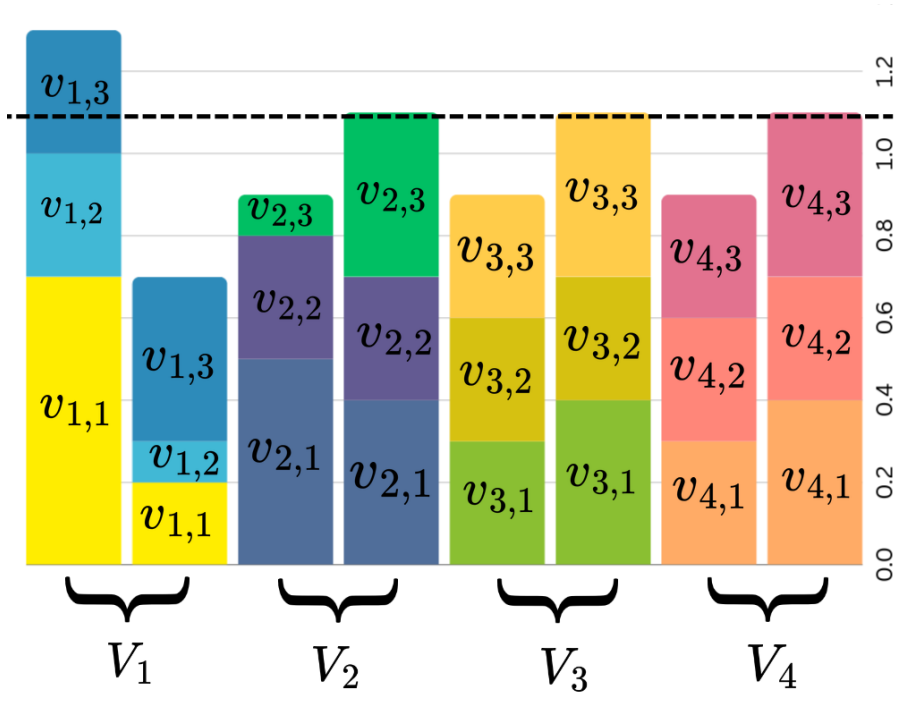

Figure 4.1: deadlocked partition $\{V_1, V_2, V_3, V_4\}$ with 2 constraints and $\epsilon = 0.1$

4.2 Deadlock Handling

The greedy rebalancer can become deadlocked, i.e. there are no more positive moves although the partition is still imbalanced. Figure 4.1 shows an example of a deadlocked partition.

As we can see, every move would increase the L1-imbalance. However, swapping $v_{1,3}$ and $v_{2,3}$ would balance the partition. Typically deadlocks occur when one block is imbalanced in one dimension and has very low weight in another dimension. Occurrence of deadlocks is the main drawback of using L1-norm for balance gains. While deadlocks also occur using other imbalance metrics like max-norm imbalance, L1-imbalance is particularly prone to this phenomenon as it does not regard how evenly distributed the weights which do not violate the balance constraints are. For example, in Figure 4.1 the second constraint is fully balanced, but the block weights in the second dimension are very unevenly distributed, which leads to the deadlock.

Note that if a deadlock once occurred, switching to other metrics like max-norm imbalance does not help in most cases. However, it might be possible to overcome deadlocks by allowing vertex swaps. A vertex swap then is considered legal if it leads to an improvement in L1-imbalance. This raises the question of how to match nodes which should be swapped. Probably in many cases swapping sets of vertices instead of single vertices is required to

Algorithm 3: Target Weight Reduction

```
1 c_t ← c_max
2 while partition imbalanced do
3     greedyRebalancer(G, Π, c_t)
4     if partition imbalanced then
5         rollback()
6         max_ib ← max_{V_i∈Π, d∈[m]} c(V_i)[d] / c_t
7         c_t ← (1/max_ib)^x · c_max
```

resolve deadlocks. Further research will be needed to investigate this approach. Instead, we propose resolving deadlocks by performing an adjusted rebalancing. Therefore, we propose two strategies, *target weight reduction* and *fallback*.

4.2.1 Target Weight Reduction

As the first deadlock handling strategy, we propose an algorithm 3 which performs several rebalancer runs while successively decreasing the target weights, i.e. the weights we use to calculate the $gain_{L1}$ of a move. This is motivated by the observation that in a deadlock scenario, in most cases the target weights are surpassed just slightly. So, if we decrease the target weights appropriately by a certain factor f, we end up with a solution surpassing our new target weights, but not surpassing our actual maximum block weight. It is important to note that we decrease the target weights used to compute $gain_{L1}$, but our stop condition remains the same, i.e. we stop when the partition satisfies the constraints given by the actual maximum block weight.
Therefore, as long as a greedy rebalancer run with target weights $c_t \leftarrow f \cdot c_{max}$ fails, we roll back the moves done by the rebalancer and try again with new target weights $c'_t \leftarrow f' \cdot c_{max}$. When this process was not successful after a certain number of tries, it is aborted.
However, it is not clear how to choose f'. It appears reasonable to choose f' based on the amount of imbalance left by the prior rebalancing run. For the last rebalancing round we compute the maximum fraction by which it surpassed one of its target weights, i.e. $max_{ib} := \max_{\substack{V_i \in \Pi \\ d<m}} \frac{c(V_i)[d]}{c_t}$. If we assumed that the next rebalancing run will surpass it's target part weights by the same factor the last one did, we could choose $f' = \frac{1}{max_{ib}}$ But since achieving target part weights becomes harder as the target weights become smaller, we expect our next rebalancing run to miss it's target weights by a larger factor than the last rebalancing run did. We therefore choose $f' = (\frac{1}{max_{ib}})^x$ for an $x \geq 1$.

4.2.2 Bin Packing Fallback

However, rolling back all moves and starting a new rebalancer run is computationally expensive. Since in most deadlock cases remaining imbalances are very small, we propose a fallback algorithm 4 which in many cases is able to resolve deadlocks by only moving a few nodes. So far, our rebalancing works by greedily moving single vertices. Since in a deadlock no single move is available improving the imbalance, we need a more global strategy. Our fallback works by first removing a set of nodes and then reinserting these nodes properly.
Potentially we do not need to fully balance the partition in order to resolve the deadlock. We suppose that nodes with small weight can be moved more easily by the rebalancer. Thus, it is sufficient if our fallback resolves imbalance produced by large nodes. We initially remove nodes v with $\max(c(v)) < \delta$ for a parameter $\delta > 0$. We call the set of these nodes L. The fallback then tries to find a balanced partition of $V \setminus L$. After fallback, the nodes from L are reinserted into their former blocks and we check if this results in imbalance. If so, we run the greedy rebalancer another time.
Our fallback starts by determining a set $S \subset V \setminus L$ of nodes so that $\Pi(V \setminus (S \cup L))$ is balanced. We then run a bin packing algorithm which tries to reinsert S into the partition without violating balance constraints. If the bin packing finds a balanced solution, we return. Otherwise we expand S until $||c(S)||_1$ has doubled. We then again try bin packing S. We keep doubling S and trying to binpack until bin packing was successful or we reach $S = V \setminus L$.
Let $S[1..i]$ be the set of the i nodes inserted into S the earliest. Once the bin packing was successful, we want to find the smallest i so that performing a bin packing on $S[1..i]$ is successful. We assume that if bin packing on $S[1..i]$ failed, bin packing on $S[1..j]$ will fail for $j < i$. Although this assumption does not necessarily hold true, it makes a binary search possible. The start points of the search are the full set S and the greatest subset of S for which bin packing failed. Note that if bin packing was successfully in its first attempt, we have nothing to do since we did the first bin packing right after $\Pi(V \setminus (V \cap L))$ became balanced.
In the following, we discuss the *extraction*-algorithm 22 which determines which node should next be moved into S and the *bin packing*-algorithm 13 which tries to pack the nodes from S into the partition.

Extraction Algorithm

Our extraction algorithm 5 first decides from which block the next node will be extracted, then it decides which node is extracted from that block.
While $\Pi(V \setminus (L \cup S))$ is imbalanced, we simply choose one of the overloaded blocks, since we anyway will have to extract from them. Otherwise, it becomes more complicated as Figure 4.2 shows.

Algorithm 4: Fallback

```
1  L ← {v ∈ V(G) | max(c(v)) ≤ δ}
2  S ← ∅
3  S' ← ∅
4  while Π(V \ (L ∪ S)) imbalanced do
5  |   v ← extract(V \ (L ∪ S))
6  |   S ← S ∪ {v}
7  Π' ← binpack(S)
8  if Π(V \ (L ∪ S)) ∪ Π' balanced then
9  |   Π ← Π(V \ (L ∪ S)) ∪ Π'
10 |   return greedyRebalancer(Π, L)
11 while Π(V \ (L ∪ S)) ∪ Π' imbalanced and V \ (L ∪ S) ≠ ∅ do
12 |   w_tmp ← ||c(S)||_1
13 |   S' ← S
14 |   while ||c(S)||_1 < 2 · w_tmp and V \ (L ∪ S) ≠ ∅ do
15 |   |   v ← extract(V \ (L ∪ S))
16 |   |   S ← S ∪ {v}
17 |   Π' ← binpack(S)
18 if Π(V \ (L ∪ S)) ∪ Π' imbalanced then
19 |   return false
20 Π' ← binarySearch(S', S)
21 Π ← Π(V \ (L ∪ S)) ∪ Π'
22 return greedyRebalancer(Π, L)
```

For readability, in this example we have chosen to scale block weights by factor 4, i.e. $c(V) = (4, 4)$ instead of $c(V) = (1, 1)$ as actually required. For simplicity, we assume that $L = \emptyset$ and $c(v)[0] \neq 0$ for every node v. We see that $\Pi(V \setminus (S \cup L))$ is balanced. If we would extract from another block than block 1, clearly the bin packing will not find a valid solution,. Given that $c(V)[1] = 4$ we see that $c(S)[1] = 4 - c(V \setminus S) = 0.6$. But the blocks 2,3 and 4 have only 0.4 space left in dimension 1 and we can not insert any node in block 1 since $c(V_1 \setminus S) = c_{max}$. We see that V_1 "wastes" 0.6 of space in dimension 1. Thus, we should extract from block 1. Assume that we now extract a node v with $c(v) = (0.05, 0.05)$ from $V_1 \setminus S$. Now $V_2 \setminus S$ is the only block which still wastes space. However, it is still more reasonable to extract from block 1. V_2 only wastes 0.1 of space in dimension 0, while probably it still is not possible to fill up $V_1 \setminus S$ in dimension 1 without causing imbalance in dimension 0.

For simplicity of notation, from now on we assume $L = \emptyset$. We use *penalty*-heuristics which try to estimate for each block how difficult it is to fill up the block weight without leaving space. Based on penalty functions, we then choose the block whose weight

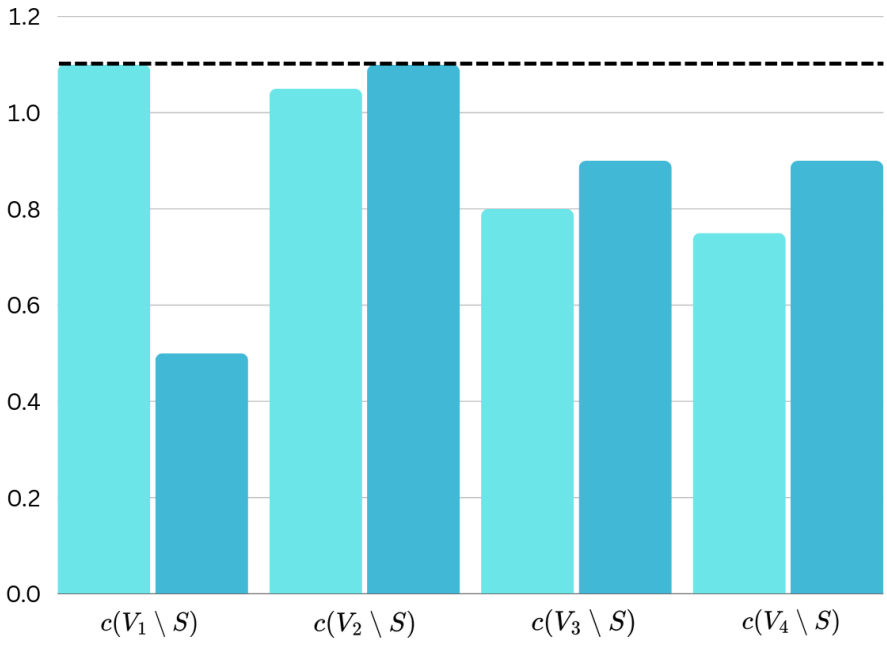

Figure 4.2: 4-way partition with 2 constraints and $\epsilon = 0.1$

has the highest penalty. We suppose that there are mainly two factors influencing how much space will be wasted for a block V_i'. First, the difference between the weights of V_i'. Secondly, the degree to which the block is already filled in its heavier-loaded dimensions. We test the following penalty metrics, which differ in how they prioritize these two aspects.

(i) $pen(w) = ib_{vert}(w) \cdot \overline{w}$
(ii) $pen(w) = \frac{ib_{vert}(w)}{c_{max} - \overline{w}}$
(iii) $pen(w) = \frac{c_{max} - \min(w)}{c_{max} - \max(w)}$
(iv) $pen(w) = \frac{c_{max} - \overline{w}}{c_{max} - \max(w)}$
(v) $pen(w) = \frac{\max(w) - \min(w)}{c_{max} - \max(w)}$
(vi) $pen(w) = \max(w)$

After we determined the block $V_i \setminus S$ to extract from, we need to determine the node v which will be extracted. We want to select a node that makes it easier to fill up $V_i \setminus S$ without wasting space. Thus, we want to reduce the vertical imbalance of the blocks weight. At the same time, we want that moving the selected node to another block does not worsen the cut too much. Let d be the heaviest dimension of $c(V_i \setminus S)$, we estimate the improvement in vertical imbalance of removing v from $V_i \setminus S$ as $c(v)[d] - avg(c(v))$. In other words, we focus on reducing the heaviest weight of $c(V_i)$. Secondly, we have to estimate the worsening in cut of moving v to another block. Our bin-packing algorithm will not consider gains when reinserting nodes in the partition. Thus, we cannot make any assumptions about in which block v or any of the nodes already in S will be reinserted. In expectation extracting and reinserting v changes the cut by $loss_{cut}^{exp}(v) := \omega(v, V_i \setminus S) - \frac{1}{k} \cdot$

$\sum_{V_j \in \Pi} \omega(v, V_j \setminus S)$. In order to find a node with high balance gain and small cut increase, we define for a node v and a dimension d we now define the priority of v with respect to dimension d as follows.

$$prio_d(v) = \begin{cases} (c(v)[d] - avg(c(v))) \cdot \frac{1}{\max(loss_{cut}^{exp}(v), \frac{1}{k+1})} & c(v)[d] - avg(c(v)) \geq 0 \\ (c(v)[d] - avg(c(v))) \cdot \max(loss_{cut}^{exp}(v), \frac{1}{k+1}) & c(v)[d] - avg(c(v)) < 0 \end{cases}$$

We lower-bound $loss_{cut}^{exp}(v)$ to $\frac{1}{k+1}$, since negative values of $loss_{cut}^{exp}(v)$ could not be handled by the formula and for nodes with $loss_{cut}^{exp}(v) = 0$ we still want the priority to differentiate between nodes with high and low $c(v)[d] - avg(c(v))$.
Our algorithm now holds m priority queues per block. For a block V_i, each node $w \in V_i$ is inserted into all m queues of V_i. w is inserted into the d-th queue with priority $prio_d(w)$. When we want to extract a node from V_i, we then retrieve the maximum v from the d-th queue where d is the heaviest dimension of V_i. When v is already part of S (it was already selected from one of the other priority queues), we retrieve the next maximum of the d-th queue. After we successfully retrieved a node v we add it to S and update the priorities of $N(v)$ since $loss_{cut}^{exp}(w)$ changed for $w \in N(v)$. For every $w \in N(v)$ we thus update its priority in all m queues.
Consider that V_i is imbalanced in its heaviest dimension d. Recall that we use $c(v)[d] - avg(c(v))$ to estimate the improvement in vertical imbalance of extracting a $v \in V_i$. Potentially this leads to extraction of a node v with $c(v)[d] - avg(c(v)) \gg c(V_i)[d] - avg(c(V_i))$, which inverts the weight distribution of $c(V_i)$. This is a problem since such a large node will very likely not be insertable to another block than V_i, thus the bin-packing will need to reinsert it into V_i. So actually V_i is still imbalanced in dimension d and further extraction steps should focus on reducing the d-th weight of V_i. But the extraction algorithm will do the opposite since $c(V_i \setminus \{v\})$ is underbalanced in dimension d.
To address this problem, we tested whether excluding vertices v with $||v||_1 > \delta$ for a threshold δ leads to better performance of the fallback. However, this did not lead to an improvement.

Bin Packing Algorithm

The bin packing 6 receives the partition Π and the set of nodes S (or a subset of S if we are already in the binary search on S). The nodes from S initially are not in Π. The bin packing then has to insert the nodes from $S$is balanced. This problem is related to the multi-capacity bin packing problem described in 3.2.4. The multi-capacity bin packing problem is about fitting *items* in *bins*, which corresponds to our problem of fitting nodes in blocks. To avoid confusion, we will refer to items as nodes and to bins as blocks when we are talking about the multi-capacity bin packing problem.
Our problem differs from multi-capacity bin packing in two ways. First, in our problem

Algorithm 5: Extraction

```
Data: S, queues[1..k][1..m]
1  i ← 0
2  if ∃j ∈ [k] : V_j \ S imbalanced then
3  |  i ← j
4  else
5  |  i ← argmax_{j∈[k]}{pen(c(V_j \ S))}
6  d_max ← argmax_{d∈[m]}{c(V_i \ S)[d]}
7  v ← 0
8  do
9  |  v ← queues[i][d_max].deleteMax()
10 |  for w ∈ N(v) \ V, d ∈ [m] do
11 |  |  queues[Π(w)][d].update(w)
12 while v ∈ S
13 return v
```

the number of blocks is given as k while multi-capacity bin packing tries to minimize the used number of blocks. Secondly, we only insert the nodes from S while leaving nodes from $V \setminus S$ in their blocks. Multi-capacity bin packing starts with empty bins and assigns all nodes. Although our extraction algorithm tries to select nodes in a way such that the vertical imbalance of the remaining block weights $c(V_i \setminus S)$ is reduced, they can still be very imbalanced. This is an important difference to multi-capacity bin packing. Existing multi-capacity bin packing algorithms like [10] can avoid this difficulty by trying to keep block weights vertically balanced throughout the algorithm.
Multi-capacity bin packing algorithms like Choose Pack or Permutation Pack [10] consider the suitability of fitting an item X into a bin B_j only based on their orders of weights. However, we suppose that for highly vertically imbalanced block weights this is not accurate, as the order of weights does not tell about how much the weights differ from each other. For instance, in a case like Figure 4.3 Choose Packs or Permutation Packs would try to fill up $V_1 \setminus S$ with nodes $w \in S$ which have $c(w)[1] \geq c(w)[0]$. Potentially they choose nodes with $c(w)[1] \approx c(w)[0]$ which would would fill up $c(V \setminus S)$ sub-optimally.

For our bin packing algorithm, we rather suggest to insert nodes based on a penalty function which estimates for a block weight $c(V_i')$ how difficult it is to fill up V_i without leaving space. We discussed such penalty functions in 22. Assume that (potentially after some nodes have already been inserted) we have blocks $\{V_1', .., V_k'\}$ and a set of nodes $X \subset S \setminus \bigcup_i V_i'$ and we want to select a node from X which will be packed next. If there is a $v \in X$ which does not fit in any block, we return that no solution could be found. If there is a $v \in X$ which fits in only one block, we select v and pack it into that block. Otherwise, we do the following. For $v \in X$, let $gain_{pen}(v, V_j') := pen(V_j') - pen(V_j' \cup \{v\})$

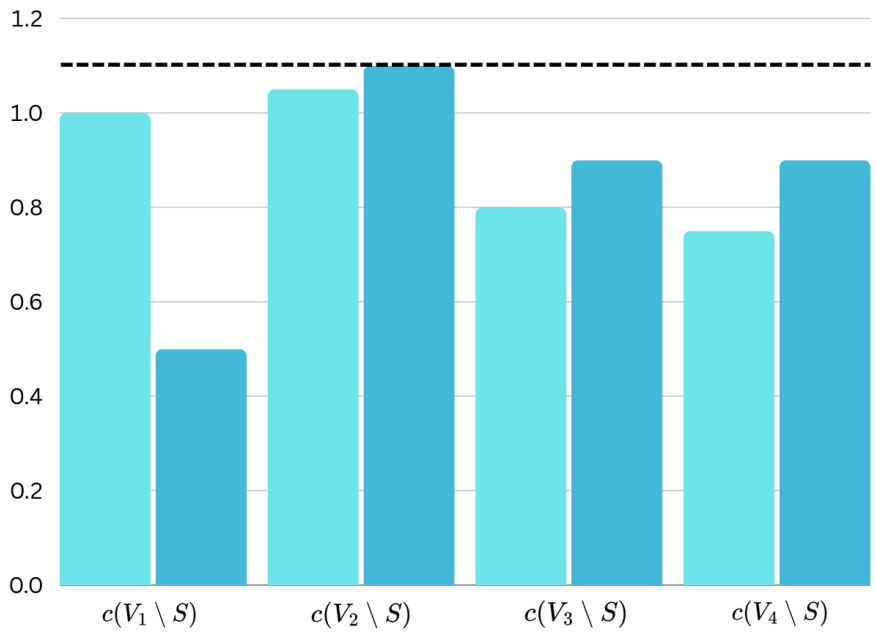

Figure 4.3: 4-way partition of $V \setminus S$ with 2 constraints and $\epsilon = 0.1$

be the reduction in penalty when adding v to the j-th block. Furthermore, we compute the two blocks with best gain, i.e. $best(v) := argmax_{j \in [k]} gain_{pen}(v, V'_j)$ and $second(v) := argmax_{j \in [k] \setminus best(v)} gain_{pen}(v, V'_j)$. We then define the priority of v as the difference in $gain_{pen}$ between the two blocks with best gain, i.e. $prio(v) := gain_{pen}(v, V'_{best(v)}) - gain_{pen}(v, V'_{second(v)})$. We then select the $v \in X$ with highest priority and pack it to $V'_{best(v)}$. Thus, the idea is to first select the node for which it matters the most which block it is assigned to. It remains to determine a set X which covers as many of these important nodes as possible. We suppose that large nodes and highly vertically imbalanced nodes are important nodes. For heavy nodes, it is more difficult to find a block to fit them in. Additionally, even when a node fits into all blocks, it brings the blocks weight closer to the maximum weight than a lightweight node would do. The closer a blocks weight is to its maximum weight, the more does a higher vertical imbalance increase difficulty in filling up that block. Thus, for a heavy node it is more important to pack it into the most suitable block.
We believe that also highly vertically imbalanced nodes should be prioritized. This is because in contrast to vertically imbalanced nodes for a perfectly balanced node the permutation of weights of a potential destination block does not matter.
Based on these insights, we decide to construct X by building $m + 2$ lists which each contain the nodes of S in a specific sorting order. X then is defined as the set which for each list contains the first element from that list which was not packed yet. In the following, we describe the lists we use.
One list is ordered by $||c(v)||_1$ in a non-ascending way to prioritize heavy nodes. Another list is ordered by vertical imbalance in a non-ascending way. Finally, for each constraint

Algorithm 6: Bin Packing

```
Data: S, lists[1..m+2], Π = {V'_1, .., V'_k}
lists ← updateLists(lists, S) // if S was expanded since last bin packing
X ← {nextUnInserted(l) | l ∈ lists}
while X ≠ ∅ do
    if ∃v ∈ X : ∀V'_i ∈ Π  c(V'_i) ∪ {v}) ≰ c_max then
        return no solution found
    if ∃v ∈ X, j ∈ [k] : ∀V'_i ∈ Π \ V'_j  c(V'_i ∪ {v}) ≰ c_max then
        V'_j ← V'_j ∪ {v}
    else
        v ← argmax_{v∈X}{prio(v)}
        V'_{best(v)} ← V'_{best(v)} ∪ {v}
return Π
```

d we add a list sorted by $c(v)[d]$ in a non-ascending order. This is done for the case that many blocks are in one dimension nearly full and few blocks are in that dimension very empty. We believe that usually this case occurs since deadlocks commonly are caused by a block which is imbalanced in one dimension being very low-weighted in another dimension while the balanced blocks are fully loaded in that dimension.
When the bin packing algorithm is run and S was increased since the last bin packing run, for each list the newly added nodes is sorted and merged with the list. During the binary search, no sorting is performed. Instead, nodes which are not in the currently observed subset of S are simply ignored.

5 Experimental Evaluation

In this chapter, we evaluate the proposed unconstrained refinement approach for partitioning multi-constraint graphs. In Section 5.1, we describe our experimental setups including hardware and software properties and the generation of our benchmark set. In the following sections, we evaluate different configurations of rebalancing 5.2, refinement 5.3 and deadlock handling 5.4. These evaluations are performed as top-level refinement on a prepartitioned graph, see 5.1 for further detail. In Section 5.5 we evaluate the best configuration on the coarsening hierarchies provided by Mt-KaHyPar [6] and METIS [8] and compare our algorithm to METIS.

5.1 Setup and Methodology

All algorithms were implemented in C++ within the Mt-KaHyPar framework[1]. Our algorithm uses some Mt-KaHyPar functionality. For reading in graphs we use Mt-KaHyPar IO, for calculation of gains of moves we use the gain cache of Mt-KaHyPar. In the following we describe the setups used for our experiments.
For the evaluation of the parameters of our algorithms (i.e. rebalancing 5.2, refinement 5.3 and deadlock evaluation 5.4) we perform top-level refinement, i.e. refinement on prepartitioned fully uncoarsened graph. Partitions are received by the METIS (3.2.2) framework. More precisely, we edited the source code[2] of multi-constraint METIS so that top-level refinement is skipped. Our experiment setup then uses Mt-KaHyPar-IO to read-in the partition computed by METIS and runs our local search algorithm. That means our algorithm gets the same input as the last local search of unmodified METIS. By that our local search algorithm can be compared to METIS very reliably. However, since deadlock handling plays a much smaller role in top-level refinement than in coarser graph levels, performance differences between different deadlock handling configurations tend to be smaller than on coarser graph levels.
In Section 5.5, we evaluate the configuration which performed best in top-level refinement on full-level refinement. For full-level refinement, our local search algorithm is supplied with a coarsening hierarchy and an initial partition and then performs a local search on each uncoarsening level. The required data (i.e. coarsening hierarchy and initial partition) is provided by another graph partitioner. We test both multi-constraint METIS and

[1] `https://github.com/kahypar/mt-kahypar/tree/master/mt-kahypar`
[2] `https://github.com/KarypisLab/METIS`

Mt-KaHyPar. For multi-constraint METIS, we modified the source code to obtain both coarsening hierarchy and initial partition as an output of the executable. For Mt-KaHyPar, we extended the source code to handle multiple constraints by simply extending balance conditions to multi-constraint. We then called Mt-KaHyPar coarsening via the framework. All experiments are performed on our benchmark set (5.1.3). Each graph is partitioned into $k = 2, 4, 8, 16, 32$ blocks for 8 different seeds each.
For a visual comparison of the result quality of different partitioners or of different configurations of one partitioner, we use plots based on *performance profiles* [3]. We always report the average performance across all 8 seeds. More precisely, for each instance and parameters k, we calculate the minimum of the objective function for all seeds for which the resulting partition is balanced. If the average imbalance over the seeds exceeds the imbalance parameter, the instance is considered imbalanced (even if there is a seed with a balanced partition). The plot is constructed by showing a curve for each included algorithm in a specific color corresponding to the algorithm. The x-axis shows the quality τ relative to the quality of the best partition found by all algorithms. The y-axis shows the fraction of instances of the algorithm where the quality is within a factor of τ relative to the best partition. For example, if an algorithm has the value 0.6 on the y-axis for τ = 1.5, then 60% of the partitions computed by this algorithm have a quality which is at most 1.5 times worse than the best solution found. For τ = 1 the y-value indicates the fraction of instances where the algorithm computed the best solution. Note that the x-axis is separated into three areas with different scale: The first area ranges from 1 to 1.1, indicating instances which are very close (within 10%) to the quality of the best result. In the second area, instances with medium quality, i.e. at most 2 times the value of the objective function for the best result, are displayed. The third area uses a logarithmic scale for the remaining instances, converging to infinity. The fraction of imbalanced results is displayed at the right side where the x-axis is marked with a ✘-tick. Remaining instances are those for which the partitioner was unable produce a balanced solution. An algorithm is considered to outperform another algorithm in terms of result quality if its corresponding curve is above that of the other algorithm.

5.1.1 Environment

We use two machines for testing. The first machine, which is used for the top-level refinement tests performed in sections 5.2, 5.3 and 5.4, has 64 Intel(R) Xeon(R) Gold 6314U CPU @ 2.30GHz processors and runs an Ubuntu 22.04.4 operating system. The other machine, which is used for the full-level refinement experiments performed in 5.5, has 160 Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz processors and runs an Ubuntu 22.04.4 operating system.

5.1.2 Tuning Parameters

In this section, we give an overview of the parameters which will be tested in this section. Further descriptions of the meanings of these parameters can be found in Chapter 4. For each parameter, we give its default value. The experiments described later in this chapter each only vary one parameter, while the other parameters are set to the default values.

Refinement

The refinements 3.2.2 main parameter is the maximal allowed imbalance γ. Per default, $\gamma = 0.2$. Another important detail of the refinement is whether vertex locking 4 is performed. Per default, vertex locking is performed. A further variable of the algorithm is the priority use for the PQ. By default, the priority of a node v equals the gain of the highest gain move of v. The last refinement parameter is the tiebreak which decides between moves with equal gain. Per default, we use max-norm balance gain as tiebreak, same as METIS does.

Rebalancing

For rebalancing 4.1, the update frequency determines how often the queue is updated. By default, it is set to 0.

Deadlock Handling

The main decision which is met for deadlock handling 4.2 is whether the fallback or the target weight reduction algorithm is used. By default, the fallback algorithm is used.
The parameters of the target weight reduction algorithm are the maximal number of unsuccessfull rebalancer runs, which is by default set to 5, and the exponent x. x is by default set to 1.01, which corresponds to the assumption that when target weights become lower the rebalancer tends to surpass them by a slightly larger factor.
The most important parameter of the fallback is the threshold δ. We assumed that nodes v with $\max(c(v)) \leq \delta$ can be ignored by the fallback since the greedy rebalancer will be able to move them. By default, δ is set to 0, which means that no nodes are ignored. During fallback twice a penalty metric is needed. First, when the set S of extracted nodes is expanded, a penalty function is used to determine the block from which a node is extracted. Secondly, the bin packing uses a penalty function to decide when which move will be packed. By default, for both uses the same function $pen(w) = \frac{ib_{vert}(w)}{c_{max} - \overline{w}}$ is used. However, both uses are evaluated independently.

Set	m	c_1	c_2	c_3	c_4
regular	2	$deg(v)$	1	-	-
regular	2	$u([1,100])$	$u([1,100])$	-	-
regular	3	$deg(v)$	1	$u([1,100])$	-
irregular	2	$deg(v)$	1	-	-
irregular	2	$u([1,100])$	$u([1,100])$	-	-
irregular	3	$deg(v)$	1	$u([1,100])$	-
VLSI	2	$c(v)$	1	-	-
VLSI	3	$c(v)$	$deg(v)$	1	-
VLSI	4	$c(v)$	$deg(v)$	1	$u([1,100])$

Table 5.1: Benchmark Set

5.1.3 Instances

We generate a benchmark set of multi-constraint graphs based on three sets of graphs. The first set consists of unweighted regular graphs. The second set consists of unweighted irregular social graphs with power-law bounded degree distribution. The third set consists of weighed VLSI graphs.
From these three sets we generate multi-constraint graphs in different ways. We include 2-constraint, 3-constraint and 4-constraint graphs. The two constraints c_1, c_2 where c_1 assigns $deg(v)$ to each node v and c_2 assigns 1 to each node are of high practical relevance as the tend to approximate the computational load for algorithms like parallel Page Rank [1] very well. Thus, for every unweighted graph G we have we generate a 2-constraint graph G' with constraints $c_1(v) \coloneqq deg(v)$ and $c_2(v) \coloneqq 1$. For every weighed graph we generate a 3-constraint graph G' with weights $c_1(v) \coloneqq deg(v)$, $c_2(v) \coloneqq 1$ and $c_3(v) \coloneqq c(v)$ where $c(v)$ is the weight of v in G. Additionally, we generate 2-constraint graphs from the weighted graphs by using constraints $c_1(v) \coloneqq c(v)$ and $c_2(v) \coloneqq 1$.
Furthermore, we generate graphs using random weights. All random weights are drawn from a uniform distribution over [1,100]. We generate 2-constraint graphs from our unweighted graphs by adding two randomly generated constraints $c_1(v) \coloneqq u([1,100])$, $c_2(v) \coloneqq u([1,100])$ where u stands for the uniform distribution. Finally, by combining $deg(v)$, 1 and $u([1,100])$ we receive 3-constraint graphs from all unweighted graphs and 4-constraint graphs from all weighted graphs.
In total, our benchmark set consists of 1320 graphs. Figure 5.1 gives an overview of our benchmark set. When comparing performances of algorithms, we will also regard the total number of instances solved by an algorithm. Keep in mind that we do not know for how many of these instances there exists a solution, so we can only compare algorithms to each other.

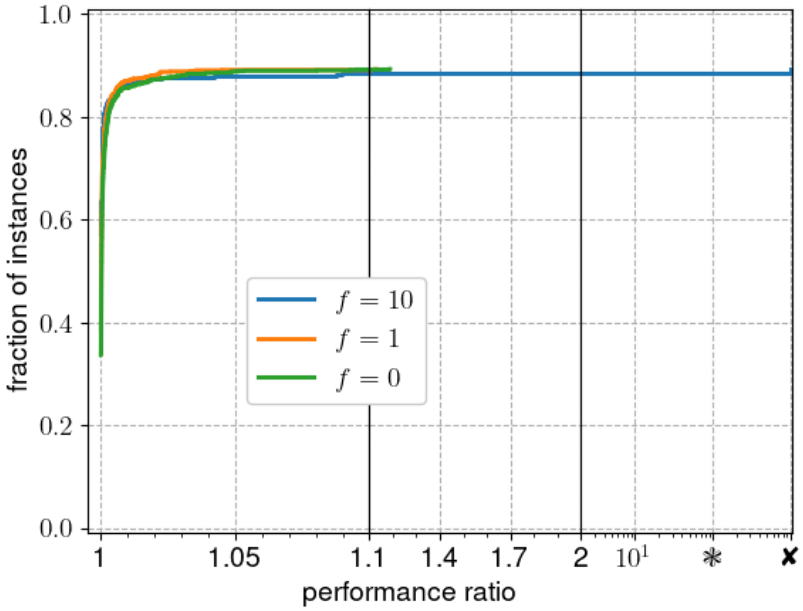

Figure 5.1: Different values for the rebalancers parameter f which determnies the frequency of periodic updates

5.2 Evaluation of the Greedy Rebalancer

One of the main conceptual differences between our multi-constraint greedy rebalancer and single-constraint greedy rebalancers like that one of Mt-KaHyPar is that our algorithm defines priority based on L1-gain instead of nodeweight. As discussed in Section 4.1, changes in L1-gain leads to the problem of outdated priorities. Our rebalancing tries to handle this problem by using lazy updating and periodic complete updates. First, we evaluate whether performing periodic updates leads to a significant improvement in performance. As described in 4.1, the updating frequency is computed based on a parameter f and an approximation of the number of moves the rebalancer will perform. In Figure 5.1, we vary the parameter f between 0, 1 and 10. We did not observe any significant changes in partitioning quality. Surprisingly, the configuration with the highest frequency parameter solved the least instances, although the difference was not significant.

Every time when a node is retrieved from the PQ, its priority is recomputed. If the recomputed priority is lower than the priority saved in the PQ, the node is not moved and instead reinserted in the PQ with its correct priority. This is what we referred to as lazy updating. Figure 5.2 shows the average overhead of lazy updating in the default configuration of our rebalancer. More precisely, for a given ratio r it shows how many percent of the rebalancer runs had $\frac{|lu|}{|V(G)|} \leq r$ where $|lu|$ denoted the number of lazy updates performed. We see that

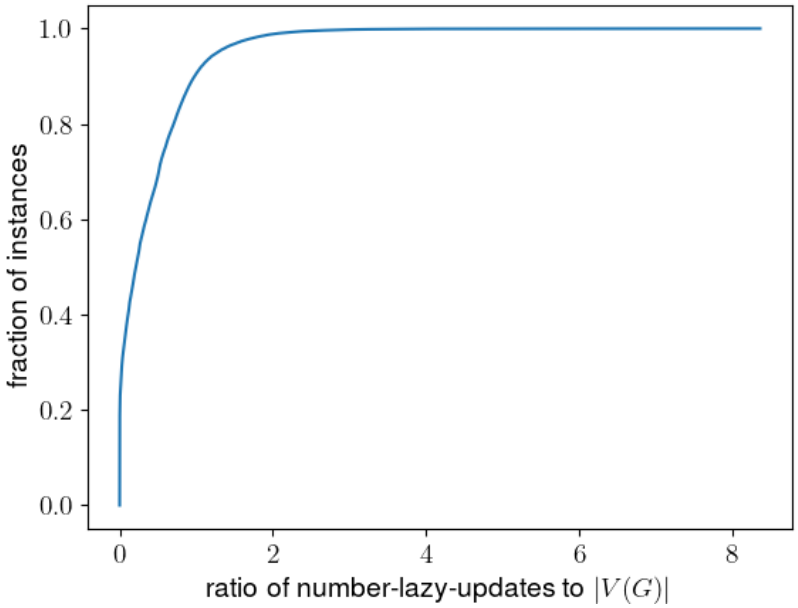

Figure 5.2: ratio between number of lazy updates and size of the graph

for most rebalancer runs $\frac{|lu|}{|V(G)|} \leq 1$. Since anyway the rebalancer initially has to touch and possibly insert all nodes, by bounding the number of lazy updates to the number of nodes we show that lazy updating does not become a main bottleneck.

5.3 Evaluation of the Refinement Algorithm

In this section we evaluate the refinement algorithm which is based on the METIS refinement algorithm 3.2.2. The most important parameter is the allowed imbalance δ. As Figure 5.3 shows, the best results were achieved by the configuration with 20% imbalance. Contrary to our result, unconstrained local search algorithms for single-constraint graphs like Mt-KaHyPar find their best results with fully unconstrained local search, i.e. without an upper limit for allowed imbalance [11]. This difference is probably because our refinement does not use penalty metrics as Mt-KaHyPar does 3.1.3.

Moreover, we evaluate different node traversal strategies for refinement. In our default configuration, nodes are retrieved from a PQ where the priority of a node is the gain of its max-gain move. We also test the use of the priority based on internal edgeweight, external edgeweight, and number of neighbored blocks, which METIS uses for refinement (3.2.2). Additionally, we test node-traversal in random order, i.e. no PQ is used. As shown in Figure 5.4, using the maximal gain as priority showed the best performance. Surprisingly, METIS priority performed slightly worse than random order. We try to give a short ex-

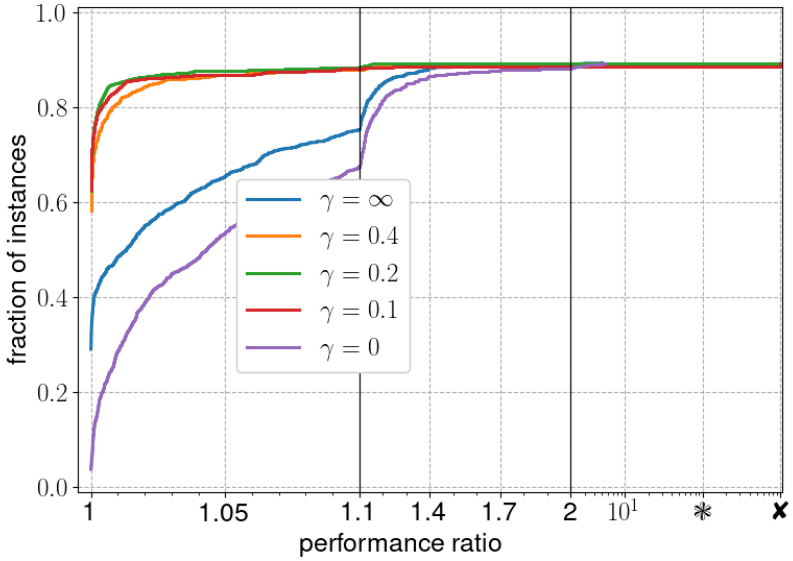

Figure 5.3: allowed imbalance for refinement

planation for this. Since the refinement algorithm is only allowed to cause imbalance to a certain degree, it is interested im first moving nodes with a high ratio between gain and imbalance increase. Many of our graphs are generated with node degree as one of the constraints (5.1.3). We believe that METIS metric tends to slightly *overestimate* the max-gain move for high degree (and thus heavy weighted) nodes and to *underestimate* the max-gain move for low degree (and thus lightweighted) nodes.

The tie-break of our refinement determines the destination block of a node when its two best moves have the same gain. We try using max-norm balance gain as tie-break, as METIS refinement does, and L1-balance gain. The results in Figure 5.5 showed no significant difference, although max-norm balance gain performed slightly better.

Finally, we evaluate whether locking vertices leads to a better performance. As Figure 5.6 shows, locking vertices lead to a significant improvement in the performance ratio area between 1 and 1.05, while no improvement could be observed for higher ratios. We believe that the main benefit of vertex locking is to overcome situations where a refinement round fails, i.e. the rebalancer worsens the quality more than the refinement did improve it and all moves are rolled back. In this case, without vertex locking it is very unlikely that any of the next refinement rounds will succeed. (Even though it is possible, since the nodes are inserted into the PQ in random order 3.2.2 and thus refinement rounds starting with the same partition Π can produce different results.)

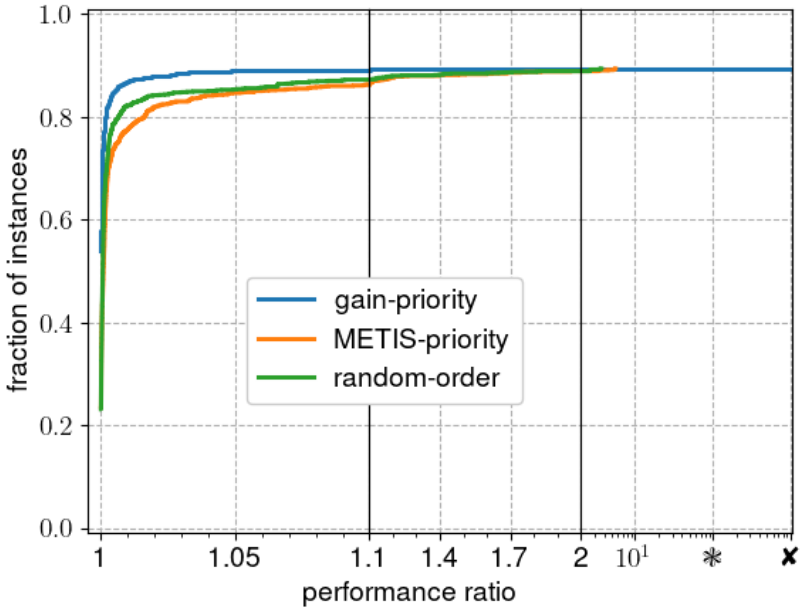

Figure 5.4: order of node traversals for refinement

5.4 Evaluation of Deadlock Handling Strategies

In this section we compare the two deadlock handling strategies we proposed in 4.2. First, we separately evaluate tuning parameters for the target weight reduction algorithm and the fallback algorithm. Subsequently we compare their best configurations to each other and to naive deadlock handling strategies to see whether deadlock handling is necessary.

5.4.1 Target Weight Reduction

As shown in Figure 5.7, we changed x which determines the rate at which the target weights are decreased. Performance differences between different values for x were very marginal. $x = 1.01$ tended to do best with a slightly superior performance and 1164 solved instances, while for the other values only 1161 to 1163 instances were solved.

5.4.2 Bin Packing Fallback

In this section we evaluate the fallback parameters. First, we evaluate the threshold δ based on which the set of ignored nodes $L = \{v \in V \mid \max(c(v)) \leq \delta\}$ is determined. We supposed that very small nodes could be ignored, i.e. it would be sufficient if the fallback manages to compute a partition so that $\Pi(V \setminus L)$ satisfies the balance constraints.

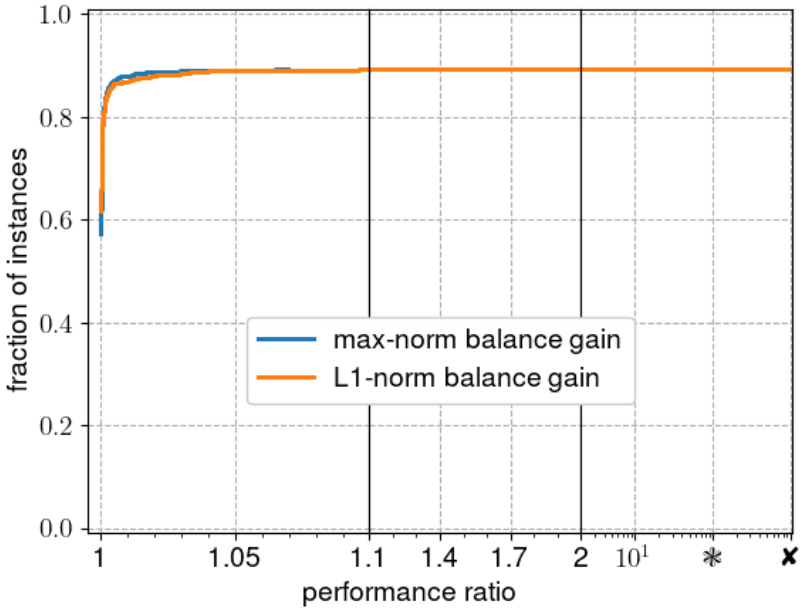

Figure 5.5: tie-break metrics

However, as shown in Figure 5.8 it turned out that even for very small $\delta = \frac{\epsilon}{100}$ this was not sufficient. While $\delta = 0.03$ and $\delta = 0.003$ performed identically (because for both values the fallback did nothing since $\Pi(V \setminus L)$ was always balanced from the beginning), $\delta = 0.0003$ performed slightly better with 1164 instances solved compared to 1161 instances solved with $\delta = 0.03$ and $\delta = 0.003$. Still, $\delta = 0.0$ clearly performed best in terms of partitioning quality and also solved significantly more instances (1180).
This experiment shows that in most cases not only heavy nodes are responsible for the occurrence of deadlocks. An imbalanced block in a deadlock can still have its imbalanced caused by many very light-weighted nodes. For instance, if V_i is overloaded in dimension d_1 and light-weighted in d_2, it can still contain many light-weighted nodes if all of these light-weighted nodes are heavier-weighted d_2 than in d_1 and the other blocks V_j, $j \neq i$ are all filled completely up in d_2. This experiment shows that in deadlocks this often seems to be the case.
Our fallback uses a $penalty$-function for extracting vertices from the partition as well as for bin packing. We evaluated the penalty functions we proposed in 22, e.g. $pen_1(w) = ib_{vert}(w) \cdot \overline{w}$, $pen_2(w) = \frac{ib_{vert}(w)}{c_{max} - \overline{w}}$, $pen_3(w) = \frac{c_{max} - \min(w)}{c_{max} - \max(w)}$, $pen_4(w) = \frac{c_{max} - \overline{w}}{c_{max} - \max(w)}$ and $pen_5(w) = \max(w)$. Both use cases of the penalty functions where tested separately. Our experiments show that the pen_2 solved the most instances both for extraction 5.9 and bin packing 5.10 with 1180 solved instances each.

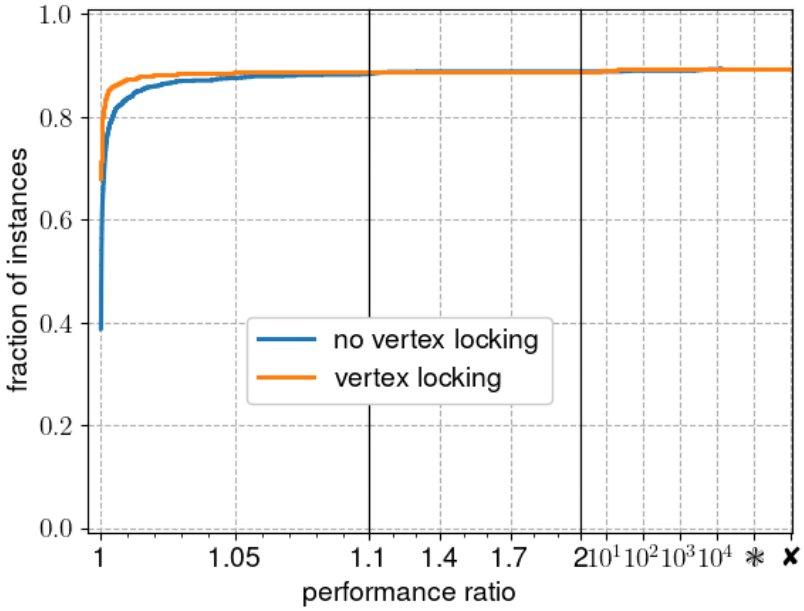

Figure 5.6: Vertex locking vs. no vertex locking

5.4.3 Naive and Lazy Deadlock Handling

Finally, we compare to naive handling and lazy deadlock handling to check whether a deadlock handling is necessary.
The naive configuration simply does not use fallback. That means when the rebalancing ends in a deadlock and the partitions max-norm imbalance is higher than at the start of the refinement round the moves are simply reverted. By using this configuration we try to find out whether handling deadlocks is necessary at all.
The lazy deadlock handling also forgoes deadlock handling. But differently to the naive configuration, it does not revert moves when an unconstrained round ended in a deadlock as long as the unconstrained round lead to an improvement in quality. Deadlocks are only handled in the last rebalancing step via the bin packing fallback. We call this configuration lazy fallback. The intuition behind this is that the imbalance of deadlocked partitions is usually very small. Since the greedy rebalancer takes more care of the gain of moves than the fallback does, in terms of partitioning quality it might be feasible to ignore the slight imbalance produced by the deadlock and hope that the greedy refiner will in further refinement rounds be able to balance it with less worsening in quality.
However, as 5.11 shows, using a non-naive deadlock handling strategy brings significant performance gains with regard to both the number of solved instances and partitioning quality. Bin packing fallback and target weight reduction slightly outperformed lazy fallback in terms of partitioning quality, while binpacking fallback solved significantly more

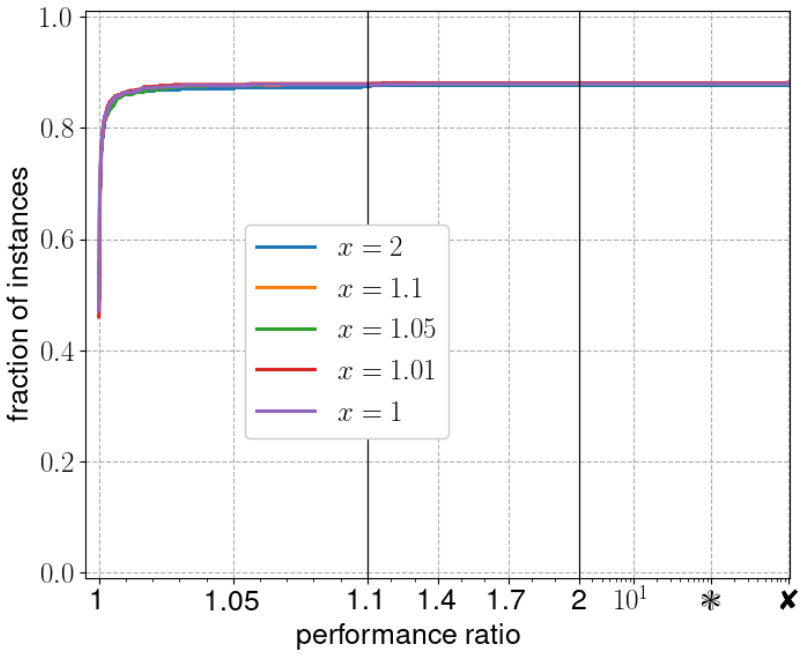

Figure 5.7: exponent x for target weight reduction

instances (1180) than lazy fallback (1167), target weight reduction (1164) and no fallback (1162).

5.5 Full-Level Refinement

In this section, we finally perform full-level refinement as described in Section 5.1. In the following experiments, for our algorithms we use the configuration which turned out best in the top-level refinement experiments. This configuration is the default configuration described in 5.1.2. In Figure 5.12, we compare the full-level refinement of our algorithm performed with METIS coarsening hierarchy and initial partition to our algorithms top-level refinement and to (unmodified) METIS. We see that both our full-level and our top-level algorithm outperform METIS. Our full-level refinement also performs significantly better than our top-level configuration, which hints that for multi-constraint graphs unconstrained refinement is also beneficial on coarser graphs where the load-balancing problem is more difficult.
In Figure 5.13 we finally compare the performance of our algorithm on the coarsening hierarchy and initial partition of METIS to the performance of our algorithm on the coarsening hierarchy and initial partition of Mt-KaHyPar. Our algorithm performs clearly better on

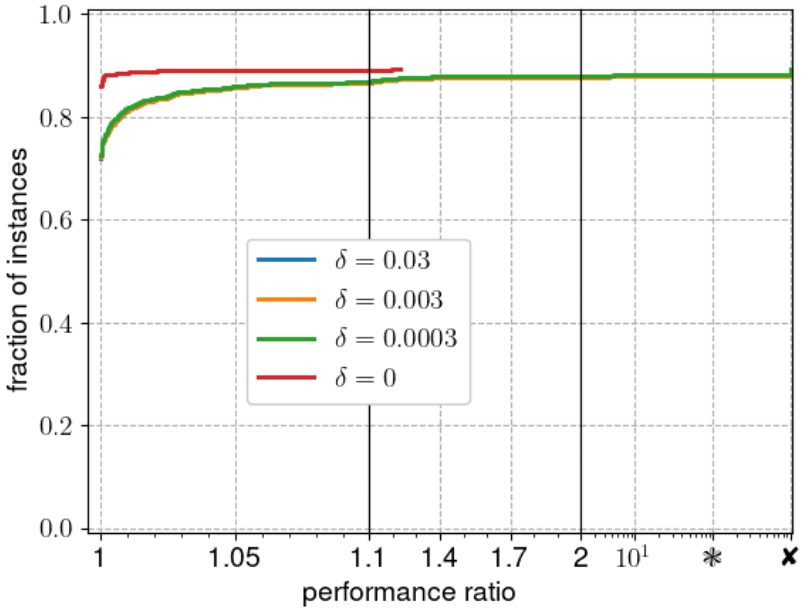

Figure 5.8: Threshold δ to determine set $L := \{v \in V(G) \mid \max(c(v)) < \delta\}$ of nodes which are ignored in the fallback algorithm

METIS coarsening hierarchy and initial partition. The reason is probably that METIS coarsening uses multi-constraint specific strategies for coarsening and initial partitioning 3.2.2.

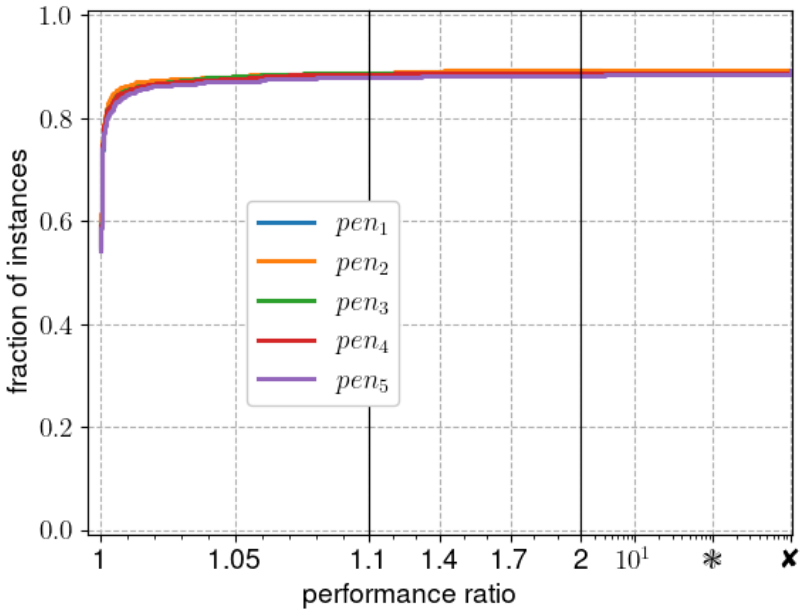

Figure 5.9: $penalty$-functions for extraction of nodes in the fallback algorithm

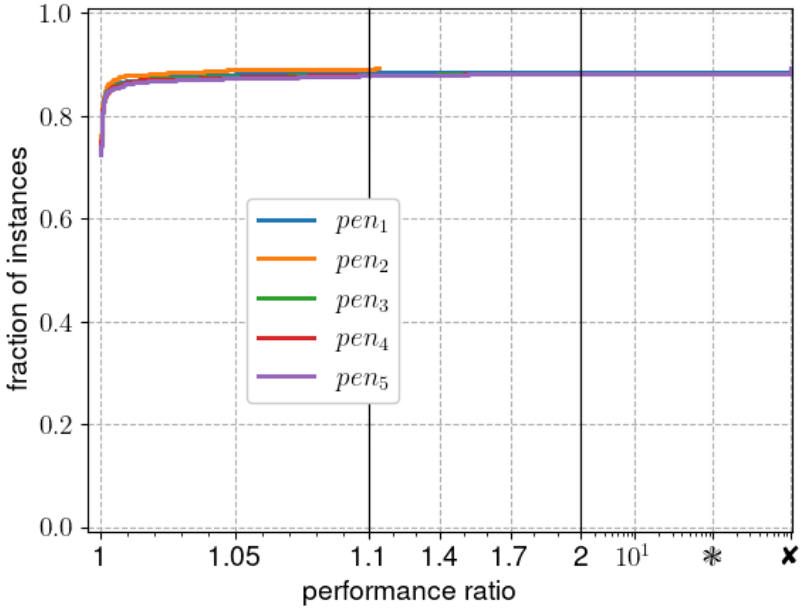

Figure 5.10: $penalty$-functions for bin packing in the fallback algorithm

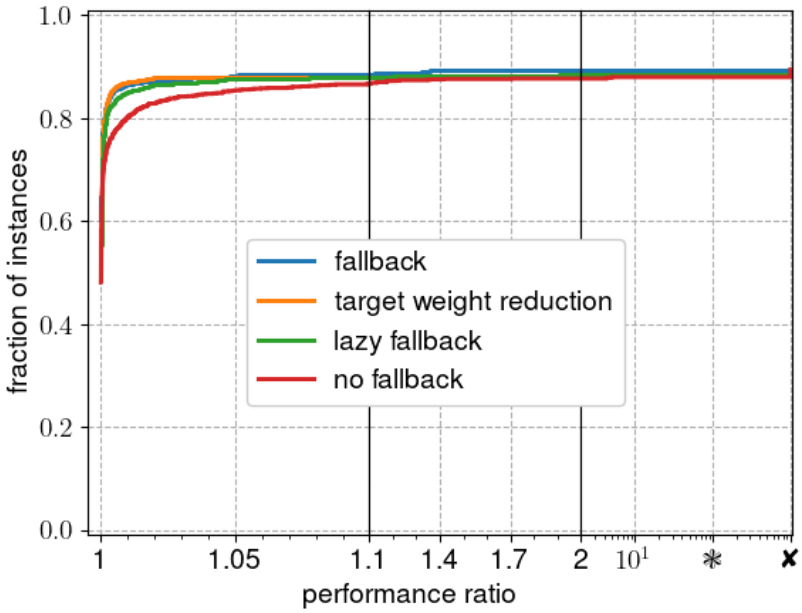

Figure 5.11: Different deadlock-handling strategies

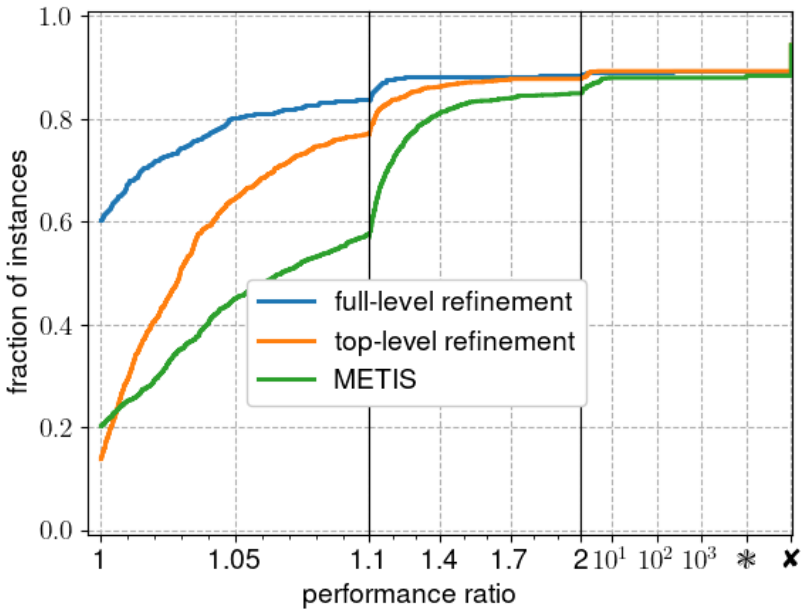

Figure 5.12: METIS refinement vs. top-level (our algorithm) vs. full-level (our algorithm)

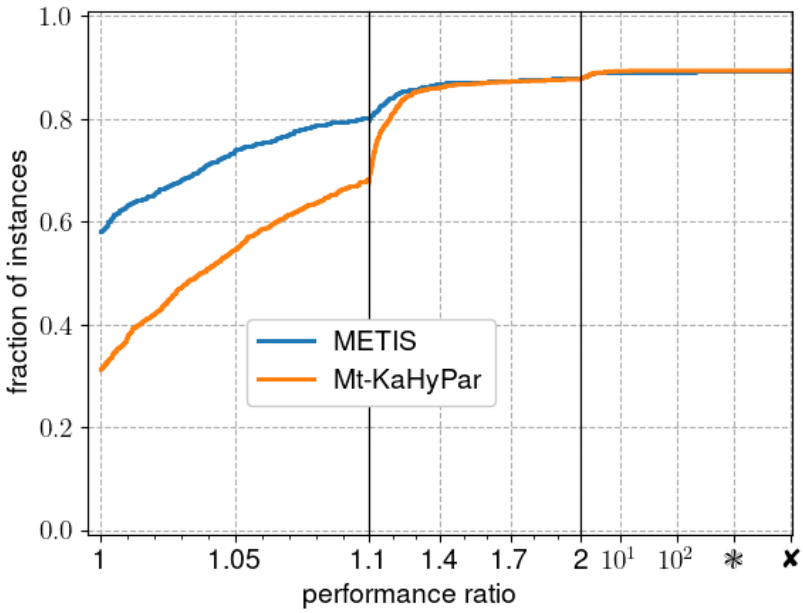

Figure 5.13: Performance of our algorithm with full-level refinement and a) coarsening hierarchy and initial partition of METIS or b) coarsening hierarchy and initial partition of Mt-KaHyPar

6 Discussion

In this thesis we proposed an approach for the multi-constraint graph partitioning problem based on unconstrained refinement. We were able to show that our algorithm produces solutions of higher quality than the state-of-the-art constrained refinement algorithm METIS [8] on a wide range of instances.
Although we did not optimize our code in terms of runtime, we were able to give a proof of concept of our algorithms efficiency. We did so by first showing that for our greedy rebalancer 4.1 lazy updating is sufficient to preserve high quality results 5.1. Furthermore, our experiments showed that the lazy updating technique does not represent a major bottleneck of our algorithm since the number of lazy updates performed usually can be bound by $|V(G)|$ 5.2.
While we did use a relatively simple refinement algorithm, we could show that by restricting the refinements allowed imbalance and by introducing vertex locking we were able to significantly gain performance 5.3.
Finally by using a bin packing approach 4.2.2 we were able to balance some difficult instances where greedy algorithms like our greedy rebalancing or the rebalancing of METIS do not find a solution 5.4.

6.1 Future Work

In order to further improve the quality of unconstrained refinement algorithms for multi-constraint graphs, future research should focus on refinement algorithms. We rather focused on rebalancing and used a pretty simple refinement algorithm. But since we still were able to significantly improve solution quality by transferring the vertex locking technique to multi-constraint graph partitioning, we believe that it should be tried to adjust other techniques from single-constraint unconstrained refinement like penalty estimations [11]. Regarding rebalancing, node swap techniques could make greedy algorithms powerful enough to overcome deadlocks 4.2. This could avoid the use of fallback algorithms and thus lead to further performance gains.
Also coarsening and initial partitioning seem to play a big role for the quality of multi-constraint graph partitioning, since our algorithm performed much better on METIS' coarsening hierarchy and initial partition than on that one of Mt-KaHyPar. This is probably because METIS uses specific multi-constraint strategies for coarsening and initial partitioning. Thus, future research should also regard coarsening and initial partitioning for

multi-constraint graph partitioning.
In order to improve the runtime of unconstrained refinement algorithms for multi-constraint graphs, future research should concentrate on parallelizing multi-constraint unconstrained refinement.

A Implementation Details

A.1 Software

A.2 Hardware

Bibliography

[1] Dmitrii Avdiukhin, Sergey Pupyrev, and Grigory Yaroslavtsev. *Multi-Dimensional Balanced Graph Partitioning via Projected Gradient Descent*. 2019.

[2] Cevdet Aykanat, B 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.

[3] Elizabeth D. Dolan and Jorge J. Moré. *Benchmarking Optimization Software with Performance Profiles*. 2004.

[4] Charles M Fiduccia and Robert M Mattheyses. "A linear-time heuristic for improving network partitions". In: *Papers on Twenty-five years of electronic design automation*. 1988, pp. 241–247.

[5] Michael S. Gilbert et al. *Jet: Multilevel Graph Partitioning on Graphics Processing Units*. 2024.

[6] Lars Gottesbüren et al. "Scalable Shared-Memory Hypergraph Partitioning". In: *2021 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pp. 16–30. DOI: 10.1137/1.9781611976472.2.

[7] Bruce Hendrickson and Robert Leland. "A Multi-Level Algorithm For Partitioning Graphs". In: Feb. 1995, pp. 28–28. ISBN: 0-89791-816-9. DOI: 10.1109/SUPERC.1995.242799.

[8] George Karypis and Vipin Kumar. "Multilevel algorithms for multi-constraint graph partitioning". In: *SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. IEEE. 1998, pp. 28–28.

[9] George Karypis and Vipin Kumar. "Multilevelk-way partitioning scheme for irregular graphs". In: *Journal of Parallel and Distributed computing* 48.1 (1998), pp. 96–129.

[10] W. Leinberger, G. Karypis, and V. Kumar. "Multi-capacity bin packing algorithms with applications to job scheduling under multiple constraints". In: *Proceedings of the 1999 International Conference on Parallel Processing*. 1999, pp. 404–412. DOI: 10.1109/ICPP.1999.797428.

[11] Nikolai Maas, Lars Gottesbüren, and Daniel Seemaier. "Parallel Unconstrained Local Search for Partitioning Irregular Graphs". In: *2024 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pp. 32–45. DOI: 10.1137/1.9781611977929.3.