Bachelor Thesis

Evaluation of Local Search Heuristics for Graph Partitioning with Large k

Cedrico Knoesel

Date of submission: July 18, 2022

Reviewer: Prof. Dr. Peter Sanders

Advisors: M.Sc. Lars Gottesbüren
M.Sc. Tobias Heuer
M.Sc. Daniel Seemaier

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

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 18. Juli 2022

Abstract

Graph partitioning is a prominent problem in computer science with many different use cases. It deals with splitting the nodes of a graph into k disjoint blocks of *roughly equal* size while cutting few edges. As graph partitioning is NP-hard, heuristic approaches are used in practice, e.g., multi-level graph partitioning that uses local search heuristics or *refinement algorithms* to improve an existing solution. A common application is parallel computation where work has to be distributed to computational nodes while balancing the work load. As the numbers of computational nodes in systems are steadily increasing, the need for partitioning into a large number of blocks rises. We investigate the quality and running times of current refinement algorithms for large values of k – in the order of thousand. Hereby, we put a special focus on *high-quality partitioning.* For that purpose we implement several advanced algorithms in the shared-memory multi-level graph partitioning framework KaMinPar. We compare algorithms based on *label propagation* [34, 46], an extension of the Fiduccia-Mettheyses algorithm [39] and an algorithm that uses flow networks to minimize cuts [18]. The latter proofs to produce the solutions with the highest quality while having moderate running times. We also introduce a novel refinement algorithm that is based on *integer linear programming* (ILP) to refine clusters of blocks. We want to investigate the possibilities of this powerful tool in the domain of graph partitioning. Experiments show that albeit finding high quality solutions, the values of the cuts are still in the median 3% worse than with flow-based refinement. Furthermore, the ILP-based approach is three orders of magnitude slower than all other algorithms. Only experiments with tighter balance constraints show promising results.

Zusammenfassung

Die Graphpartitionierung ist ein prominentes Problem in der Informatik mit vielen verschiedenen Anwendungsfällen. Es beschäftigt sich mit dem Aufteilen der Knoten eines Graphen in k disjunkte Blöcke mit *ungefähr gleicher* Größe, wobei wenige Kanten geschnitten werden. Da Graphpartitionierung NP-schwer ist, werden in der Praxis heuristische Ansätze verwendet, z.B. *Multi-Level Graphpartitionierung*, die lokale Suchheuristiken oder *Refinement-Algorithmen* benutzt, um eine existierende Lösung zu verbessern. Eine gängige Anwendung ist die parallele Verarbeitung, bei der Arbeit auf Rechenknoten aufgeteilt wird, wobei die Last gleichmäßig verteilt werden soll. Da die Anzahl von Rechenknoten in Systemen stetig steigt, wächst die Nachfrage nach Partitionierung in eine große Anzahl an Blöcken. Wir untersuchen die Qualität und Laufzeit von aktuellen Refinement-Algorithmen für große Werte von k – in der Größenordnung von Tausend. Dabei fokussieren wir uns speziell auf *hochqualitative Partitionierung.* Zu diesem Zweck implementieren wir verschiedene fortschrittliche Algorithmen in dem Multi-Level Graphpartitionierungsframework KaMinPar mit geteiltem Speicher. Wir vergleichen Algorithmen, die auf *Label Propagation* [34, 46] basieren, eine Erweiterung des Fiduccia-Mattheyses Algorithmus [39] und einen Algorithmus der Flussnetzwerke nutzt, um den Schnitt zu minimieren [18]. Letzterer zeigt hierbei die Lösungen mit höchster Qualität bei moderaten Laufzeiten. Wir stellen außerdem einen neuartigen Algorithmus vor, der auf ganzzahliger linear Optimierung (*Integer Linear Programming*, ILP) basiert, um Cluster von Blöcken zu verbessern. Wir wollen die Möglichkeiten dieses leistungsfähigen Werkzeugs im Bereich von Graphpartitionierung untersuchen. Experimente zeigen, dass obwohl hochqualitative Lösungen gefunden werden, die Werte der Schnitte trotzdem im Median 3% schlechter sind, als mit flussbasiertem Refinement. Desweiteren ist der ILP-basierte Ansatz drei Größenordnungen langsamer als alle anderen Algorithmen. Nur Experimente mit strikteren Bedingungen an die gleichmäßige Verteilung zeigen erfolgversprechende Ergebnisse.

Contents

1 Introduction

Graphs are a prominent model to represent real-world structures. They are used throughout various application domains where objects are related to each other. One important processing task is *balanced graph partitioning*, i.e., splitting the nodes of the graph into k disjoint blocks of *roughly equal* size such that the number of edges running between different blocks is minimized. Graph partitioning is often used as a building block for many different graph algorithms. Note that this problem is NP-hard [15, 6]. In addition, no constant factor approximation exist [6]. Therefore, heuristic approaches are used in practice. *Multi-level graph partitioning* is a prominent metaheuristic. It uses *local search heuristics* or *refinement algorithms* to successively improve a given partition by moving nodes between the blocks.

A notable application of graph partitioning can be found in the distribution of work across parallel machines, e.g., in the field of scientific simulations [42, 2]. Here, work units are modeled as nodes and necessary communication between them as edges. Minimizing the edges between different machines can reduce the overall communication overhead, as communication within one is significantly faster than between different machines. We can use balanced graph partitioning to assign each work unit to one of the k available machines such that the connections between them are minimized while balancing the workload across them. Further applications are the sharding of data [46], optimization of VLSI circuit design [28] or as preprocessing step for finding shortest paths [35].

For some applications it is sufficient to partition the graph only once. Here solution quality is more important than speed, as for example a better distribution of data can improve the execution times of any future operation on the data. In total, this outweighs the additional partitioning time. One recent advance in the field of high quality partitioning is the shared-memory framework Mt-KaHyPar by Gottesbüren et al. [18]. Here they use, among others, flow-based local search to improve the solution quality. With this approach, they outperform many other partitioner in terms of quality. However, it is more designed for partitioning into a small number of blocks. They analyze the performance and quality only for $k \leq 128$. This is analogous to traditional research which also focused on rather small values for k.

Though, the need for partitioning into a large number of blocks is steadily increasing due to recent developments in the application domains. For example, current supercomputers for high-performance computing reach up to millions of cores [45] which requires partitioning into many blocks for efficient work distribution. Only recently, solutions tailored for large k have been introduced. One notable example is the shared-memory framework KaMinPar by Gottesbüren et al. [20]. They modify a state-of-the-art heuristic, the *multi-level graph partitioning*, to efficiently handle partitioning into large k. They use however rather simple local search heuristics.

Our goal in this work is now to combine both worlds: Partitioning into large k while using advanced local search heuristics to improve the solution quality. Therefore on the one hand, we want to examine the performance of current refinement algorithms for this case. On the other hand, we want to explore novel methods using *integer linear programming* (ILP). Using ILPs for local search or graph partitioning in general is not a highly studied field, albeit being a prominent general purpose tool with many highly optimized solution strategies. Further, ILPs are a powerful tool and have the potential of providing high quality solutions. For example, Henzinger et al. [23] use ILPs to improve solution quality and their results are promising. However, a downside is the computational expense of solving ILPs.

1.1 Problem Statement

In this thesis, we analyze the solution quality and running times of different local search heuristics for graph partitioning with large k. A novel refinement algorithm using integer linear programming should be developed. This algorithm should then be compared with other advanced refinement algorithms, considering quality and running times. All algorithms should be integrated into the shared-memory graph partitionier KaMinPar. The goal is to improve the solution quality of KaMinPar and evaluate the potential of integer linear programming in local search heuristics.

1.2 Contribution

We propose a refinement algorithm that is based on ILPs. Hereby, we construct an ILP from a set of possible move candidates for a subset of the nodes. The ILP encodes information about the improvement of the objective function for each individual node move, while considering balance and conflicts between adjacent nodes. It is used to decide which subset of the moves should be executed to optimize the partition. We integrate this novel algorithm into the partitioner KaMinPar [20]. Further, we implement a sequential k-way variant of the Fiduccia-Mattheyses algorithm [12, 39], the Ugander-Backstrom algorithm [46] and a refinement algorithm based on swapping nodes between blocks [16]. Additionally, we integrate flow-based refinement by directly using the implementation from the Mt-KaHyPar [18] framework. The results show that for $1000 \leq k \leq 1600$ flow-based refinement provides the overall highest solution quality while having moderate running times. The FM-algorithm also provides promising results with the second highest quality, and is also faster than flow-based refinement. Our ILP-based refinement is more than an order of magnitude slower than all other algorithms without producing solutions with the highest quality. However, using stricter limits for the block weights increases the quality relative to the other algorithms.

1.3 Outline

We introduce required notation and define the graph partitioning problem, (integer) linear programming and flow networks in Section 2. In Section 3, we present related work concerning refinement algorithms and the multi-level graph partitioning heuristic. We present our ILP-based refinement algorithm in Section 4. Then, we outline our implementations of the other evaluated refinement algorithms in Section 5. In Section 6 we present our experimental results before concluding our findings and discussing future work in Section 7.

2 Preliminaries

Let $G = (V, E)$ be an undirected graph with node weights $c : V \to \mathbb{N}_{\geq 0}$ and edge weights $\omega : V \times V \to \mathbb{N}_{\geq 0}$. The weight $\omega(u, v)$ for nonadjacent nodes u, v is defined to be 0. We extend the weight functions to sets, i.e., $c(V') := \sum_{v \in V'} c(v)$ and $\omega(E') := \sum_{\{u,v\} \in E'} \omega(u, v)$. We define $n := |V|$ and $m := |E|$ as the number of nodes and edges respectively. We write $G[V']$ for the subgraph of G induced by the subset $V' \subseteq V$. We further define a *contraction* of a subset $V' \subseteq V$ of nodes as follows. The nodes in the subset are contracted into the single node v' with the weight $c(v') = c(V')$. Possible parallel edges are replaced with a single edge with accumulated weight, i.e., $\forall u \in V \setminus V' : \omega(v', u) = \omega(V' \times \{u\})$.

A solution to the *graph partitioning problem* for a number $k \in \mathbb{N}_{>1}$ is a partition of a graph into non-empty *blocks* of nodes $\Pi := \{V_1, \ldots, V_k\}$, i.e., $V_1 \cup \cdots \cup V_k = V$ and $\forall i \neq j : V_i \cap V_j = \emptyset$. We use $\Pi[u]$ to denote the block in which the node u is in the current partition Π. The special case $k = 2$ is called *bipartitioning.* The *balance constraint* further demands for a partition that

$$\forall i \in \{1, \ldots, k\} : c(V_i) \leq L_k := (1 + \varepsilon) \left\lceil \frac{c(V)}{k} \right\rceil \qquad (\textit{balance constraint})$$

for some imbalance parameter ε.[1] A partition that maintains the balance constraint is called *feasible.* For a given partition the set $E_{ij} := \{(u, v) \in E \mid u \in V_i, v \in V_j\}$ for any $i \neq j$ is the set of *cut edges* between the blocks V_i and V_j. The objective is to minimize $\text{cut}(\Pi) := \sum_{i<j} \omega(E_{ij})$, the *total cut* of a partition. We call a node with an incident cut edge a *boundary node.* The set of all boundary nodes of a block is called its *boundary.* An abstract view of a partition Π for a graph G is the *quotient graph* $\mathcal{Q}(\Pi, E_\Pi)$ where each block of the partitioned graph is represented by a single node. Two blocks are connected via an edge if they share a non-empty cut, i.e., $E_\Pi := \{(V_i, V_j) \in \Pi \times \Pi \mid E_{ij} \neq \emptyset\}$. The weight of an edge equals the sum of the weights of all cut edges between the corresponding blocks.

For each node u in block V_i we define the *gain*

$$g_j(u) := \omega(\{(u, v) \in E \mid v \in V_j\}) - \omega(\{(u, v) \in E \mid v \in V_i\}) ,$$

that is the change in the total cut when moving u to block V_j. If the target block V_j is given in the context, we use the short notation $g(u)$. Note that the gain can be either positive or negative.

(Integer) Linear Programming. Linear Programming (LP) is a method to describe optimization problems subject to linear constraints. The goal is to find a vector of variables $x = (x_1, \ldots, x_n)^T$ that minimizes or maximizes a given objective function. The linear constraints restrict the solution space. Mathematically we can define this as follows:

Definition 2.1 (Linear Program)**.** *Let $A \in \mathbb{R}^{m \times n}$ be a matrix, $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$ be vectors. A linear program (LP) is defined as*

$$\begin{array}{ccc} \textit{maximize} & c^T \cdot x & \textit{s.t.} \\ & Ax \leq b & \end{array}$$

where $x \in \mathbb{R}^n$ is a vector of variables, $c^T \cdot x$ is called the objective function and $Ax \leq b$ represents the linear constraints. Note that the objective function can be not only maximized but also minimized.

[1]There is an ongoing debate on the definition of the upper bound for the balance constraint. Some authors use $L_{max,k} := \max\{(1 + \varepsilon)\frac{c(V)}{k}, \frac{c(V)}{k} + \max_v c(v)\}$ as upper bound.

A common metric for a given LP is the number of *non-zeroes*, that is the number of non-zero entries in the matrix A. An *Integer Linear Program* (ILP) is a linear program where each solution x must be a vector of integer values. Note that while linear programming is in P [27], integer linear programming is NP-hard [26]. Accordingly, we can use ILPs to model many different problems. In addition, much research has already been done in this area and many efficient solvers have been developed. This recommends reducing a new problem to ILP and using a solver for the latter, instead of developing a completely new algorithm.

One strategy for solving an ILP is the *Branch-and-Bound* algorithm [4]. We now shortly describe it. Let therefore P be any integer linear program. The algorithm first relaxes P to a linear program P'. This only removes the integrality restrictions. It then computes an optimal solution x' for P'. If all variables in x' already have integer values, x' is also an optimal solution for P. In the other case, we begin branching into the two ILPs P_1 and P_2. Let x_i be any variable in x' with a fractional value a. The two new problems are now defined as

$$\begin{aligned} P_1 &= P \,\cup\, \{x_i \leq \lfloor a \rfloor\} \\ P_2 &= P \,\cup\, \{x_i \geq \lceil a \rceil\} \end{aligned}$$

where the union of the ILP P and a linear constraint means that the latter is added to the set of constraints of P. Note that P_1 and P_2 are more restrictive versions of the original ILP that both prevent $x_i = a$ to be part of a valid solution. Note however that for any optimal solution for P, one of the additional constraints is fulfilled. The algorithm then recursively solves P_1 and P_2 by possibly branching further. This is repeated until the relaxation returns an integer solution, which is then propagated to the higher levels. From the solutions for P_1 and P_2, the algorithm picks the one with the higher objective. This gives us an optimal solution for P. Each generated ILP in this method is called a *node* of the *search tree.*

Flows. A flow network is a directed graph $\mathcal{N} = (\mathcal{V}, \mathcal{E}, c)$ where each edge $e \in \mathcal{E}$ additionally has a *capacity* $c(e) \geq 0$. It has furthermore a dedicated source $s \in \mathcal{V}$ and sink $t \in \mathcal{V}$. The goal is to find a function $f : \mathcal{V} \times \mathcal{V} \to \mathbb{R}$ that assigns to each edge a number of flow units that are sent over this edge. If f satisfies the following three constraints, it is called a (s,t)-flow.

$$\begin{aligned} \forall u, v \in \mathcal{V} :\ & f(u,v) \leq c(u,v) && (\textit{capacity constraint}) \\ \forall u, v \in \mathcal{V} :\ & f(u,v) = -f(v,u) && (\textit{skew symmetry constraint}) \\ \forall u, v \in \mathcal{V} \setminus \{s,t\} :\ & \sum_{v \in \mathcal{V}} f(u,v) = 0 && (\textit{flow conservation constraint}) \end{aligned}$$

The value of a flow is defined by $|f| := \sum_{v \in \mathcal{V}} f(s,v) = \sum_{v \in \mathcal{V}} f(t,v)$, i.e., the total amount of flow units sent from the source to the sink. A flow f is called a *maximum* flow if no other flow f' exists with $|f| < |f'|$. For a given flow f we define the *residual capacity* as $r_f(e) := c(e) - f(e)$. An edge with residual capacity $r_f(e) = 0$ is called *saturated.* The residual capacities induce a *residual network* or *residual graph* $\mathcal{N}_f = (\mathcal{V}, \mathcal{E}_f, r_f)$ where $\mathcal{E}_f := \{(u,v) \in \mathcal{V} \times \mathcal{V} \ \mid \ r_f(u,v) > 0\}$.

According to the max-flow min-cut theorem [13], the value $|f|$ of a maximum flow equals the weight of a minimum (s-t)-cut. The latter is a set of edges that disconnect s and t. The value of such a cut is the sum of the capacities of the contained edges. Each (s-t)-cut induces a node bipartition $(\mathcal{S}, \mathcal{V} \setminus \mathcal{S})$ with $s \in \mathcal{S}$ and $t \in \mathcal{V} \setminus \mathcal{S}$. The subsets $\mathcal{S}$ and $\mathcal{V} \setminus \mathcal{S}$ can be also calculated from a maximum flow f by traversing the corresponding residual graph $\mathcal{N}_f$ from the source or the sink, respectively.

3 Related Work

Graph partitioning is a highly researched area. Therefore, we refer the reader to survey papers [5, 7, 8, 43, 49] for a general overview of the topic. Here, we introduce local search heuristics that are relevant for this work. Further, we present the *multi-level graph partitioning* which is the metaheuristic that we used for our evaluation.

3.1 Refinement Algorithms

We start with presenting different *refinement algorithms* based on local searches. A refinement has a graph G with an already feasible partition $\Pi := \{V_1, \dots, V_k\}$ as input. The goal is to reduce the total cut by moving nodes between the blocks. The algorithms can be based on moving single nodes or whole groups at once.

3.1.1 Kernighan-Lin

One of the first refinement algorithms was introduced by Kernighan and Lin [32]. The motivation is that for each bipartition V_1, V_2, there exist two subsets $A \subset V_1, B \subset V_2$ such that swapping these sets results in a partition with minimum total cut. Finding these sets A, B is however NP-hard. Hence, Kernighan and Lin propose a greedy algorithm to approximate these sets. The general idea of this algorithm is to iteratively find pairs of nodes in different blocks that decrease the total cut when swapped. In the following, we first describe the basic version of the algorithm, that is designed to refine a perfectly balanced bipartition. Afterwards, we present the extension to k-way partitioning with $\varepsilon \geq 0$.

2-way Partitioning Algorithm. They start with an arbitrary partition V_1, V_2 and calculate the gains $g(v)$ for each node $v \in V$ when moved to the other block. The next step is to find a pair $(a_1, b_1) \in V_1 \times V_2$ that maximizes the swap gain

$$g(a_1, b_1) = g(a_1) + g(b_1) - 2\omega(a_1, b_1) \ , \tag{3.1}$$

that is the decrease in edge cut when swapping a_1 and b_1. Note that if a_1 and b_1 are adjacent, double the weight of their connection has to be subtracted to get the actual gain. They save this pair (a_1, b_1) with maximum gain and remove it from further calculations. Afterwards they recalculate the gains for the remaining nodes as if a_1 and b_1 had been swapped. With the updated gain values they calculate and save a new maximum pair (a_2, b_2). This is repeated until all nodes have been picked once. Thus, they obtain a sequence of pairs $(a_1, b_1), \dots, (a_n, b_n)$. The partial sum $G_K = \sum_{k=1}^{K} g(a_k, b_k)$ is hereby the total gain when swapping the top K pairs. Choose now $K := \arg\max_{0 \leq k \leq n} G_k$, i.e., the number of pairs that maximize the gain when swapped. Now, they swap the pairs $(a_1, b_1), \dots, (a_K, b_K)$ such that $b_i \in V_1$ and $a_i \in V_2$ for each $1 \leq i \leq K$. Afterwards the whole procedure is repeated with the improved partition as initial solution. Note that if no partial sum is positive, the optimum is to swap no nodes. If this is the case, the refinement is stopped as they reached a local minimum. The authors provide an implementation with a running time of $\Omega(n^2 \log n)$. However, Dutt [11] could reduce it to $\Theta(\max(m \cdot \Delta, m \log n))$ where Δ is the maximum degree of the graph.

Extension to k-way partitioning. For the general graph partitioning problem, Kernighan and Lin [32] propose the following algorithm that uses the procedure for 2-way partitioning described in the previous paragraph. They start with an arbitrary partition $V_1, \dots, V_k$. The refinement algorithm for 2-way partitions is then used to refine pairs of the k blocks. They repeat the pairwise refinement for several rounds, but only reschedule a pair if one of the blocks changed in the last iteration. Further, they extend the algorithm to also handle $\varepsilon > 0$ by adding isolated nodes. This enables that a perfectly balanced partition of a graph with the additional nodes results in a feasible partition after removing them. Note that the cut is not affected by isolated nodes.

3.1.2 Fiduccia–Mattheyses Algorithm

An improvement to the Kernighan-Lin algorithm was proposed by Fiduccia and Mattheyses [12]. Their refinement algorithm (FM-algorithm) was originally designed for bipartitioning hypergraphs. Note however that graphs are special cases of hypergraphs. One difference to the Kernighan-Lin algorithm is that they move a single node in each step, instead of swapping pairs. Further, they reduced the running time to $O(m)$ by using a bucket array for saving the gain values of move candidates. This significantly reduces the time required to find the maximum gain move and to update gain values after moving nodes. We present further details of the original bipartitioning algorithm in the following. Afterwards, we present modifications to this version that remove the restriction to $k = 2$ blocks.

In each step of the FM-algorithm, the nodes with the currently highest gain of each block are eligible for moving. Let these nodes be $v_1 \in V_1$ and $v_2 \in V_2$. If moving v_1 into V_2 would violate the balance constraint, v_2 is moved and vice versa. In case both nodes can move without violating the constraint, they move the node with the higher gain. Ties are broken by the better resulting balance. Note that they explicitly allow negative gain moves to hopefully escape a local minimum. After moving a node, the algorithm updates the gain values of the other nodes and the now two highest gain nodes become eligible. To guarantee linear running time, each node can move at most once. When all nodes were moved once or the balance constraint prevents further moves, it rolls back to the best found partition during the move sequence. This algorithm can be repeated until no improvement of the total cut is achieved.

k-way Local Search. One method to extend any bipartitioning algorithm to the k-way partitioning problem is recursive bisection. With this strategy, we first calculate a bipartition of the original graph. Each block is then recursively split, until we reach the desired number of blocks. This approach allows directly using 2-way refinement algorithms, however Simon and Teng [44] have shown that recursive bisection can create solutions that are very far away from the optimum. Hence, we are interested in direct k-way partitioning algorithms.

Several approaches exist to extend the FM-algorithm to k-way partitioning. An early modification to it was proposed by both Sanchis [38] as well as Hendrickson and Leland [22]. They use $k(k-1)$ priority queues, i.e., two for each block pair. Each queue P_{ij} contains all possible moves from V_i to V_j. The priority of a move is its gain. Similar to the original FM-algorithm, they move in each step the node with the highest gain that does not violate the balance constraint.

Karypis and Kumar [30] improve this idea and use only one global priority queue P containing nodes. The priority of $v \in V$ is the maximum gain $g(v) = \max_i g_i(v)$ for the respective node. Further they initialize P with the nodes v where $g(v) \geq 0$, i.e., a subset of the boundary nodes. This improves the performance, as previously all possible moves of all nodes were maintained in a queue. Otherwise, the algorithm works similar to the original algorithm: They select in each step the top node v in P and find the block V_i that maximizes the gain $g_i(v)$ while satisfying the balance constraint. Then they move v into V_i. Note that the gain of the executed move can differ from the priority as they ignore the balance constraint when initializing the queue. Hence, also moves with negative gains are possible.

Osipov and Sanders [36] introduce a highly localized variation. They initialize the search with single nodes. After moving a node its unmoved neighbors become eligible to move. They further introduce an adaptive stopping criterion. This allows to stop a search early if further improvements are unlikely. Starting from their approach, Sanders and Schulz [39] introduce a FM variant called *multi-try k-way FM*. They add all boundary nodes into a todo list T. Then they start a k-way local search from one random node $v \in T$. Afterwards they remove v from T and start a new local search with a different initial node from T. Note that each local search is restricted to nodes that have not been touched by any other local search before. This allows an implementation in linear time, as each node is processed at most once.

We now shortly motivate the localization of the searches. Consider the case when the current partition is in a local optimum, such that each boundary node has negative gain. To escape, we have to move at least two nodes as the positive gain move is *hidden* behind the boundary. If we now start a local search initialized with the whole boundary, we likely move many other nodes before we *find* the correct move that allows us to escape the local minimum. However the other moves maybe have worsened the partition too much already. If we localize the searches, we rather roll back a move sequence when we do not find any improvement and start a new search in a different location.

3.1.3 Label Propagation

The *label propagation* algorithm was introduced by Raghavan et al. [37] and is originally developed for *community detection*. This is a research field similar to balanced graph partitioning. The task is to partition the node set as well, however the number of blocks is not fixed in advance. Further the sizes of the block are not restricted. Instead of only minimizing the cut edges, each block should represent a *community*. This is a group of nodes that share some structural similarities, for example a clique, i.e., pairwise adjacent nodes. The algorithm now works as follows: Each node has a label representing its block. Initially each node has its own label, i.e., each node is in its own block. They then iterate over all nodes in random order. When visiting a node u, they move u to the block V_i that maximizes the gain $g_i(u)$. Note however that a node is not moved if the maximum gain is negative. Ties are broken uniformly randomly. This is repeated until convergence.

Meyerhenke et al. [34] introduce a modification of this algorithm, called *size-constrained label propagation*. Their version allows limiting the size of the resulting blocks. This enables them to use this algorithm for refining a balanced partition. Therefore, they initialize each node with its block in the current partition. Then they again traverse the nodes and greedily move them to the best blocks. However, for each node u they only consider moves into blocks V_i such that the balance constraint is respected, i.e., $c(V_i) + c(u) \leq W_{max}$ where W_{max} is the maximum weight of a block. Thus, they always maintain a balanced partition after each move. Again, they repeat the traversals until convergence, however they also stop after a maximum number repetitions.

Ugander-Backstrom Algorithm. Another refinement algorithm that is based on the idea of label propagation was proposed by Ugander and Backstrom [46]. The main difference to (size-constrained) label propagation is that instead of directly moving nodes to the best block, they gather these moves and later synchronously execute as many of them as possible while maintaining balance. They use linear programming to decide how many nodes can move between the different block pairs while respecting given lower and upper bounds for the block sizes. Note that the algorithm is only designed for unweighted graphs. We now present further details of this algorithm.

They start with gathering the moves that would be performed by classical label propagation. Therefore, let $V(i,j)$ be the nodes in V_i that would move to V_j, i.e., $V(i,j) := \{u \in V_i \mid g_j(u) = \max_{j'} g_{j'}(u)\}$. They then sort each $V(i,j)$ in decreasing gain which results in the ordered sequence of nodes $u_1, \dots, u_K$ with corresponding gains $g_j(u_1) \geq \dots \geq g_j(u_K) \geq 0$. For each of the sequences, they define the *relocation utility function*

$$f_{ij}(x) := \sum_{k=1}^{x} g_j(u_k)$$

that describes the total gain by moving the leading x nodes from V_i to V_j. They now want to find for each block pair (V_i, V_j) how many of the top gaining nodes should be moved to maximize the gain while maintaining balance. Therefore, they introduce for each f_{ij} a variable x_{ij} and maximize the sum $\sum_{i,j} f_{ij}(x_{ij})$. Further, they introduce the constraints

$$\begin{aligned} \forall i: \quad & S_i \leq & |V_i| + \sum_{j \neq i} (x_{ji} - x_{ij}) &\leq T_i \\ \text{and } \forall i,j: \quad & 0 \leq & x_{ij} &\leq |V(i,j)|, \end{aligned}$$

where the first restricts the size of each block V_i after moving the respective number of nodes into and out of the block. The second constraint bounds the variables, as at most all nodes in $V(i,j)$ can move. Note that this is not yet a linear program, as f_{ij} is not linear. However, they provide a translation into one. Note therefore that each f_{ij} is a piecewise linear concave function because $g_j(u_k) \geq 0$ and $g_j(u_k) \geq g_j(u_{k+1})$. Note further that any piecewise linear concave function $f(x)$ can be transformed to $f(x) = \min_{k=1,\dots,l}(a_k x + b_k)$, for appropriate a_k's, b_k's and l. This leads to the following linear program:

$$\begin{aligned} & \max_{X,Z} \sum_{i,j} z_{ij} \quad s.t. & & \\ S_i \leq \quad & |V_i| + \sum_{j \neq i} (x_{ji} - x_{ij}) \leq T_i, & & \forall i \\ 0 \leq \quad & x_{ij} \leq |V(i,j)|, & & \forall i,j \\ & -a_{ijk} x_{ij} + z_{ij} \leq b_{ijk}, & & \forall i,j,k \end{aligned}$$

where the constants a_{ijk} and b_{ijk} can be directly derived from the relocation utility functions f_{ij}. Solving this linear program returns for each block pair (V_i, V_j) the number x_{ij} of nodes that should be moved to maximize the gain while maintaining balance. Afterwards the algorithm moves the top x_{ij} nodes of each pair accordingly.

This procedure of gathering moves, solving the induced linear program and executing the selected moves is one refinement round. The algorithm can be repeated for several rounds. One drawback of this algorithm is that it is only designed for unweighted graphs. Hence, it can not be used on every level of the multi-level partitioning, a prominent metaheuristic for graph partitioning. In Section 5, we introduce a weighted variant for the Ugander-Backstrom algorithm. Note that the latter variant uses integer linear programming instead of linear programming.

3.1.4 Flow-Based Refinement

Sanders and Schulz [39] proposed an algorithm based on flows to refine a partitioned graph. It was generalized to hypergraphs by Heuer et al. [24], further refined by Gottesbüren et al.

[17] and parallelized by Gottesbüren et al. [18]. It is a pairwise refinement algorithm, i.e., they refine pairs of blocks. It can move groups of nodes, however only between two blocks at once. The general idea is to find a minimum cut, that separates two blocks. Therefore they construct a flow network and find a maximum flow, which induces a minimum cut. They then move the nodes such that this minimum cut is the new boundary between the blocks. We present further details of the implementation in Section 5.

3.1.5 Refinement with Integer Linear Programming

Henzinger et al. [23] introduce an integer linear program that solves balanced graph partitioning to optimality. It is a generalization of a formulation for balanced bipartitioning. This does however not scale well to large graphs. Hence, they combine the ILP with local search heuristics to develop a refinement algorithm. We start with presenting the ILP and then show its integration into a heuristic refinement algorithm.

For the ILP, they introduce a binary decision variable $e_{uv} \in \{0,1\}$ for every edge $(u,v) \in E$, that decides whether (u,v) is a cut edge. The goal of the ILP is to find a partition with minimal total cut. This translates into the objective function

$$\min \sum_{(u,v)\in E} e_{uv} \cdot \omega(u,v) \ .$$

They further introduce for every node $u \in V$ and block V_i a variable $x_{u,i} \in \{0,1\}$, which is one if u is in block V_i. Every node has to be assigned to exactly one of the k blocks, hence they add the constraint

$$\forall u \in V : \quad \sum_{1\leq i\leq k} x_{u,i} = 1$$

which guarantees that exactly one variable per node equals 1. Further, the edge variables have to be constrained to correctly represent its cut state. This means if both endpoints are in different blocks, the variable has to equal 1 and 0 otherwise. Therefore, they add the following two constraints:

$$\begin{aligned} \forall (u,v) \in E, \forall 1 \leq i \leq k : \quad & e_{uv} \geq x_{u,i} - x_{v,i} \\ \forall (u,v) \in E, \forall 1 \leq i \leq k : \quad & e_{uv} \geq x_{v,i} - x_{u,i} \end{aligned}$$

If both endpoints of an edge (u,v) are in the same block, i.e., $\exists i : x_{u,i} = x_{v,i}$, every right-hand side of the inequalities equals 0. Because the objective minimizes the sum of the edge variables, e_{uv} correctly equals 0. Consider now the case where the endpoints are in different blocks. Here one of the right-hand sides is 1, i.e., e_{uv} is correctly constrained to be 1. Lastly, they maintain balance with the constraint

$$\forall 1 \leq i \leq k : \quad \sum_{u\in V} x_{u,i} \cdot c(u) \leq L_k$$

where L_k is the maximum block weight. Minimizing the objective subject to the above constraints results in an optimal partition. Note however, that the size of the ILP grows linear with the number of nodes and edges and therefore does not scale to large inputs.

We now present their refinement algorithm that uses the above ILP. Their main idea is to use the ILP to partition a coarse representative, called *model*, of the original graph with less nodes and edges. Therefore they select a subset of nodes $\mathcal{K} \subseteq V$ that are expected to be close to the optimal cut. They contract the other nodes into one node per block of the original partition. This means each $\mathcal{V}_i := V_i \backslash \mathcal{K}$ is contracted into the single node v_i. Recall that a contracted node aggregates the node weights and edges. Their model now contains $k + |\mathcal{K}|$ nodes. Then they use the ILP to find an optimal partition of the model. This partition translates to a partition of the original graph by assigning the contracted nodes $v \in V_i$ to the block of its coarse representative v_i. Note that the partition has the same cut and balance properties due to the definition of a contraction. However, this partition is not necessary optimal, as the nodes of each $\mathcal{V}_i$ can not be assigned to different blocks. The quality is restricted by the selection of $\mathcal{K}$, i.e., the nodes that are not contracted.

They propose several strategies to find this subset $\mathcal{K} \subseteq V$. The best strategy according to their experiments is to find the nodes via a breadth first search that is initialized with all nodes whose maximum gain is greater than or equal to ρ, i.e., $\{v \in V \mid \max_i g_i(u) \geq \rho\}$. Here ρ is a tuning parameter. With this strategy they select on the one hand nodes with high improvement potential and on the other hand nodes that can counterbalance the block weight changes of the moves.

3.2 Multi-Level Graph Partitioning

One metaheuristic for partitioning graphs is *multi-level graph partitioning* (MGP) [22]. The motivation of this heuristic is to combine a more global view on the graph with local improvements. Many current graph partitioners are based on this approach. A few examples are KaHIP [40], KaMinPar [20], Metis [29] and Jostle [49]. This metaheuristic consists of three steps: *coarsening*, *initial partitioning* and *refinement*.

The goal of the first step (*coarsening*) is to reduce the size of the original graph. This is done by creating a hierarchy of successively smaller graphs by contracting matchings or clusters of nodes. This allows a more global view of the graph on the coarse levels. In the *initial partitioning* phase, a partition of the coarsest graph is calculated. A more expensive algorithm can be used because of the decreased size of the graph. Many partitioners use a portfolio approach composed of multiple different algorithms [41, 28, 20]. This initial solution is propagated to the finer levels in the following *refinement* phase. We can extend the partition of the coarsened graph to the finer graph by assigning each node to the block of its coarse representative. Note that this partition has the same cut and balance properties as the coarsened graph. In each uncoarsening step, refinement algorithms search for possible improvements of the current cut.

The multi-level graph partitioning combines at least two advantages: Firstly, on the coarse level, groups of nodes can be moved in constant time. This global view on the graph is beneficial for finding major improvements that would be harder to find on the finer levels [7]. Secondly, the refinement on finer levels start from a good initial solution. This can improve the overall running time, because the starting partition impacts the performance of many refinement algorithms [32, 46, 7].

Deep Multilevel Graph Partitioning. One improvement to the general multi-level graph partitioning is *Deep Multilevel Graph Partitioning* (deep MGP) by Gottesbüren et al. [20]. It is especially designed for handling partitioning with large values for k, i.e., in the order of millions.

Two typical strategies for the initial partitioning in classical MGP are *direkt k-way partitioning* and *recursive bipartitioning*. The former directly calculates a k-way partition on the coarsest representation of the graph. Therefore the coarsest graph has to have more then k nodes left, usually kC nodes for an input parameter C. This means however that for large values of k, even the coarsest graph still has relative many nodes. This contradicts the design of MGP and the initial partitioning can heavily impact the overall running time. Therefore many systems use recursive bipartitioning after fully coarsening the graph when handling large k [30, 39, 19]. This method first calculates a bipartition of the coarsest graph. It then recursively further divides both blocks until it obtains the desired k blocks. This can however become a scalability bottleneck when executing in parallel, especiall for large k [1].

Deep multilevel graph partitioning now tries to combine the advantages of both methods, while mitigating the drawbacks. It starts with *coarsening* the graph until $2C$ nodes are left. They then calculate a *bipartition* of the coarsest graph. During uncoarsening they successively increase the number of blocks for the respective levels up to the final value for k. They use recursive bipartitioning to further divide a partition from lower levels. The expected number of blocks per level is chosen such that each bipartitioning handles roughly $2C$ nodes. Further they use a *balancer* after uncoarsening to always guarantee that the balance constraint is fulfilled. On each level they use a *refinement algorithm* to improve the current partition as in classic MGP. When executing in parallel, in coarse levels, they copy the graph to fully use the potential of all processing units. This is because the graph may become too small to be efficiently processed by several units. They also justify the overhead of copying because it diversifies the search.

KaMinPar. Gottesbüren et al. [20] also present an implementation of the deep multilevel graph partitioning scheme called KaMinPar. We used this framework as a basis for evaluating the different refinement algorithms in this work. We now shortly describe the original implementations that they used for the *coarsening*, *bipartitioning* and *refinement*.

For coarsening they use size-constrained label propagation [34] with a fixed upper bound for the weight per cluster. This bound is dependent to k, ε and the total node weight. The label propagation stops after at most five rounds. For the initial bipartitioning, they use a multilevel algorithm. In this algorithm they again use label propagation for coarsening until it converges. To initially partition the coarsest graph, they use random bipartitioning, breadth-first searches and greedy graph growing [29]. They select the partition with the lowest total cut from several tries of each algorithm. To refine the initial bipartition, they use 2-way FM [12]. After uncoarsening this initial partition to the finer levels, they use the same size-constrained label propagation as in the coarsening step to refine the resulting partition.

We used this implementation mostly unchanged. The only modification was replacing the refinement algorithm with the other algorithms that are presented and compared in this work.

4 Refinement Based on Integer Linear Programming

Many refinement algorithms are based on moving single nodes at a time to improve the total cut, whereby balance has to be respected after every move. However, this can make it difficult to escape a local minimum, especially for strict balance constraints. In a local minimum, no node can be moved to improve the cut without violating the balance constraint. At least one node must be moved, temporarily worsening the result, to *free up space* in a block. Then a different node can move into this block which possibly overcompensates the temporary loss. Without considering multiple moves at once, it is however difficult to decide which negative moves may later result in an overall gain. Therefore, approaches exist that move multiple nodes at once, for example flow-based refinement. The latter has proven to compute high-quality solutions [18]. However, this technique only refines the blocks pairwise, which hampers finding more complicated move sequences between multiple blocks. We want to combine both worlds: Direct k-way refinement that moves groups of nodes.

Based on this intuition, we propose an ILP-based refinement algorithm as sketched in Algorithm 1. The algorithm works in rounds. In each round, we start by selecting a set of move buffers $\mathcal{M}$. Each move buffer $M \in \mathcal{M}$ consists of heuristically chosen *move candidates*, i.e., pairs (u, b) of nodes u and target blocks b. Note that a move buffer can contain multiple moves for the same node. We then proceed by constructing an ILP for each move buffer $M \in \mathcal{M}$, which selects the subset $M_{OPT} \subseteq M$ of gain-maximizing moves. Using multiple move buffers allows us to solve several smaller ILPs for different regions of the graph, rather than a single large one. Note that we restrict moves in different buffers to not interfere with each other (see Section 4.4 for further details). Finally, we apply the optimal moves M_{OPT} to the current partition and repeat the process for the next move buffer. After the last move buffer, we continue with the next round or terminate if we exceeded the configured maximum number of rounds i_{max}.

Algorithm 1: High-level structure of the ILP refinement algorithm.

```
  Data: $G = (V, E)$, initial partition $\Pi := \{V_1, \dots, V_k\}$
1 for $i = 0$; $i < i_{max}$; $i$++ do
2     $\mathcal{M}$ := selectMoves($G$, $\Pi$)
3     foreach $M \in \mathcal{M}$ do
4         $M_{opt}$ := solveILP($G$, $\Pi$, $M$)          // construct and solve the ILP
5         execute moves $M_{opt}$
```

For this section, we introduce the notation m_i^u to describe the move of the node u to block V_i. Note that we do not consider m_i^u to be a valid move if u is already in block V_i. A set of moves M is *feasible* if no two moves for the same node exist in M and if the partition remains feasible after executing all moves. Informally, this means that we can safely execute all moves in M with respect to balance. We further define for a set M of moves the *local gain* $g^L(M)$ as the sum of the individual gains, i.e.,

$$g^L(M) := \sum_{m_i^u \in M} g_i(u) .$$

We further define the *actual gain* $g^A(M)$ as the cut reduction after executing all moves, i.e.,

$$g^A(M) := \mathrm{cut}(\Pi) - \mathrm{cut}(\Pi') ,$$

where Π is the original partition and Π' the partition after executing the moves in M. Note that the local gain and actual gain can differ if M contains moves for adjacent nodes (see Section 4.2 for further insights).

We call two moves $M' := \{m_i^u, m_j^v\}$ of two different nodes $u \neq v$ *conflicting*, if their local gain differs from the actual gain, i.e., $g^L(M') \neq g^A(M')$. Otherwise they are *conflict-free*. We extend these definitions to pairs of nodes $u, v \in V$, i.e., they are *conflict-free* if and only if *no* conflicting pair of moves $\{m_i^u, m_j^v\}$ exists.

This section is structured as follows: In Section 4.1 we introduce a basic version of the ILP that is constructed and solved in Line 4 of Algorithm 1. This version selects the subset of moves $M' \subseteq M$ that maximize the local gain $g^L(M')$. Note that this restriction simplifies the formulation of the objective function of the ILP. However as we want to maximize the actual gain, this version is only optimal if no conflicting moves exist in M. In Section 4.2, we elaborate how conflicts arise and their influence on the difference between local and actual gain. We then introduce an extended version of the ILP that optimizes directly the actual gain. Here, more constraints are necessary to accurately model conflicts. Lastly, we introduce two algorithms for the selection of move candidates in Line 2 of Algorithm 1.

4.1 Basic Integer Linear Program

In this section, we define the basic version of our ILP for selecting an optimal subset of moves $M_{OPT} \subseteq M$ such that the local gain $g^L(M_{OPT})$ is maximized. We expect the moves in M to be pairwise conflict-free. Recall that for conflict-free nodes the local gain equals the actual gain. Note that only adjacent nodes can conflict (see the following section). Hence a simple method to satisfy this restriction is to only add moves for disconnected nodes into M.

We now present the full definition of the basic ILP before shortly describing the different parts. Afterwards, we show the correctness of the objective function and constraints.

Definition 4.1 (Integer Linear Program for Conflict-Free Moves)**.** *Let $G = (V, E)$ be a graph with node weights c and let $\Pi := \{V_1, \dots, V_k\}$ be a partition of G. Let M be a set of pairwise conflict-free moves of nodes in G. The ILP for conflict-free moves is defined as*

$$\textit{maximize} \quad \sum_{m_i^u \in M} g_i(u) \cdot x_i^u \quad \textit{s.t.}$$

$$\forall u \in V : \quad \sum_{m_i^u \in M} x_i^u \leq 1 \tag{4.1}$$

$$\forall 1 \leq i \leq k : \quad W_{in}^i - W_{out}^i + c(V_i) \leq W_{max}^i \tag{4.2}$$

where each x_i^u is a binary decision variable, called move variable*. A solution vector X for these variables induces a subset of moves*

$$M_X := \{m_i^u \in M \mid x_i^u = 1\}\,.$$

Further W_{max}^i is a constant, that is the maximum weight of each block V_i and W_{in}^i and W_{out}^i represent the incoming and outgoing weight, when executing the selected moves. The latter two are linear functions of the move variables and defined as

$$W_{in}^i := \sum_{u \notin V_i, m_i^u \in M} c(u) \cdot x_i^u \qquad \textit{and} \qquad W_{out}^i := \sum_{u \in V_i, m_j^u \in M} c(u) \cdot x_j^u\,.$$

As can be seen, the ILP introduces one binary decision variable $x_i^u \in \{0,1\}$ for each move candidate $m_i^u \in M$. The move variables induce the subset of moves that should be executed. The constraint 4.1 enforces that at most one move for each node $u \in V$ is selected. Note that if M contains at most one move for a node u, i.e., $|\{i \mid m_i^u \in M\}| \leq 1$, we do not add this constraint for u, as it is always respected. The constraint 4.2 enforces that the block weights remain below their respective limits after executing the selected moves. Note that for each block arbitrary maximum weights or additional lower bounds are supported. We do not add this constraint for a block V_i if moving all possible nodes into V_i does not exceed the maximum block weight. Now, we show the correctness of the constraints and that an optimal solution to the ILP induces a subset of moves that maximizes the gain.

Theorem 4.1. *Let $G = (V, E)$ be a graph with a partition $\Pi = \{V_1, \dots, V_k\}$ and let M be a set of pairwise confict-free moves of nodes in G. For any optimal solution X to the ILP from Definition 4.1 for G, Π and M, the induced subset $M_X \subseteq M$ is feasible and has maximal actual gain, i.e., $g^A(M_X) \geq g^A(M')$ for any feasible $M' \subseteq M$.*

Proof. Let X be any optimal solution to the ILP above. We first show that M_X is feasible. Therefore, we have to show that

(i) at most on move per node is contained in M_X and

(ii) the partition remains feasible after executing all moves in M_X.

The constraint 4.1 implies that at most one move variable per node equals 1. This is due to the domain of the variables. With the definition of M_X, we can now directly conclude (i).

Further, consider any block V_i. When inserting the solution X into the definition of W_{in}^i, we see that it equals the sum of the weights of the nodes moved into V_i. Note therefore that due to (i) no node weight is counted twice. Analogously, W_{out}^i is the total weight of the nodes moved out of V_i. Hence, $W_{in}^i - W_{out}^i + c(V_i)$ is the weight of V_i after executing the moves. With constraint 4.2 we can now conclude (ii).

It remains to show the optimality of M_X. From the definition of M_X and the local gain, we can conclude that the value of the objective function equals the local gain $g^L(M_X)$ when inserting X. Because of the optimality of X, we can now follow for any solution X' to the ILP that

$$g^L(M_X) \geq g^L(M_{X'}) \; .$$

Now note that the moves in M are per requirement pairwise conflict-free, which holds for M_X as well. From the definition of conflicts, we can now conclude that $g^L(M_X) = g^A(M_X)$, from which we can conclude that M_X has maximal actual gain. □

4.2 Move Conflicts

In the previous section, we have restricted the ILP to conflict-free move candidates. To drop this restriction and allow arbitrary sets of move candidates, we have to consider the various types of conflicts that can occur when moving adjacent nodes simultaneously. To this end, we now present the various types of conflicts and their influence on the actual gain. Note that the following types are known within the research field of parallel refinement algorithms [31]. In the next section, we then extend the basic ILP with constraints that properly model the different conflicts from this section.

Before introducing the different types, we first define the *conflict value* $\chi(m_i^u, m_j^v)$ of a conflict between two moves $M := \{m_i^u, m_j^v\}$. Recall therefore that a conflict arises when the actual gain $g^A(M)$ differs from the local gain $g^L(M)$. We define

$$\chi(m_i^u, m_j^v) := g^A(M) - g^L(M) = \underbrace{(\mathrm{cut}(\Pi) - \mathrm{cut}(\Pi'))}_{\text{actual gain}} - \underbrace{(g_i(u) + g_j(v))}_{\text{local gain}},$$

where Π' is the partition after executing both moves. Hence, the sum of the local gain and the conflict value equals the actual gain, which is the desired value as it states the actual reduction of the total cut. We say that a conflict is *negative* if $\chi(m_i^u, m_j^v) < 0$ and *positive* otherwise. In the former case, we overestimate the actual gain of the two moves and underestimate it in the latter.

We now describe the possible conflicts of two moves m_i^u and m_j^v for two nodes $u, v \in V$ that are connected via an edge $e \in E$. Recall that only adjacent nodes can create a conflict. The possible values of a conflict depend on the source and target blocks of both nodes. Therefore, we inspect every combination of source and target blocks. We start with the cases where u and v are originally in different blocks and then where they are in the same block. Note that we only have to consider the edge e for the calculation of the value of the conflict. Other incident edges of u or v are not affected by moving the other node. Their cut state solely depends on the move of u or v respectively. Therefore, we ignore other edges in the following. Note that we use the short notations χ, g^L and g^A for the conflict value, local gain and actual gain of m_i^u and m_j^v, respectively.

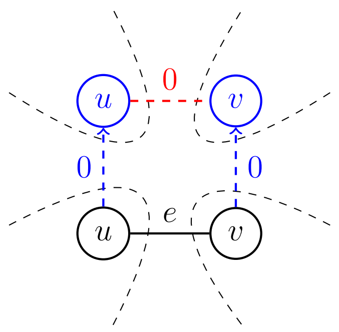
Let u be in V_a and v in V_b with $a \neq b$, i.e., the source blocks are *different*. We can observe either a negative or positive conflict between the moves m_i^u and m_j^v, depending on the target blocks V_i and V_j. We can observe four different cases in total. These are depicted in Figure 1 and described in the following.

No Conflict. The target blocks are completely different from each other and the source blocks, i.e., $V_b \neq V_i \neq V_j \neq V_a$. This case creates no conflict. Both moves have no gain individually, as the edge e remains a cut edge. After executing both moves, e again remains a cut edge and therefore no gain is achieved. This results in no difference between the local and the actual gain. Therefore the value of this conflict is $\chi = 0$.

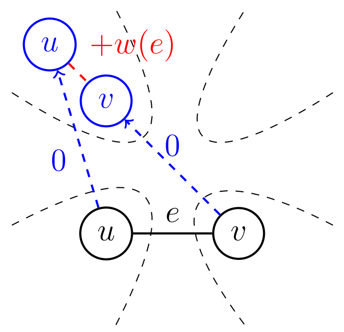
Positive Conflict. Both nodes move to the same block, i.e., $V_b \neq V_i = V_j \neq V_a$. Here, the local gain is again zero as e remains a cut edge in the local perspective of both moves. When we however execute them together, e is removed from the cut and the actual gain is $\omega(e)$. Hence, the value of this conflict is $\chi = \omega(e)$.

Single Negative Conflict. One node moves to the source block of the other node, but the latter moves into a complete different block, i.e., $V_b = V_i \neq V_j \neq V_a$. Here the individual gain for m_i^u is $\omega(e)$ as it removes e from the cut from its local perspective. However v moves to a different block itself, whereby e remains a cut edge. Hence, the actual gain is 0 and the value of this conflict is $\chi = -\omega(e)$.

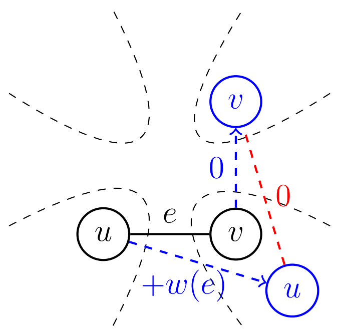
Double Negative Conflict. Both nodes swap their blocks, i.e., $V_b = V_i \neq V_j = V_a$. Here the gain is $\omega(e)$ for both moves individually. Therefore, the local gain is $g^L = 2\omega(e)$. However, after moving both nodes at once, e remains a cut edge and the actual gain is $g^A = 0$. This results in a value of $-2\omega(e)$ for this conflict.



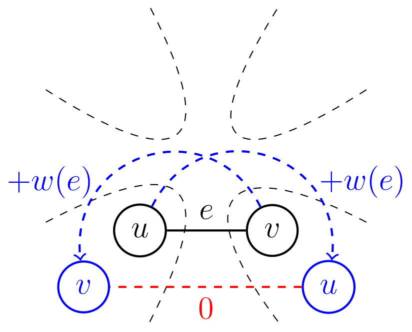
(a) *No conflict.* Both nodes move to two different blocks. $g^A = g^L = \chi = 0$.



(b) *Positive conflict.* Both nodes move to the same block. $g^A = \omega(e)$, $g^L = 0$, $\chi = \omega(e)$.



(c) *Single negative conflict.* The node u moves into the block of v while v moves to a different block. $g^A = 0$, $g^L = \omega(e)$, $\chi = -\omega(e)$.



(d) *Double negative conflict.* Both nodes swap their blocks. $g^A = 0$, $g^L = 2\omega(e)$, $\chi = -2\omega(e)$.

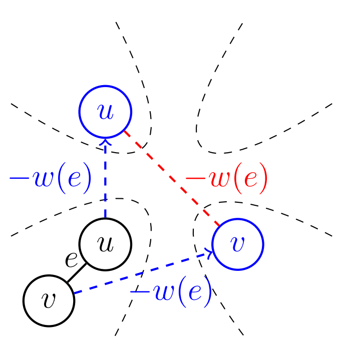
Figure 1: Visualization of possible move conflicts between adjacent nodes that are initially assigned to *different* blocks. The position of the nodes prior to and after moving them is drawn in black and blue, respectively. The individual gain of each move is drawn in blue, while the actual gain after moving both nodes is drawn in red.

These are the different types of conflicts for moves with different source blocks. Note that the case where both nodes move to either V_a or V_b does not exist as neither u nor v can move to its own source. We do not allow such moves as they are irrelevant. To conclude, the value of the conflict between m_i^u and m_j^v is $x \cdot \omega(e)$ for $x \in \{-2, -1, 0, 1\}$. The concrete value depends on the respective target blocks.

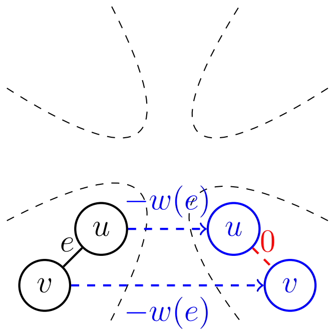
Let now u and v be both in V_a, i.e., the source blocks are *identical*. We can again observe conflicts between the moves m_i^u and m_j^v. In this case two different scenarios exist. Both scenarios result in a positive conflict. We visualize them in Figure 2 and explain them in the following.

Single Positive Conflict. Both nodes move to different blocks, i.e., $V_i \neq V_j$. From a local perspective, each move adds the edge e to the cut and has therefore a gain of $-\omega(e)$. Hence, the local gains is $g^L = -2\omega(e)$. However the actual gain is $g^A = -\omega(e)$ as the weight of the edge is only added once to the total cut. To this end, the value of this conflict is $\chi = \omega(e)$.

Double Positive Conflict. Both nodes move to the same block, i.e., $V_i = V_j$. Here the local gain is $g^L = -2\omega(e)$ again for the same reasons. The actual gain is however $g^A = 0$ as after moving both nodes, they are again in the same block and the edge is not added to the cut. This results in a value of $\chi = 2\omega(e)$ for this conflict.



(a) *Single Positive Conflict.* Both nodes move to two different blocks. $g^A = -\omega(e)$, $g^L = -2\omega(e)$, $\chi = \omega(e)$.



(b) *Double Positive Conflict.* Both nodes move to the same block. $g^A = 0$, $g^L = -2\omega(e)$, $\chi = 2\omega(e)$.

Figure 2: Visualization of possible move conflicts between adjacent nodes that are initially in the *same* block. The position of the nodes prior to and after moving them is drawn in black and blue, respectively. The individual gain of each move is drawn in blue, while the actual gain after moving both nodes is drawn in red.

These two cases represent all types of conflicts between moves with the same source block. To conclude, the value of the conflicts in this case are always $\chi = x \cdot \omega(e)$ where $x \in \{1, 2\}$, depending on the target blocks. Note that the conflicts are always positive.

4.3 Modeling Move Conflicts as Constraints

Recall that the ILP introduced in Section 4.1 is only accurate in the absence of any of the move conflicts introduced in Section 4.2. Now, we incorporate these conflicts into the ILP to accurately estimate the actual gain in all cases. Therefore, we add additional constraints and a conflict term to the objective function. We calculate the sum of the values of each conflict and add it to the objective function of the basic version. This results in the following ILP:

Definition 4.2 (Integer Linear Program for Conflicting Moves)**.** *Let $G = (V, E)$ be a graph with node weights c, edge weights ω and let $\Pi := \{V_1, \dots, V_k\}$ be a partition of G. Let M be any set of moves of nodes in G. The ILP for conflicting moves is defined as*

$$\text{maximize} \qquad \sum_{m_i^u \in M} g_i(u) \cdot x_i^u + \sum_{(u,v) \in E} \omega(u,v) \cdot \chi_{u,v} \qquad \text{s.t.}$$

$$\forall u \in V: \qquad \sum_{m_i^u \in M} x_i^u \leq 1 \tag{4.3}$$

$$\forall 1 \leq i \leq k: \qquad W_{in}^i - W_{out}^i + c(V_i) \leq W_{max}^i \tag{4.4}$$

and $\forall (u,v) \in E$ *with* $V_a := \Pi[u] \neq \Pi[v] =: V_b$:

$$\chi_{u,v} + 2.5\,(x_b^u + x_a^v) + \sum_{i \neq b} x_i^u + \sum_{i \neq a} x_i^v \leq 3 \tag{4.5}$$

$$(k+1) \cdot \chi_{u,v} + \sum_{1 \leq i \leq k} (i \cdot x_i^u - (i+1) \cdot x_i^v) \leq k \tag{4.6}$$

$$(k+1) \cdot \chi_{u,v} + \sum_{1 \leq i \leq k} (i \cdot x_i^v - (i+1) \cdot x_i^u) \leq k \tag{4.7}$$

and $\forall (u,v) \in E$ *with* $\Pi[u] = \Pi[v]$:

$$(k+1) \cdot \chi_{u,v} + \sum_{1 \le i \le k} \Big(i \cdot x_i^u - (i+k+2) \cdot x_i^v \Big) \le k \tag{4.8}$$

$$(k+1) \cdot \chi_{u,v} + \sum_{1 \le i \le k} \Big(i \cdot x_i^v - (i+k+2) \cdot x_i^u \Big) \le k \tag{4.9}$$

where each x_i^u *is again a binary decision variables. Further* $\chi_{u,v}$ *is a bounded integer variable for each* $(u,v) \in E$ *that represents the type of the conflict between* u *and* v*. For* $\Pi[u] \neq \Pi[v]$ *the limits are* $-2 \le \chi_{u,v} \le 1$ *and* $0 \le \chi_{u,v} \le 2$ *otherwise. The maximum, incoming and outgoing weights* W^i_{max}, W^i_{in} *and* W^i_{out}, *respectively, are defined as in the basic version in Definition 4.1.*

As can be seen, this ILP contains all constraints from the basic version (see constraints 4.3 and 4.4), uses the same move variables x_i^u for each move $m_i^u \in M$ and has the same sum as first part of the objective function. The main additions are the *conflict variables* $\chi_{u,v}$. The ILP introduces one of these variables for each pair of conflicting nodes. The main idea of these variables is to model the type of conflict. Consider for example two adjacent nodes u and v in different blocks. Let further these nodes switch their blocks according to the move variables. In this case the conflict variable has to be $\chi_{u,v} = -2$. Recall therefore from Section 4.2 that this is a double negative conflict. Analogously, the variable is expected to be -1, 1, 2 or 0 if the corresponding nodes create a single negative, single positive, double positive or no conflict, respectively. Note that for readability reasons we defined conflict variables and corresponding constraints for every adjacent node pair $(u,v) \in E$. This is not necessary in practice, as we only have to consider adjacent node pairs for which both nodes have at least one move in M. Otherwise, no conflict can arise.

The objective function consists of two sums. The first sum is equivalent to the objective function of the basic version, i.e., the local gain $g^L(M_X)$ for any solution X. The second sum adds the total conflict value due to the expected values of each $\chi_{u,v}$. The sum of both equals the actual gain $g^A(M_X)$, as the conflict value is defined to be the difference between actual and local gain. Hence the ILP maximizes the actual gain instead of just the local gain.

The constraints 4.5 - 4.7 enforce that the conflict variable $\chi_{u,v}$ has the correct value if $\Pi[u] \neq \Pi[v]$, i.e., u and v are initially in different blocks. The correct values for $\Pi[u] = \Pi[v]$ are enforced by the constraints 4.8 and 4.9. Note that the types of conflict differ if the nodes are initially in different or the same block, hence we use different constraints for both cases. We explain the constraints in the following. Note therefore, that the move variables x_i^u are binary decision variables and that constraint 4.3 guarantees that for each node at most one move variable is 1. Note further that we maximize the conflict variables in the second sum of the objective function. Hence, an upper bound $\chi_{u,v} \le C$ is sufficient to force the conflict variable to be equal to C.

4.3.1 Handling Conflicts between Nodes in Different Blocks

We first explain the constraints 4.5 - 4.7 for conflicts between nodes in different blocks. Therefore, we derive the constraints step by step. We first give an intuition to the general structure of them. Then we parameterize the numerical values in the constraints. We can derive requirements for these parameter from the requirements for the conflict variables for the different types of conflicts. Finally, we show that the values we used in the ILP definition respect all requirements.

Let therefore $u, v \in V$ be adjacent nodes where $V_a := \Pi[u] \neq \Pi[v] =: V_b$ in the original partition. We need to add constraints that set an upper bound to the conflict variable for the four different cases: double negative conflict, single negative conflict, no conflict and positive conflict. We first derive the necessary constraint for the cases of a negative conflict and later the constraints for handling no or a positive conflict.

Handling Negative Conflicts. Recall that a double negative conflict arises if both nodes swap their blocks. This means $x_b^u = x_a^v = 1$ and we need the constraint $\chi_{u,v} \leq -2$ to represent the conflict correctly. A first idea would be the constraint

$$\chi_{u,v} + x_b^u + x_a^v \leq 0 \; . \tag{4.10}$$

It simplifies to the required inequality when a double negative conflict arises.

We now consider the case of a single negative conflict. Recall that in this case one node moves into the source block of the other node. Let the first node be u and the latter v. In addition, the node v has to move to a block different than V_a. This results in $x_b^u = 1$ and $x_j^v = 1$ for $j \neq a$. When we insert these values into Equation 4.10, we can simplify the constraint to $\chi_{u,v} \leq -1$. This is the expected behavior. So the constraint above correctly handles the cases of negative conflicts.

When we however now consider the cases where the moves create no conflict, we will see that this constraint wrongfully restricts the conflict variable. Therefore let $x_b^u = 1$ and $\forall j : x_j^v = 0$, i.e., only one node moves to the source block of the other while the other node does not move. This does not create any conflict value, Equation 4.10 however simplifies to $\chi_{u,v} \leq -1$. Hence we need to factor every move variable into our constraint to guarantee that the conflict variable is only restricted if both nodes move. We however still have to differentiate between moving a node into the source block of the other node and moving it into a completely different block. This motivates the constraint

$$\chi_{u,v} + c_1 \left(x_b^u + x_a^v \right) + \sum_{i \neq b} x_i^u + \sum_{i \neq a} x_i^v \leq c_2 \tag{4.11}$$

for proper values of the constants c_1 and c_2. To find these values, we evaluate this constraint for every conflict type. This results in inequalities that restrict the conflict variable depending on c_1 and c_2. With the expected upper bounds for $\chi_{u,v}$ in each case, we can derive requirements for c_1 and c_2. From these requirements for the different cases, we can derive proper values for these constants. We start with the cases of negative conflicts. Recall therefore that for a single or double negative conflict the equation must simplify to $\chi_{u,v} \leq -1$ or $\chi_{u,v} \leq -2$ respectively.

Single Negative Conflict. Let u be the node that moves into the source block of v. For a single negative conflict we have the situation $x_b^u = 1$ and $x_j^v = 1$ for a $j \neq a$. Inserting this into Equation 4.11 results in $\chi_{u,v} \leq c_2 - 1 - c_1$. To achieve the expected upper bound of -1, the inequality $-1 \leq c_2 - 1 - c_1 < 0$ must be satisfied. The range of $[-1, 0)$ is allowed as upper bound, as the conflict value is an integer. Simplifying this inequality results in the requirement

$$c_1 \leq c_2 < c_1 + 1 \tag{4.12}$$

for the constants c_1 and c_2.

Double Negative Conflict. For a double negative conflict, both nodes swap their blocks, i.e., $x_b^u = 1$ and $x_a^v = 1$. With these values, we can simplify the constraint 4.11 to $\chi_{u,v} \leq c_2 - 2c_1$. In this case, the conflict variable has to equal to -2, hence the right side of the simplified constraint has to be in $[-2, -1)$. We therefore obtain the requirement

$$2c_1 - 2 \leq c_2 < 2c_1 - 1 \tag{4.13}$$

for c_1 and c_2.

Combining these two inequalities, we obtain $2c_1 - 2 < c_1 + 1$ and thus $c_1 < 3$. Substituting this result into the right inequality of 4.12, we get $c_2 < c_1 + 1 < 4$. We now calculate lower bounds of these constants by evaluating the cases where the move variables are set such that no conflict or a positive conflict arises. In the former case, we expect the constraint to be equivalent to $\chi_{u,v} \leq C$ with a $C \geq 0$. While in the latter, we expect it to be $\chi_{u,v} \leq C$ for a $C \geq 1$. Note that we do not force the direct upper bound of 0 or 1 respectively. This is because we later introduce two additional constraints that properly set the upper bounds for no and a positive conflicts. By using this loose upper bound here, we guarantee to not interfere with restrictions from these constraints. We now present every possible move variable assignment that creates no or a positive conflict.

Both Move without Negative Conflict. If u and v move into blocks $V_i \neq V_b$ and $V_j \neq V_a$, respectively, both nodes move, but no negative conflict arises. To translate this case to the assignment of the move variables, we have $x_i^u = 1$ and $x_j^v = 1$ for $i \neq b$ and $j \neq a$. If $V_i = V_j$, we observe a positive conflict. Otherwise, there is no conflict between the moves. However we can not distinguish between both cases with the here discussed constraint 4.11. Therefore we later introduce two other constraints. To not interfere with them, we expect that the constraint here is in both cases equivalent to $\chi_{u,v} \leq c$ with $c \geq 1$. If we now substitute the assignment of the move variables into our constraint 4.11, we get $\chi_{u,v} \leq c_2 - 2$. With the expected lower bound of $c \geq 1$ for the right side of this inequality, we derive the lower bound

$$3 \leq c_2 \,. \tag{4.14}$$

One Node is Moving. We now explain the case when exactly one node moves. Let u be this node without loss of generality. This means $x_i^u = 1$ for exactly one i and $\forall j : x_j^v = 0$. We can simplify our constraint 4.11 to either $\chi_{u,v} \leq c_2 - c_1$ or $\chi_{u,v} \leq c_2 - 1$, depending whether the target of the move is the source block of v or not. As both cases create no conflict because only one node moves, we expect the right sides of each inequality to be greater or equal to 0. Hence, we can derive $c_2 - c_1 \geq 0$ from the first case. This is however equivalent to $c_1 \leq c_2$ which is identical to the left inequality in 4.12. From the second case we can derive $c_2 \geq 1$, which is less restrictive then 4.14. Therefore, we do not derive further restrictions for c_1 or c_2 in this case.

No Node is Moving. Consider the case where each move variable for u and v is zero. Here Equation 4.11 evaluates to $\chi_{u,v} \leq c_2$. From the requirements to the upper bound, we can derive $c_2 \geq 0$ which again is less restrictive then 4.14.

In total we get $3 \leq c_2$ in order to properly handle the cases of no or a positive conflict. Substituting this result into the right inequality of 4.12, we get $2 < c_1$. In total we result in the ranges

$$2 < c_1 < 3 \tag{4.15}$$
$$3 \leq c_2 < 4 \tag{4.16}$$

for c_1 and c_2. We can now select one value for either c_1 or c_2 in these ranges and find atleast one value for the other constant while maintaining the requirements from above. We select $c_2 := 3$. Substituting this into the left inequality in 4.13, we get $2c_1 - 2 \leq 3$ which is equivalent to $c_1 \leq 2.5$. The other requirements derived from negative conflicts (4.12 and 4.13) do not further restrict c_1. Hence we select $c_1 := 2.5$. With this we get the final constraint for handling negative conflicts and we can conclude:

Corollary 4.2 (Constraint for Negative Conflicts Between Nodes in Different Blocks)**.** *For each adjacent nodes $u, v \in V$ with $\Pi[u] \neq \Pi[v]$ add the constraint*

$$\chi_{u,v} + 2.5\,(x_b^u + x_a^v) + \sum_{i \neq b} x_i^u + \sum_{i \neq a} x_i^v \leq 3 \qquad \text{(negative conflicts)}$$

to properly restrict the conflict variable $\chi_{u,v}$ when single or double negative conflicting moves are selected.

By substituting $c_1 = 2.5$ and $c_2 = 3$ in the case study above, we can see, that this constraint leads to the desired upper bounds for each conflict type. For single or double negative conflicts it is equivalent to $\chi_{u,v} \leq -1$ or $\chi_{u,v} \leq -2$ respectively. If the selected moves create no conflict, the upper bound on the conflict variable is greater than or equal to 0. Further if a positive conflict arises, it is greater or equal to 1. We now introduce the constraints to distinguish between a positive conflict and no conflict.

Handling Positive Conflicts. We obtain a positive conflict if both nodes move into the same block, i.e., $x_i^u = x_i^v = 1$ for any $a \neq i \neq b$. In this case, we want the constraint $\chi_{u,v} \leq 1$ as the value of this conflict is $1 \cdot \omega(u, v)$. In any other case, we need the constraint $\chi_{u,v} \leq 0$ as this is either no conflict or a negative conflict. We use 0 as upper bound for both case to not interfere with the constraint above, which distinguishes between single negative, double negative and no conflict. This means we need different upper bounds when moves into the same block are selected. Therefore we use the sum

$$S := \sum_{1 \leq i \leq k} (i \cdot x_i^u - (i+1) \cdot x_i^v)$$

which equals -1 if and only if moves into the same block are selected, i.e., a positive conflict arises. In all other cases the sum is either less or greater than -1 Note that we use $(i+1)$ as the factor of x_i^v instead of i. This is necessary to distinguish between the case of a positive conflict and the case where both nodes do not move at all, where the sum always equals 0. We can now use the properties of this sum to force the desired upper bound of 1 for positive conflicts and 0 in all other cases. The constraint $\chi_{u,v} \leq f(S)$ for a function f, where

$$1 \leq f(-1) < 2 \text{ and}$$
$$\forall x \neq -1 : 0 \leq f(x) < 1$$

would have the expected behavior. Such a function exists, but it can not be linear, as it has a global maximum at -1. Recall therefore that no linear function has a global maximum at one specific point. We can however use a trick similar to modeling the absolute value of a variable. The constraint $|x| < 1$ is equivalent to the two linear constraints $x < 1$ and $-x < 1$. This motivates using the two constraints

$$c_1 \cdot \chi_{u,v} + \sum_{1 \leq i \leq k} (i \cdot x_i^u - (i+1) \cdot x_i^v) \leq c_2 \tag{4.17}$$

$$c_1 \cdot \chi_{u,v} + \sum_{1 \leq i \leq k} (i \cdot x_i^v - (i+1) \cdot x_i^u) \leq c_2 \tag{4.18}$$

to handle positive conflicts. Note that these two constraints only differ in the order of x_i^u and x_i^v inside the sum. We now derive proper values for c_1 and c_2 by evaluating every possible move variable assignment, similar to above. Here our expected behavior is as follows: If moves that are positive conflicting are selected, both constraints have to be equivalent to $\chi_{u,v} \leq 1$. In all other cases, we expect *at least one* of the constraints to be equivalent to $\chi_{u,v} \leq 0$. The other constraint can be equivalent to $\chi_{u,v} \leq C$ for $C \geq 0$. We now evaluate every possible move variable assignment and derive requirements for c_1 and c_2 to achieve the expected behavior.

Positive Conflict. Let $x_i^u = x_i^v = 1$ for any $a \neq i \neq b$. Recall that the case where i equals a or b does not exist, as then either u or v would move into its own source block, which we do not allow. We can simplify both constraints to $c_1 \cdot \chi_{u,v} \leq c_2 + 1$. The constants c_1 and c_2 must be chosen so that this is equivalent to $\chi_{u,v} \leq 1$, as this case is a positive conflict. Therefore, we obtain the requirement

$$1 \leq \frac{c_2 + 1}{c_1} < 2 \;\Leftrightarrow\; c_1 - 1 \leq c_2 < 2c_1 - 1 \tag{4.19}$$

for c_1 and c_2. Recall that $\chi_{u,v}$ is an integer and therefore the upper bound can be any value between 1 and 2 (excluding 2).

Both Move into Different Blocks. Let $x_i^u = 1 = x_j^v$ for any $i \neq j$. We expect in this case that at least one of the constraints 4.17 or 4.18 simplifies to $\chi_{u,v} \leq 0$. Without loss of generality let $i > j$. In this case, we expect that the first constraint simplifies to the expected upper bound of 0 and the second is equal or less restrictive. Substituting the assumed move variable assignment into 4.17, we get $c_1 \cdot \chi_{u,v} \leq c_2 - i + j + 1$. From this we can derive the requirement

$$0 \leq \frac{c_2 - i + j + 1}{c_1} < 1 \;\Leftrightarrow\; i - j - 1 \leq c_2 < c_1 + i - j - 1$$

where $0 \leq i - j - 1 \leq k - 2$ because $i > j$ and $1 \leq i, j \leq k$. This is the most restrictive limit for $I := i - j - 1$ as $I = 0$ for $i = j + 1$ and $I = k - 2$ for $i = k$ and $j = 1$. When we insert these limits into the inequality above, we obtain the lower and upper bound

$$k - 2 \leq c_2 < c_1 \tag{4.20}$$

for c_2. Note that this implies $0 \leq c_2$ and $0 < c_1$ because $k \geq 2$. The second constraint, 4.18, simplifies to $c_1 \cdot \chi_{u,v} \leq c_2 - j + i + 1$. Here we expect the upper bound to be greater or equal to 0. Because $c_2 \geq 0$ and we assumed $j < i$, we can conclude that $c_2 - j + i + 1 \geq 0$. Further, we already know that $c_1 > 0$. Thus this constraint is, when the already found restrictions for c_1 and c_2 hold, always equivalent to $\chi_{u,v} \leq C$ for $C \geq 0$. This is the expected behavior and we therefore do not have to restrict c_1 and c_2 further.

One Node is Moving. Consider the case where exactly one node moves. Note that the results are different, depending on which node is moving. As this creates no conflict, we again expect that at least one of the constraints is equivalent to $\chi_{u,v} \leq 0$. Without loss of generality let u be the node that is moving, i.e., $x_i^u = 1$ for exactly one i and $\forall m_j^v \in M : \; x_j^v = 0$. The first constraint, 4.17, simplifies to $c_1 \cdot \chi_{u,v} \leq c_2 - i$. To achieve the expected upper bound for the conflict variable, we derive the requirement

$$0 \leq \frac{c_2 - i}{c_1} < 1 \;\Leftrightarrow\; i \leq c_2 < c_1 + i$$

for c_1 and c_2. As i can be any number between 1 and k, we can derive the limits $k \leq c_2 < c_1 + 1$. However the upper bound is here less restrictive than in the requirement 4.20. Hence, we can ignore it. To conclude, we obtain from the case where only u moves the requirement

$$k \leq c_2 \tag{4.21}$$

for c_2. We now have to check that the second constraint for handling positive conflicts, 4.18, is not more restrictive then the first. Substituting the values for the move variables, we get $c_1 \cdot \chi_{u,v} - (i + 1) \leq c_2$. This translates to the requirement

$$0 \leq \frac{c_2 + i + 1}{c_1}$$

for c_1 and c_2. Requirement 4.20 already demands, that $c_1 > 0$ and $c_2 \geq 0$. Further is i greater than zero. Hence, the requirement above is always true under the already derived requirements and we do not have to further restrict c_1 or c_2.

No Node is Moving. Let $x_i^u = x_i^v = 0$ for every i, i.e., both nodes do not move. Both constraints 4.17 and 4.18 evaluate to $c_1 \cdot \chi_{u,v} \leq c_2$. As this case creates no positive conflict, the expected constraint is $\chi_{u,v} \leq 0$. This creates the requirement

$$0 \leq \frac{c_2}{c_1} < 1 \;\Leftrightarrow\; 0 \leq c_2 < c_1$$

for the constants c_1 and c_2. Because we already derived the tighter limits $k \leq c_2 < c_1$ (see 4.20 and 4.21), this does not further restrict c_1 and c_2.

To conclude, we obtain the two requirements $c_1 - 1 \leq c_2 < 2c_1 - 1$ (4.19) and $k \leq c_2 < c_1$ (4.20 and 4.21). If we now pick any values for c_1 and c_2 that satisfy these requirements, we have shown that the constraints 4.17 and 4.18 together correctly set the limits for the conflict variable to handle positive conflicts. We have also shown that then these constraints do not conflict with the constraint for handling negative conflicts. To find correct values, we pick $c_2 := k$. We can then derive that $c_1 - 1 \leq c_2 = k$. To also satisfy $c_2 = k < c_1$, we pick $c_1 := k+1$. The last requirement $c_2 = k < 2(k+1) - 1 = 2c_1 - 1$ is also satisfied with this value for c_1. Hence, we found two values that satisfy all requirements and we conclude the following constraints to correctly handle positive conflicts:

Corollary 4.3 (Constraints for Positive Conflicts Between Nodes in Different Blocks)**.** *For each adjacent nodes $u, v \in V$ with $\Pi[u] \neq \Pi[v]$ add the constraints*

$$(k+1) \cdot \chi_{u,v} + \sum_{1 \leq i \leq k} (i \cdot x_i^u - (i+1) \cdot x_i^v) \leq k \qquad \text{(positive conflict I)}$$

$$(k+1) \cdot \chi_{u,v} + \sum_{1 \leq i \leq k} (i \cdot x_i^v - (i+1) \cdot x_i^u) \leq k \qquad \text{(positive conflict II)}$$

to properly restrict the conflict variable $\chi_{u,v}$ when positive conflicting moves are selected.

These two constraints together with Theorem 4.2 are the three constraints 4.5 - 4.7 in the complete definition of our ILP in Definition 4.2. These are added for every adjacent node pair in different blocks. We have shown above, that with these three constraints, the conflict variable $\chi_{u,v}$ is correctly restricted for each possible assignment of the move variables. This means, that we correctly calculate the conflict value in the objective function of our ILP, if no conflicting nodes share the same block originally. Now we derive and show the correctness of the constraints for conflicts between nodes in the same block.

4.3.2 Handling Conflicts between Nodes in the Same Block

Let $u, v \in V$ be adjacent nodes where $V_a := \Pi[u] = \Pi[v]$ in the original partition Π. Recall that if both of these nodes move, we always have a positive conflict. It can be either single positive, if they move into different blocks, or double positive, if they move into the same block. If only one or none of them move, no conflict arises. Analogously to handling positive conflicts for nodes in different blocks, we have to distinguish between moving in the same or different blocks. This motivates using similar constraints. However we also have to distinguish between moving in different blocks (positive conflict) and moving at most one node (no conflict). Hence, we use the slight modification

$$(k+1)\cdot\chi_{u,v} + \sum_{1\le i\le k} (i\cdot x_i^u - (i+c_3)\cdot x_i^v) \le k \tag{4.22}$$

$$(k+1)\cdot\chi_{u,v} + \sum_{1\le i\le k} (i\cdot x_i^v - (i+c_3)\cdot x_i^u) \le k \tag{4.23}$$

where we modified the sum, by replacing $(i+1)$ with $(i+c_3)$. Here c_3 is a properly picked constant. We can use this additional constant to distinguish between moving both nodes and at most one of them. We now evaluate every possible cases for assigning the move variables. We derive requirements for c_3 to properly restrict $\chi_{u,v}$ by 0, 1 or 2, depending on the resulting conflict.

Double Positive Conflict. Let $x_i^u = x_i^v = 1$, i.e., u and v move into the same block. This is a double positive conflict, hence $\chi_{u,v} \le 2$ is expected. If we substitute the assumed values for the move variables, both constraints simplify to $(k+1)\cdot\chi_{u,v} \le k + c_3$. To achieve the expected upper bound, we derive the requirements

$$2 \le \frac{k+c_3}{k+1} < 3 \;\Leftrightarrow\; 2(k+1) - k \le c_3 < 3(k+1) - k$$

for c_3. We can simplify the lower and upper bound for c_3. This results in the limits

$$k+2 \le c_3 < 2k+3\ . \tag{4.24}$$

Single Positive Conflict. Let $x_i^u = 1 = x_j^v$ for $i \ne j$, i.e., u and v move into different blocks. The results of the individual constraints differ, depending on whether $i > j$ or $i < j$. In both cases we expect that at least one of the constraints implies $\chi_{u,v} \le 1$, while the other only has to be less or equal restrictive. This is because the assumed assignment of the move variables implies a single positive conflict.

Without loss of generality, we assume that $i > j$. The first constraint (4.22) simplifies to $(k+1)\cdot\chi_{u,v} \le k - i + j + c_3$. We expect that c_3 is picked such that this constraint implies the upper bound of zero for the conflict variable. Hence, we obtain the requirement

$$1 \le \frac{k-i+j+c_3}{k+1} < 2 \;\Leftrightarrow\; 1+i-j \le c_3 < k+2+i-j$$

which we can further simplify. Because $i \le k$ and $j \ge 1$, the lower bound for c_3 above, i.e., $1+i-j \le c_3$, implies $1+k-1 = k \le c_3$. This is however less restrictive then the lower bound of $k+2$ derived from the case of a double positive conflict. We can therefore ignore it. We can however derive from the upper bound $c_3 < k+2+i-j$ the requirement

$$c_3 < k+3 \tag{4.25}$$

as $i-j \ge 1$ because we assume $i > j$. The second constraint (4.23) simplifies to $(k+1)\cdot\chi_{u,v} \le k - j + i + c_3$. Here we do not expect that this implies $\chi_{u,v} \le 1$, as the first constraint already enforces this upper bound for properly selected c_3. We only expect, that the second constraint is not more restrictive. Hence, we demand

$$1 \le \frac{k-j+i+c_3}{k+1} \;\Leftrightarrow\; k+1-k+j-i \le c_3$$

which implies $j-i+1 \le c_3$. This is however less restrictive for c_3 than the lower bound of $k+2$ (see 4.24) because $j-i<0$ and $k+2 \ge 2$. Hence, we do not obtain further restriction for c_3 in this case.

One Node is Moving. Let $x_i^u = 1$ for exactly one i and $\forall 1 \leq j \leq k : x_j^v = 0$, i.e., only u is moving. Note that the case when only v is moving is analogous. The first constraint evaluates to $(k+1) \cdot \chi_{u,v} \leq k - i$. Because of $1 \leq i \leq k$, we conclude $0 \leq \frac{k-i}{k+1} < 1$. This shows that the first constraint 4.22 correctly implies the upper bound of 0 for the conflict variable. To evaluate the second constraint 4.23, we first substitute the move variables for this case and we get $(k+1) \cdot \chi_{u,v} \leq k + i + c_3$. We know that $i \geq 1$ and we already derived the requirement $c_3 \geq k+2$ (see 4.24). This implies that this constraint is not more restrictive than the first constraint, as $k + i + c_3 > k + 1$. In total, we have shown that our constraints correctly handle the case where only one node moves when the already found requirements for c_3 are satisfied.

No Node is Moving. Let $x_i^u = x_i^v = 0$ for all i, i.e., no node is moving. Here both constraints simplify to $(k+1) \cdot \chi_{u,v} \leq k$. This is equivalent to $\chi_{u,v} \leq 0$, as $0 \leq k < k+1$. This is the expected behavior. We have therefore shown, that our constraints handle this case correctly for every value of c_3.

To conclude, we have two requirements for c_3. First we derived $k + 2 \leq c_3 < 2k + 3$ from the case of a double positive conflict (see 4.24). From a single positive conflict we further derived $c_3 < k + 3$ in 4.25. In total we can select $c_3 := k + 2$ while satisfying all requirements. Thus, we conclude the following constraints that have to be added for every conflicting node pair in the same block:

Corollary 4.4 (Constraints for Conflicts Between Nodes in the Same Block)**.** *For each adjacent nodes $u, v \in V$ with $\Pi[u] = \Pi[v]$ add the constraints*

$$(k+1) \cdot \chi_{u,v} + \sum_{1 \leq i \leq k} \left(i \cdot x_i^u - (i + k + 2) \cdot x_i^v \right) \leq k$$

$$(k+1) \cdot \chi_{u,v} + \sum_{1 \leq i \leq k} \left(i \cdot x_i^v - (i + k + 2) \cdot x_i^u \right) \leq k$$

to properly restrict the conflict variable $\chi_{u,v}$ when conflicting moves are selected.

The correctness of these constraints follows directly from the case study above. This means after deriving the constraints to handle conflicts between different blocks, we have now derived the constraints to properly restrict the conflict variable between nodes in the same block. In total we have explained every constraint from Definition 4.2, the full description of our ILP.

4.4 Move Selection

We now present two different strategies for the move selection phase in Line 2 of Algorithm 1. The goal of this phase is to find promising moves that could reduce the cut when executed. We then use the above described ILP to find the optimal subset of the selected candidates, such that the total cut is minimized after executing them. Note that we need to reduce the search space for the ILP, as solving it for all possible move combinations is not feasible. We also encourage selecting moves, that do not achieve an optimal gain from a local perspective. The motivation behind this is to escape local minima by e.g. moving nodes out of an overloaded block to allow moving other nodes into this block. This is especially essential as we expect to be already close to a local minimum.

The output of the move selection phase is a set of sets $\mathcal{M}$. Each $M \in \mathcal{M}$ is hereby a set of moves from which we construct a single individual ILP. This means that we construct and solve $|\mathcal{M}|$ ILPs after each move selection phase. This enables us to solve several smaller ILPs for different regions of the graph, instead of one large ILP. It further allows us to solve each of the move buffers in parallel. This approach can however lead to conflicts, if we allow arbitrary moves in each M. If for example one node u has moves in several move sets, two ILPs could move this node twice. This can then result in a faulty gain calculation, if both ILPs expect that u is in its original block before moving. When we solve the move sets sequentially, we can avoid these conflicts by recalculating the local gain for each move. This is however not possible if we want to solve the move sets in $\mathcal{M}$ in parallel. Hence we expect that each pair $(M_1, M_2) \in \mathcal{M} \times \mathcal{M}$ fulfills the following two requirements:

(i) No node $u \in V$ has moves in M_1 and M_2.

(ii) No two moves $m_1 \in M_1$ and $m_2 \in M_2$ exist, that are conflicting.

The first requirements avoids moving nodes twice. For the second requirement recall Section 4.2. We have shown there, that the actual gain for moving two adjacent nodes can differ from the sum of each individual gain. If now two different ILPs move two adjacent nodes, the actual result can differ from the expected total cut, because each ILP does not necessarily know that the other node will be moved. Hence we only allow conflicting moves in one move set.

We now present two different strategies for selecting moves. First a simple fully randomized selection strategy, the *Random Move Selection*. Then we introduce our move selection algorithm, the *Promising Cluster Move Selection*, that is based on the idea to cluster the quotient graph. We then select moves only between blocks of each cluster.

Random Move Selection. The most simple move selection strategy is the Random move selection. For the final result $|\mathcal{M}| = 1$ always holds, i.e., we only populate one move buffer M. This strategy selects the moves completely random. This means picks a random node u and a random block V_i and adds the move m_i^u to the move buffer M. This is repeated up to a maximum number of moves. Because we only have one move buffer, we always respect the necessary requirements to avoid conflicts between different ILPs.

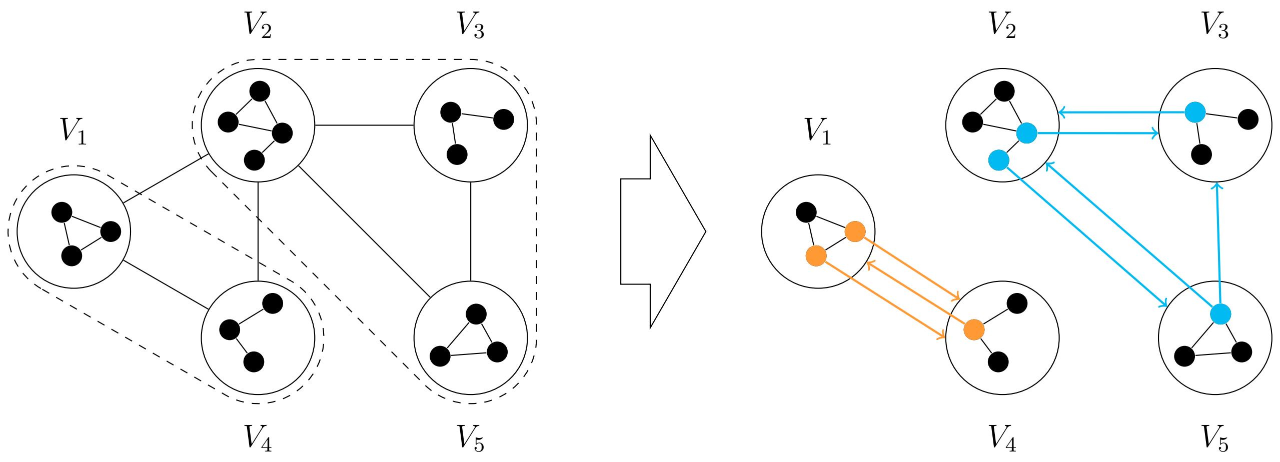


Figure 3: Visualization of the PromisingCluster move selection strategy. The left side shows the clustered quotient graph where the dashed outlines mark a cluster. The right side shows the selected moves through coloring. The moves with the same color belong to the same move buffer.

Promising Cluster Move Selection. We now present the PromisingCluster move selection (PC) strategy. The general idea of this strategy is to find regions where we expect potential to improve the total cut. We then select moves only within the respective regions. The general structure of this strategy can be seen in Algorithm 2 and is visualized in Figure 3. We start with clustering the quotient graph $\mathcal{Q}$ (see Line 1 and the left side of Figure 3). Recall that the quotient graph is an abstract view on the partitioned graph, where each block is represented by a single node. We restrict the size of each cluster with a tuning parameter c_{max}. By clustering the quotient graph, we find groups of blocks that have heavy connections between each other. We expect that mainly moves within each cluster reduce the total cut. This is because if for example two blocks V_i and V_j are not connected in $\mathcal{Q}$, nothing can be gained by moving nodes between them, as no edge can be removed from the cut. Note however, that in some cases it could be useful to move a node from V_i to V_j, if for example V_i is overloaded and we can then move another node to V_i to reduce the total cut. But we expect that in general regions with many connected blocks provide the best improvement potential. Hence, we then find moves only within each cluster (see Line 4). For each cluster we populate a single move buffer M. Therefore we result in the total set of move buffers $\mathcal{M}$, where $|\mathcal{M}| = |\mathcal{C}|$.

Algorithm 2: PromisingCluster Move Selection

```
Data: $G = (V, E)$, initial partition $\Pi := \{V_1, \dots, V_k\}$, quotient graph $\mathcal{Q}(\Pi, E_\Pi)$,
      maximum cluster size $c_{max}$
1 $\mathcal{C}$ := cluster($\mathcal{Q}$, $c_{max}$)
2 $\mathcal{M} := \emptyset$
3 foreach $C \in \mathcal{C}$ do
4     $M$ := find_moves_in_region($G$, $\Pi$, $C$)
5     $\mathcal{M} := \mathcal{M} \cup \{M\}$
6 return $\mathcal{M}$
```

Before introducing our strategy to find moves within a single cluster, we now show that our constructed set of move buffers $\mathcal{M}$ fulfills the two requirements mentioned above. Recall therefore that the clusters in $\mathcal{C}$ are pairwise disjoint. Firstly, each node $u \in V$ has moves in only at most one move buffer, because its current block $\Pi[u]$ is part of exactly one cluster $C \in \mathcal{C}$. Secondly, no two moves in different move buffers are conflicting. Let therefore $M_1, M_2 \in \mathcal{M}$ be any two move buffers. Every two moves $m_i^u \in M_1$ and $m_j^v \in M_2$ have different source and target blocks. This is because we demanded that the nodes move only within the respective cluster for M_1 and M_2. In subsection 4.2 we have shown, that in this case no conflict between m_i^u and m_j^v arises. Hence, no two conflicting moves in different move buffers exist.

To finalize the introduction of this move selection strategy, we describe how we find moves within a given cluster, as in Line 4 of Algorithm 2. Let therefore $C \in \mathcal{C}$ be our currently processed cluster. The general idea is to add some of the moves with the highest gain of each block. Algorithm 3 shows the general structure of the selection strategy. We iteratively select a block $V_{cur} \in C$ and then find the move $m_i^u \notin M$ from V_{cur} to any other block in C with maximum gain (see Line 4). We then add m_i^u to M and select the next block from which we pick the best move. This is repeated until we selected a maximum number of moves. We use $|\cdot|_{ilp}$ to calculate the *ILP-size* of a move buffer M. This is the number of non-zeroes of the ILP constructed from M. We stop adding moves if the constructed ILP has more then z_{max} non-zeroes.

V_{cur} is originally the block with the move with the highest gain in C (see Line 2). In Line 6 we then select the next block with two different methods. The first method is called *along* *path.* Here if we added the move m_i^u to M, we select $V_{cur} := V_i$, i.e., the block to which the move moves the node. This is motivated by the balance constraint. If we for example only add moves into V_i, the ILP can not select all of these moves, because V_i would exceed its size limit. Hence we offer the ILP a move out of V_i to possibly counter balance the additional weight of u. But this could result in only picking moves from two alternating blocks. Therefore we have a second method called *equally distributed.* Here we choose equally distributed a block from C as next block. We combine these two methods by using the first until M reaches a certain size, and then switch to the second method.

Algorithm 3: Finding Moves in a Cluster of Blocks

```
  Data: $G = (V, E)$, current partition $\Pi$, cluster of blocks $C = \{V_{i_1}, \ldots, V_{i_c}\}$
1 $M := \emptyset$
2 $V_{cur}$ := block_highest_gain($C$)              // get block with highest gain move in C
3 while $|M|_{ilp} < z_{max}$ do
4     $m_i^u$ := max_move($C$, $V_{cur}$, $M$)   // find move with maximum gain from $V_{cur}$ to $V_i \in C$
5     $M := M \cup \{m_i^u\}$                                  // add this move to M
6     $V_{cur}$ := next_block($V_{cur}$, $C$)                  // find next block
```

5 Other Refinement Algorithms

We now shortly present our implementation of the other refinement algorithms that we will evaluate in Section 6. We used KaMinPar [20] as underlying framework where we implemented each of the algorithms.

Fiduccia-Mattheyses Algorithm. We use the FM-algorithm in the version of Sanders and Schulz [39], i.e., *multi-try k-way FM local search*. The so-called *multi-try k-way FM-algorithm* works as follows. We first iterate over the nodes to find the set of boundary nodes B. Then we pick a random node $u \in B$ that has not been touched earlier in this round. From u we start a local search. Therefore we add u into a priority queue P, where the priority is the maximum gain when moving into any block. We then repeat the following steps until P is empty or a stopping criterion is reached. Note that we use the adaptive stopping criterion introduced by Osipov and Sanders [36].

Let v be the node in P with the highest priority. We remove v from P and try to move it. Let V_j be the block that achieves the maximum gain for v, i.e., $\forall_{k \neq i} : g_j(v) \geq g_k(v)$. Ties are broken first by better balance and then by random. We move v into V_j and add all neighbors into P. However, we skip nodes that have been touched by a previous local search. If a node is already in P, we update its priority. Note that we do not move v if no feasible block V_j exists. If the expected gain from the priority queue was positive and $g_j(v)$ is negative, we add v again into P with the updated priority. This occurs when the best block at the time of adding v into P is now overloaded.

After finishing one local search we roll back to the best solution. Then we randomly select the next boundary node from B until it is empty. We repeat this whole procedure for up to 10 rounds, unless the last round did not improve the partition.

Size-Constrained Label Propagation. In the original version of KaMinPar, Gottesbüren et al. [20] used *size-constrained label propagation* [34] as refinement algorithm. We did not modify their implementation, but we still shortly describe it in the following. They iterate over all nodes in parallel. To avoid conflicts, they use two atomic operations: fetch-and-add to move nodes between blocks and compare-and-swap to update the weight of a block. This enables them to strictly enforce the maximum weight per block. They repeat the label propagation at most five times, but terminate early if no node was moved.

Ugander-Backstrom Algorithm. In our implementation of the refinement algorithm proposed by Ugander and Backstrom [46] (UB-algorithm) we construct the proposed linear program and solve it. We then execute the moves according to the values of the variables x_{ij} that specify the number of nodes that should be moved from V_i to V_j. Note that we floor each x_{ij}, because we can not partially move nodes. We repeat this up to 10 times, unless one round did not move any nodes.

However this algorithm still has one drawback. It is exclusively designed for unweighted graphs. Per design of MGP we however always obtain weighted instances on the coarse levels, even if the input was unweighted. In order to solve this problem, we have two approaches. The first approach is to only use the algorithm at the top level. For the coarse levels we use size-constrained label propagation instead. As second approach, we introduce a weighted alternative UB$_w$ to the original proposed algorithm. We now shortly explain the modifications.

We use the same variables x_{ij} to specify the number of nodes that should be moved between block V_i and V_j. For the calculation of the gain and hereby the objective, we use the same piecewise linear concave *relocation utility function*. Only the constraints for limiting the block weights needs to be updated. Recall that in the original version they simply use the difference between x_{ij} and x_{ji} to calculate the weight that is moved from V_i and V_j. This is not correct for weighted graphs. We define, similar to the relocation utility function, a function w_{ij} with

$$w_{ij}(x) := \sum_{k=1}^{x} c(m_k)$$

where $m_1, \ldots, m_K$ are the positive gain moves between V_i and V_j sorted by decreasing gain. Further $c(m_k)$ is the weight of the node corresponding to the move m_k. This means that $w_{ij}(x)$ is the weight of the top x nodes between V_i and V_j. Note that each w_{ij} is a piecewise linear function. It is however not necessary concave, unlike the relocation utility function. Hence, we can not approximate w_{ij} with a linear program. An ILP can however model any piecewise linear function with some additional variables [9]. Thus we transform the original linear program into an ILP and use

$$S_i \leq c(V_i) + \sum_{j \neq i} (w_{ji}(x_{ji}) - w_{ij}(x_{ij})) \leq T_i \tag{5.1}$$

for each i to restrict the resulting weight of each block. This is our modification to the original algorithm, which allows using it on all levels within the multi-level context.

Flow-Based Refinement Algorithm. We use the implementation of Gottesbüren et al. [18] that is integrated into the Mt-KaHyPar framework. The latter is a parallel multilevel framework for hypergraph partitioning. For each refinement step, we write the current graph and partition into a temporary file. We then call a modified version of Mt-KaHyPar with these files as input, where only one refinement step is executed, instead of a complete partitioning process. Afterwards, we read the results and modify the current graph accordingly.

We now describe the implementation of flow-based refinement by Gottesbüren et al. [18]. Recall that the general idea is to construct a flow network for a pair of blocks, where finding a maximum flow translates to finding the minimum cut between them. The details of the algorithm can be seen in Algorithm 4 and are further described in the following.

Algorithm 4: Flow-Based Refinement (taken from [18] with modifications)

```
  Input: Graph $G = (V, E)$, $k$-way partition $\Pi$
1 $\mathcal{Q} \leftarrow$ buildQuotientGraph($G, \Pi$)
2 while $\exists$ active $(V_i, V_j) \in \mathcal{Q}$ do                    // select block pair
3     $B := B_i \cup B_j \leftarrow$ constructRegion($G, V_i, V_j$)
4     $(\mathcal{N}, s, t) \leftarrow$ constructFlowNetwork($G, B$)
5     $(M, \Delta_{exp}) \leftarrow$ FlowCutterRefinement($\mathcal{N}, s, t$)
6     if $\Delta_{exp} \geq 0$ then
7         applyMoves($G, \Pi, M$)
8         if $\Delta_{exp} > 0$ then mark $V_i$ and $V_j$ as active
```

After building the quotient graph, the algorithm starts with selecting a pair of blocks to refine in Line 2. They use *active block scheduling* [39] for the selection of pairs to refine next. Roughly speaking in this scheduling strategy, they first select all pairs that share a non-empty boundary and afterwards pairs where one of the blocks changed during a previous refinement. After selecting a pair (V_i, V_j) they try to improve the total cut of the bipartitioned graph induced by $V_i \cup V_j$ (see Line 3 - 8). Therefore they grow a size-constrained region $B := B_i \cup B_j$

with $B_i \subseteq V_i$ and $B_j \subseteq V_j$ from two breadth-first-searches (BFS). This region is the subset of nodes that are eligible to move in the current refinement step. The first BFS is initialized with the boundary nodes of V_i. They add each traversed node during this search to B_i until

$$c(B_i) \leq (1 + \alpha\varepsilon) \left\lceil \frac{c(V)}{2} \right\rceil - c(V_j)$$

for an input parameter α. In the case $\alpha = 1$ this guarantees that each flow computation induces cuts that result in a balanced partition, as at most the nodes B_i can be added to V_j. Hence, the resulting weight of V_j is at most $c(B_i) + c(V_j) \leq (1 + \varepsilon) \left\lceil \frac{c(V)}{2} \right\rceil$. The second BFS constructs B_j analogously.

They now contract the nodes $V_i \setminus B_i$ and $V_j \setminus B_j$ into the source s and sink t respectively. This results in a flow network $\mathcal{N} = (B \cup \{s, t\}, \mathcal{E}, c)$ where incident edges of s and t have infinite capacity. The capacity of all other edges equals the weight of the corresponding edge in the original graph. For this network they now calculate a maximum flow. Afterwards they search a minimum cut that separates s and t by traversing the residual graph. This cut induces a new partition $\{V_i', V_j'\}$ where some of the nodes in B_i or B_j are moved to V_j' and V_i', respectively. They collect all these moves. The expected gain Δ_{exp} when executing these moves is the difference between the weight of the minimum cut (which equals the weight of the maximum flow [13]) and the total cut value of the original partition V_i, V_j. If the expected gain is greater than 0 the algorithm executes the moves and marks both blocks, which makes them eligible for further refinements. If no improvement can be found, this pair will not be scheduled for refinement again.

Node Swapping Refinement Algorithm. Furthermore, we implemented a refinement algorithm that is based on a deterministic refinement algorithm for weighted hypergraphs by Gottesbüren and Hamann [16]. There they refine by swapping some of the nodes with the highest gain between each pair of blocks. This approach is influenced by Kabiljo et al. [25]. However their version is only designed for unweighted graphs. We now describe our implementation for weighted graphs.

First, we calculate for each node $u \in V_i$ the move m_j^u with the highest gain and add v into a set of nodes S_{ij}. Note however, that we exclude nodes with only negative gain moves. Each S_{ij} then contains the nodes that *want* to move from V_i into V_j. We then refine each pair of blocks (V_i, V_j) where either $|S_{ij}| > 0$ or $|S_{ji}| > 0$, i.e., at least one node in one of the blocks wants to move to the other. The best solution would be to completely swap S_{ij} and S_{ji}, because the corresponding moves all have positive gain. But this could result in imbalanced solutions. Hence, we want to find good subsets of S_{ij} and S_{ji} that maintain balance when swapped. Therefore, we sort the nodes in decreasing gain. We now search the longest prefixes x_i and x_j of S_{ij} and S_{ji} that maintain balance when swapped, i.e.,

$$c(V_i) + c(S_{ji}[x_j]) \leq L_k \quad \text{and} \quad c(V_j) + c(S_{ij}[x_i]) \leq L_k$$

where $S[x]$ denotes the top x nodes of S and L_k is the maximum block weight. We call prefixes that maintain balance, *feasible*. From the trivial feasible solution $x_i = x_j = 0$ we iteratively increase both prefixes.

If $c(S_{ij}[x_i]) < c(S_{ji}[x_j])$, i.e., less weight is moved from V_i into V_j, we increment x_i, otherwise we increment x_j. If one prefix already contains its complete set, we always increment the other prefix. When the new prefixes are feasible, we save them as current best solution. We stop, when $x_i = |S_{ij}|$ and $x_j = |S_{ji}|$ and swap the largest feasible prefixes found. If the actual gain after swapping the nodes is negative, we revert the moves. Recall therefore Section 4.2 where we have seen, that moving adjacent nodes at once can manipulate the actually achieved gain. Then we proceed with the next block pair until we have refined all of them.

6 Experimental Evaluation

In this section, we present an extensive experimental evaluation of the introduced ILP-based refinement algorithm and the other heuristics from Section 5. We first examine the optimal parameter configuration for the former, then compare it to the latter. We integrated all evaluated refinement algorithms into the shared-memory graph partitioner KaMinPar. The code is written in `C++` and compiled with `g++-10.3` with the flags `-O3 -march=native`. As ILP solver we used Gurobi version 9.5.0 [21].

Benchmark Set. We used a subset of the benchmark set from Gottesbüren et al. [20] consisting of 55 graphs (set A). The selection strategy was based on the number of nodes and edges. We selected every graph that fulfills $10^5 \leq n \leq 10^7$ and $m \leq 10^7$. We restricted the benchmark set due to the high running times of our ILP refinement algorithm and limited computational power. Further, we set a lower limit for the number of nodes to guarantee that for high values of k (we used $1000 \leq k \leq 1600$) each block has at least 100 nodes. The set A consists of 28 graphs from the 10th DIMACS Implementation Challenge [3], 2 randomly generated graphs [14], 13 large social networks [47, 33] and 12 graphs from different application domains [50, 48]. For parameter tuning we used the benchmark set B, which is a subset of A. This set contains 12 of the smallest graphs in A that still properly represent each of the four sources. See Appendix A for a statistical evaluation of both benchmark sets.

Setup. We performed our experiments for each graph with $k \in \{1000, 1200, 1400, 1600\}$, $\varepsilon = 0.01$ and three different seeds. We did not restrict the running times. All experiments were run on a single machine equipped with an AMD EPYC 7551P processor (32 cores on one socket) clocked at 2 GHz. The machine has 256 GB DDR4 main memory.

Due to the long running times of our ILP refinement algorithm and limited computational resources, we start independent single-threaded runs for 64 instances in parallel, i.e., one for each available thread as the processor has hyperthreading enabled. Note that the memory of our machine was sufficient for 64 parallel runs. We use the same technique for the experiments of the other refinement algorithms to mitigate any influence on the relative running times or quality. Note however that due to this setup, the running times are not comparable in general. But we experienced that especially for the ILP refinement the running times differ in more than an order of magnitude, which gives a good indication of the actual execution time. The only exception from the parallel execution of multiple instances is the flow-based refinement algorithm that we execute strictly sequential. This is because the Mt-KaHyPar framework uses thread pinning to the first processor core. Hence, when we start multiple instances on one machine, they all compete for the same core.

Methodology. We define an *instance* to be a combination of a graph and a specific value for k. We use the arithmetic mean to aggregate the experimental data over the three different seeds for each instance. To further aggregate the running times over multiple instance, we use the geometric mean. Note that all runs neither produced imbalanced solution nor exceeded a time limit. Therefore we do not have to handle these cases in the aggregation of our results.

We use *performance profiles* [10] to compare the quality of the different algorithms. We now shortly introduce them following the description from Gottesbüren et al. [20]. Let $\mathcal{A}$ be the set of algorithms that we want to compare and $\mathcal{I}$ the set of instances. For each $A \in \mathcal{A}$ and $I \in \mathcal{I}$, we define $q_A(I)$ as the quality of algorithm A on the instance I. Further, let $\mathcal{I}_A(\tau) = \{I \in \mathcal{I} \mid q_A(I) \leq \tau \cdot \min_{A' \in \mathcal{A}} q_{A'}(I)\}$ be the set of instances for which A produces a solution that is less than a factor τ from the best solution for the respective instance. We now plot $\frac{|\mathcal{I}_A(\tau)|}{|\mathcal{I}|}$ on the y-axis with τ on the x-axis. The y-value for $\tau = 1$ indicates the percentage of instances where an algorithm produces the best solution. Note that if we include more than two algorithms, we can not directly use the values at $\tau = 1$ to rank the algorithms.

6.1 Algorithm Configuration

In this section we evaluate different tuning parameters for our ILP-based refinement algorithm. First we compare the random move selection with the promising cluster move selection strategy. We then evaluate every parameter for the latter. This results in the final configuration for our algorithm that we will use for the comparison with other algorithms in section 6.3. All experiments were done on the small benchmark set B, with $k \in \{1000, 1200, 1400, 1600\}$ and $\varepsilon = 0.01$. Note that $\varepsilon = 0.03$ is a commonly used value, we however want to explore stricter balance constraints, as we expect that ILPs have a greater potential for improvements here. Note that we always call label propagation before each refinement to bring the partition to a local optimum. Further, we used five repetitions of our algorithm, i.e., $i_{max} = 5$.

Evaluating Different Move Selection Strategies. We start our parameter configuration with the move selection strategy. In Section 4.4 we have presented two different strategies: Random and PromisingCluster. The former selects random moves for the ILP, while the latter clusters the quotient graph and refines each cluster individually. For this evaluation we selected the optimal parameter configuration for both strategies. For PromisingCluster the optimal configuration is a maximum number of 50 000 non-zeroes, a maximum cluster size of 10 and a time limit of 20 seconds for each ILP. For Random we limit the non-zeroes to 100 000 and use the same time limit for each ILP. We evaluated the latter strategy for up to 2 million non-zeroes and a time limit of 120 seconds and could not observe an increase in quality.

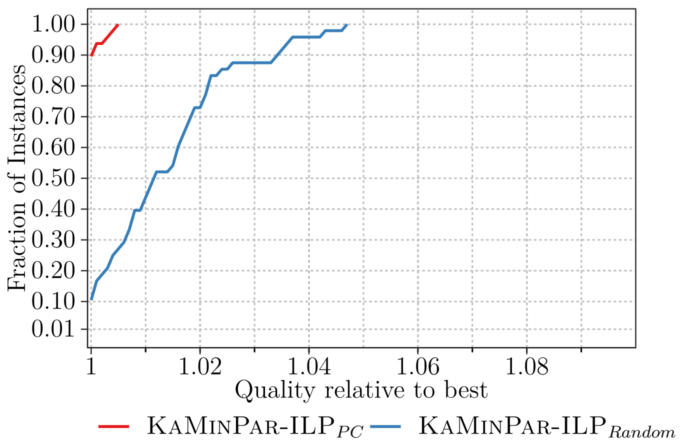


Figure 4: Comparison of the Random move selection strategy KaMinPar-ILP$_{Random}$ with the PromisingCluster strategy KaMinPar-ILP$_{PC}$. We performed the experiments with $\varepsilon = 0.1$ and $k \in \{1000, 1200, 1400, 1600\}$.

Figure 4 shows the difference in solution quality between both selection strategies. We see that the PromisingCluster strategy outperforms the other. The quality drops in the median by about 1% and for 10% of the instances by more than 3%. This underlines the importance of a good move selection strategy on the general quality. Hence, we focus our following evaluations completely on the PromisingCluster move selection strategy.

ILP Solver Time Limit. We now determine the optimal parameter configuration for the PromisingCluster move selection strategy. One tuning parameter is the maximum running time of each individually constructed ILP, i.e., each cluster. We stop the solver after this maximum time and use the best solution found. This is not an optimal solution in most cases, but in some cases we can still improve the cut. To find a proper time limit we evaluated a run with a time limit of 60 seconds and a maximum of 50 000 non-zeroes. Figure 5 left shows the results. It visualizes the sum of all gains found from each ILP run after x seconds per graphs. The sums are presented relative to the sum of all final gains, i.e., the gain of an optimal solution or of the best solution after running into the time limit. We have selected three exemplary graphs. Two other graphs from the benchmark set show similar results as `scircuit`. All other graphs show results between the curve of `m14b` and `amazon0302` with a stronger shift towards the distribution of the latter.

We can see that the solver finds almost no further improvements after 20 to 30 seconds. Over 75% of the total gain is found within the first ten seconds. About half of all graphs show almost no improvement after five seconds. This motivates using a time limit between 5 and 20 seconds. To further study this, we have evaluated the solution quality for a time limit of 1, 5, 10, 20 and 60 seconds. The quality between 10, 20 and 60 seconds is almost identical. Reducing the time limit to 5 seconds slightly decreases the quality (see Figure 5 right). Further, we can observe a significant drop for a limit of 1 second. For 20% of the instances, the quality drops by over 2%.

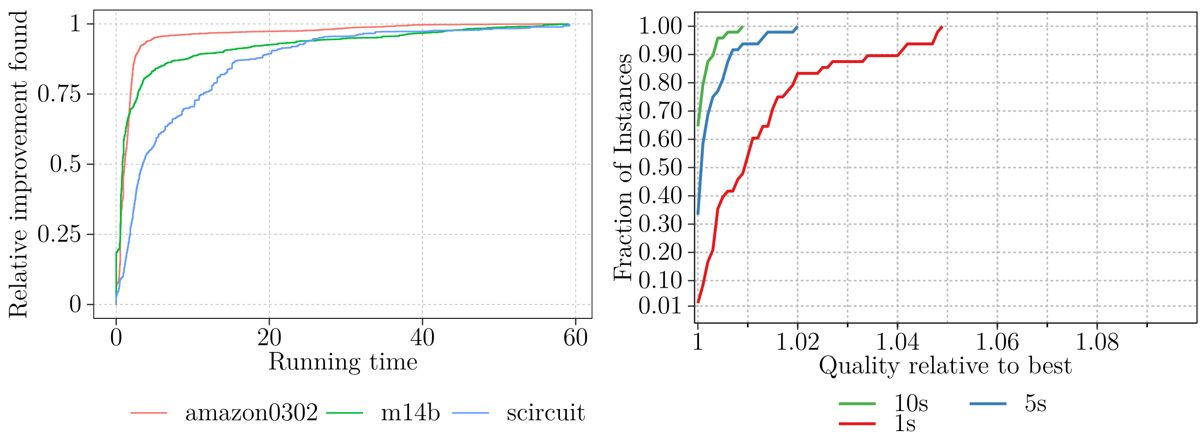


Figure 5: Left: The improvements found by the ILP solver after x seconds relative to the highest gain after a maximum of 60 seconds. Right: Solution quality for a time limit of 1, 5 and 10 seconds.

Besides the quality, we also evaluated the percentage of ILP runs that were stopped due to the time limit (see Figure 6). For a limit of 5 and 10 seconds this percentage is almost identical (both rounded 43%). When increasing the limit to 20, we see a drop to 26%. For 60 seconds still 16% of the ILP runs did not finish within the limit. This motivated us to use a limit of 20 seconds for further parameter tuning. This limit decreases the running time (geometric mean running time of 13 195s) compared to 60 seconds (29 233s) and further has a significant lower ILP time out percentage compared to 5 and 10 seconds. We want to reduce the time out percentage for parameter tuning to mitigate any effect of the time limit on the evaluation of the other parameter. However, for the final evaluation we used a time limit of 10 seconds because of the faster running times (8093s) and limited computational resources.

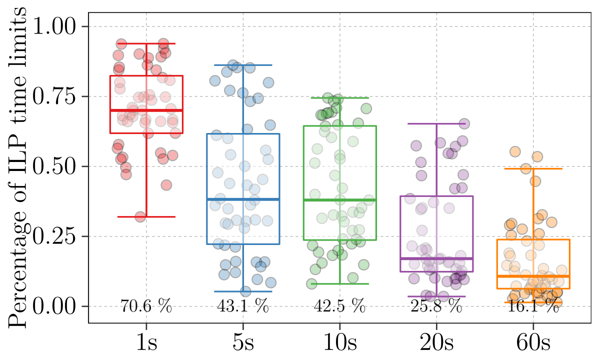


Figure 6: Percentages of ILPs that were stopped due to the time limit for a limit of 1, 5, 10, 20 and 60 seconds. The percentages at the bottom indicate the geometric mean over the time out ratio of all instances.

Maximum Cluster Size. For the PromisingCluster move selection strategy, we cluster the quotient graph and construct an individual ILP for each cluster. Each cluster contains hereby a maximum number of blocks c_{max}. Selecting small values for c_{max} restricts the search space for possible refinements as nodes can only move within blocks of its cluster. On the other hand, using large values increases the number of possible move candidates which could require larger and more complicated ILPs. To find optimal values for c_{max} we started by analyzing the actual sizes of the clusters when using $c_{max} = 30$. The experiments were done with a maximum of 50 000 non-zeroes. We observe that the clustering for almost every graph falls into one of two categories. Either most of the clusters have minimum size (2) or maximum size (30), where each case occurs about the same number of times. The relative frequency of each cluster size for four exemplary graphs can be found in Figure 7 left. The graph `coAuthorsCiteseer` is the graph where the trend is least visible over the whole benchmark set. More than 50% of the clusters have minimum size, however also around 25% have maximum size. The results for all other graphs are similar to one of the other three depicted graphs.

Figure 7 shows on the right side the relative gain achieved by refining cluster with a specific size. This is the ratio between the sum of the gains for a specific cluster size and the total gain over all clusters. The results show that most of the graphs where minimum sized clusters are dominant also achieve most of the gain via theses cluster (see `coAuthorsCiteseer`). For graphs where maximum sized clusters predominate, nevertheless minimum sized clusters achieve the most gain. Only two of them also have the highest gain in this cluster size (see e.g. `citationCiteseer`). For the other graphs the total gain is either almost split between cluster size 2 and 30 or up to 75% of the total gain was found in minimum sized clusters (see `coAuthorsDBLP`). The graph `m14b` is the only exception where the gain is almost equally split between clusters of size 2 and 30, whereby almost all clusters have size 2.

Looking at the solution quality for different maximum cluster sizes, we can see that it depends on the maximum number of non-zeroes (see Figure 8). For lower number of non-zeroes (50 000), the quality drops by almost 0.5% in the median and for 20% of the instances by over 1% when increasing c_{max} from 10 to 30. When using higher limits on the non-zeroes, e.g. 150 000, we observe almost no difference in quality. However, using $c_{max} = 30$ (geometric mean running time 7697s) is notably faster than $c_{max} = 10$ (14 540s). Still, both running times are more than an order of magnitude higher than the times of all other refinement algorithms. To conclude, we select in the following $c_{max} = 10$ because we focus on higher quality instead of lower execution time.

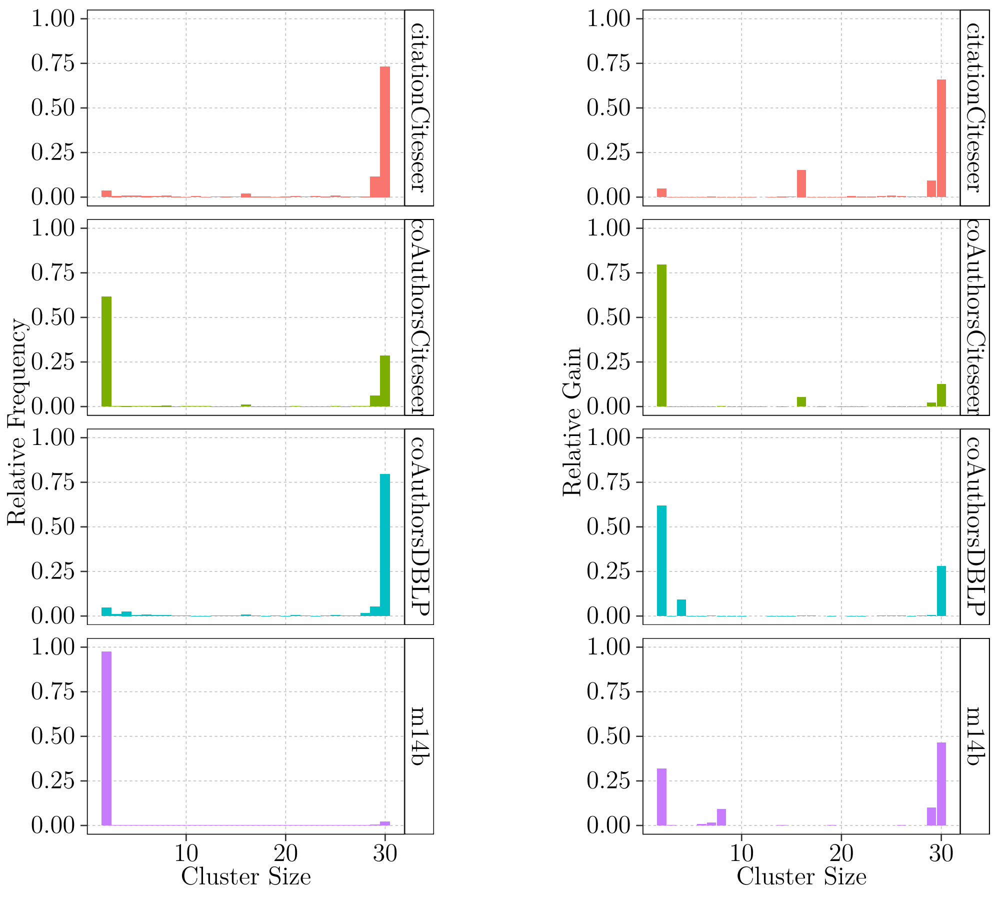


Figure 7: Comparison of the frequency (left) and relative gains (right) of the different cluster sizes for four exemplary graphs.

Restricting Number of Non-zeroes. We restrict the maximum number of *non-zeroes* (z_{max}) for each constructed ILP. Figure 9 shows a pairwise comparison when using a maximum number of 50 000, 100 000 and 150 000 non-zeroes. We see almost no difference in quality between each value for z_{max}. However, using 50 000 non-zeroes results for most instances in the lowest running time. The geometric mean running time is about 10% lower compared to the other values for z_{max}. The running times for 100 000 and 150 000 non-zeroes are almost identical. We further examined the percentage of ILP runs that were stopped due to reaching the time limit of 20 seconds. Interestingly, we do not see a significant difference here, analogously to the quality. This shows that the time limit is here probably no bottleneck for the quality.

In the following, we use a limit of 50 000 non-zeroes because of the slightly better running time and similar performance. Note however, that this limit is absolute and does not scale with the size of the input graphs. We therefore expect, that the optimal limit for z_{max} differs when using smaller or larger graphs.

6.1.1 Impact of Different Optimization Strategies

Next, we evaluate multiple approaches to mainly reduce the running times of our algorithm, as this is its biggest drawback. Several results from the evaluation of our algorithm motivated us to try different optimizations. We present these motivations and evaluate the impact of the proposed optimization strategy concerning execution time and quality. Unfortunately, most optimizations showed an unfavorable trade-off between running time reduction and solution quality degradation. We still shortly present the optimization attempts in the following.

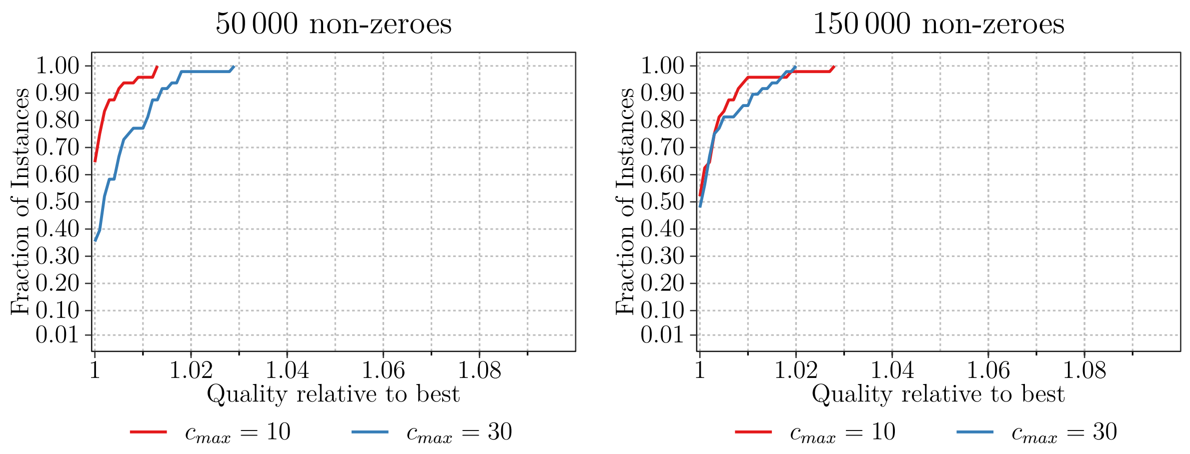


Figure 8: Solution qualities for different number of non-zeroes and different maximum cluster sizes c_{max}. We used a time limit of 20 seconds.

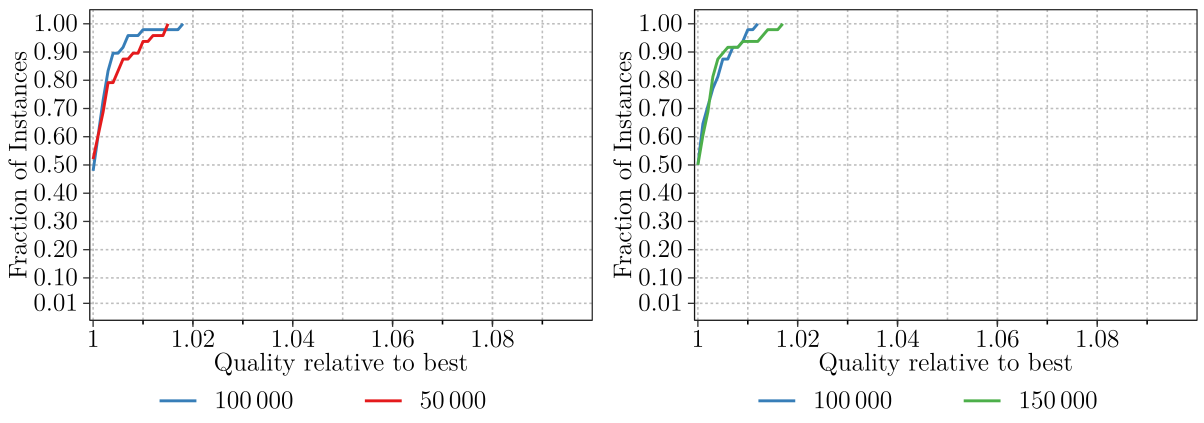


Figure 9: Solution qualities when using different maximum numbers of non-zeroes per constructed ILP. We executed each run with an ILP time limit of 20 seconds and $c_{max} = 10$.

Early Stopping Policies. Instead of using a strict time limit for each ILP run, we can limit the number of improvements after which we stop the solver. The motivation for this is that we found that the first and second improvement had the most impact relative to the final gain. The first improvements are found in about 75% of the cases in under 10 seconds. Afterwards, the solver often searches unsuccessfully for further improvements. Limiting the maximum number of improvements has therefore the potential to decrease the running times while maintaining similar quality. Note that this approach differs from setting a lower time limit for the ILPs. When using a strict time limit, every ILP is stopped after this limit. Here, optimizations that need a long time to find any gain can run longer. We only stop ILPs earlier where a solution is found fast.

The experimental evaluation shows that stopping after the second improvement can decrease the running time. Note however, that they are still more than an order of magnitude slower than for other refinement algorithms. However the solution quality also drops in the median by 0.5% and for 20% of the instances by over 1%. Stopping after the third improvement neither reduces the running times nor decreases the quality significantly. We do not use this approach, as either the solution quality dropped to far or the running times did not improve.

Ignore Unchanged Regions. Kernighan and Lin [32] propose a strategy to schedule the refinement of pairs of blocks. The general idea is that only pairs where at least one of the block changed in a previous round are eligible for refinement. Sanders and Schulz [39] use a similar approach, called *active block scheduling*. In our algorithm we select cluster of blocks that should be refined. Since this is just a generalization of refining pairs of blocks, we tried a similar approach.

In the first round we refine every cluster. Afterwards, we skip a cluster if all of its blocks did not change in the previous round. With this optimization we could achieve a speed up of about 2. However, the solution quality dropped in the median by 0.5%. For 20% of the instances, it dropped by more than 1%. Further, the version without this approach records a better quality in over 95% of the instances (46 out of 48). We expect that this is due to the heuristic approach for selecting moves. If in one round the ILP could not find any improvements for a cluster of blocks, it could still find improvements in the next round if for example the blocks are differently clustered or the move selection strategy selects different moves.

Although we can obtain a good speed up with this optimization, we did not include it in the configuration for the final evaluation. This is because the running time is still more than an order of magnitude slower compared to the other algorithms. Therefore we focus again on solution quality instead of faster running times.

Increase Locality of Selected Moves. The gain of each solution from a single ILP is the sum of two terms. Firstly the sum of the gains of the individual moves, i.e., the local gain. Secondly the sum of the conflict values. Recall therefore that moving two nodes at once can create a higher or lower gain than originally expected. The conflict value equals this mismatch between the local and the actual gain. We have evaluated which of the two parts dominates the actual gain of the individual ILPs. The results for four representative graphs can be seen in Figure 10. Here we see on the y-axis the difference between the conflict value and the local gain. This means positive values indicate that the conflict value was higher than the local gain. Note that we removed every ILP run that achieved no gain at all.

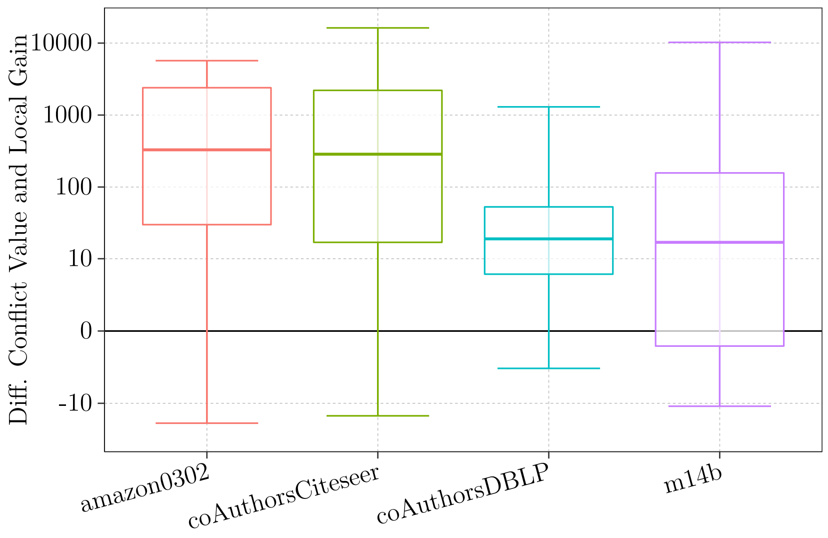


Figure 10: Distribution of the difference between the conflict value and local gain of each ILP for four representative graphs. Positive y values correspond to a higher conflict value than local gain.

We can see that in most cases, the conflict value dominates. This also holds for all other graphs. The geometric mean of the difference lies for all graphs between 10 and 1000. This indicates that the solver achieves the most gain by moving groups of adjacent nodes, as only these can create conflicts. This motivated us to try and modify the method to find moves within a cluster as follows:

In Algorithm 3 we have introduced a strategy to select moves in a given cluster, which we call the TopGain-version. Generally speaking, we select the moves between the blocks with the highest gain. Now, we instead pick a random node from the boundary of the blocks. Then we start a breadth-first search (BFS) from this node with a maximum depth d. For each touched node we add $\gamma|C|$ moves to the ILP, where $|C|$ is the size of the current cluster and $0 < \gamma < 1$ is a tuning parameter. This means, we select $\gamma|C|$ different target blocks for each node. Our strategy is to select the target with the highest gain and fill with random blocks until the desired number of moves is reached. After we processed every node from the BFS, we repeat this process with a new random node. We repeat this until our move buffer M has enough moves for the current cluster. We call this modification the BFS-version.

We made an initial guess of $d = 2$ and $\gamma = 0.7$. We then compared the modified version using these parameter with the original version in the currently optimal configuration ($c_{max} = 10$, $z_{max} = 50\,000$ and a time limit of 20 seconds). For the BFS-version we used the same c_{max}, z_{max} and time limit. The solution quality dropped for the modified version in the median by 0.5% and for 20% of the instances by between 1.5% and 4%. The BFS-version (geometric mean running time 4463s) is however also significantly faster than the original version (13 194s). With further parameter tuning, we could improve the quality of the BFS-version by using $z_{max} = 100\,000$ and $d = 4$. This however resulted in still slightly worse quality and almost similar running time as in the original version.

An hypothesis for the drop in quality is that the BFS-version adds too few moves with high individual gains. Hence we modified the method of selecting the root nodes for the individual BFS searches. Instead of picking a random boundary node, we selected the next node according to the TopGain strategy, i.e., we pick the nodes with the highest gains. This resulted in similar results considering quality and performance compared to the original version.

To conclude, we chose neither of the two modification and did not investigate increasing the locality of selected moves further. Although the BFS-version with the initial parameters significantly reduces the running time, we did not chose this version as the quality dropped while still not competing with the other algorithms in terms of execution time. We also did not use the BFS-version with other parameters or the combination of both methods, since we could not notice any improvement.

Balance as Secondary Objective. Graph partitioning is not only about minimizing the total cut, but also about maintaining balance between the blocks. Hence, we modified our ILP to not only maximize the gain, but also maximize the balance as secondary objective. This means that the solver firstly optimizes the gain. Afterwards, it uses balance as tiebreaker. Note that we restricted the running time for the first optimization to a fraction of the total time limit per ILP. This guarantees that the solver optimizes the balance for at least some time.

In order to model balance as objective, we add a new variable w to the ILP. We then restrict this variable to be greater than the linear expressions that calculates the resulting weight of each block, i.e.,

$$\forall 1 \leq i \leq k : W^i_{in} - W^i_{out} + c(V_i) \leq w$$

where W^i_{in} and W^i_{out} evaluate to the total weight of each node moving into or out of V_i, respectively. The left side is identical to the balance constraint of our original ILP (see Definition 4.2). This means that w is always greater than the weight of the heaviest block. We now minimize

w. This maximizes balance, as the maximum block weight is minimal if the blocks are perfectly balanced.

We compared the quality and performance when using balance as secondary objective with the current optimal configuration. Therefore, we used the same values for c_{max}, z_{max} and time limit. The quality is almost identical while the running time increased by about 30%. Therefore, we did not use this approach for our final configuration.

6.2 Insights into ILP-based Refinement

Apart from parameter optimization, we also evaluated general properties of our refinement algorithm. We experimented with parallelization and we evaluated statistics from the ILP solver itself. In this section, we explain our findings.

Parallelization. We evaluated two different approaches at parallelizing our refinement algorithm. In the first version, we run Gurobi - the used ILP solve - in its multi-threaded version, i.e., each individual ILP is solved using multiple threads. The rest of the algorithm remains sequentially. In our second version, we construct and solve the ILPs for the different move buffers in parallel. This means, we parallize the `foreach` loop starting at Line 3 of Algorithm 1. In our PromisingCluster move selection strategy this translates to refining the different clusters of blocks in parallel. Note that this also moves nodes in parallel. However, this does not create any conflicts because the source and target blocks of moves in different move buffers are always disjoint. This is because the moves are only within its respective cluster. Therefore we can safely refine in parallel.

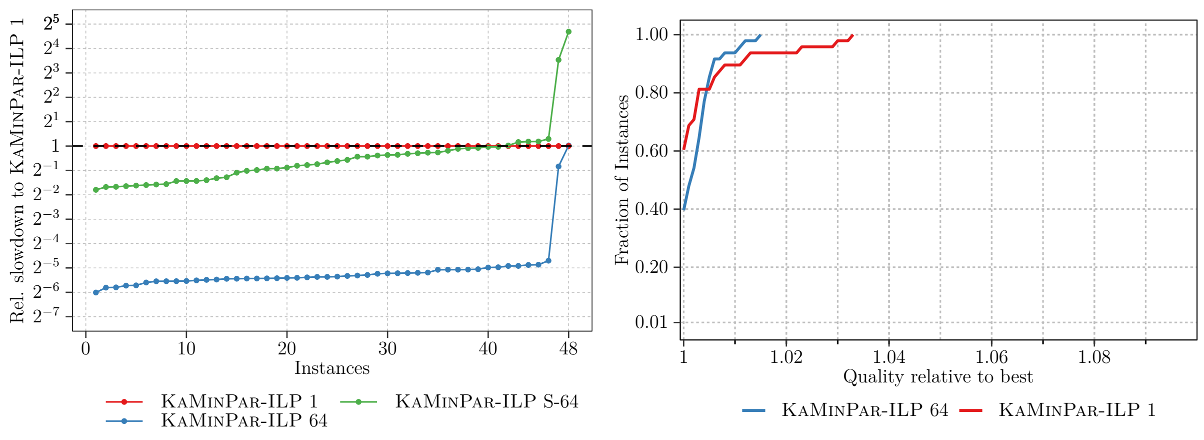


Figure 11: Comparison of sequential and parallel quality and performance. Left: Box plot of the running times when executing with one thread (KaMinPar-ILP 1), with 64 threads (64) and when executing only the solver for each ILP with 64 threads (S-64). Right: Comparison of the solution quality between sequential and multi-threaded execution.

We now evaluate the performance of both versions compared to the fully sequential version. Note that we did not invoke label propagation before our algorithm (neither in the parallel nor sequential versions). This is to minimize any external effects from the parallelization of label propagation. In Figure 11, on the left, we can see that using multiple solver threads only slightly reduces the running times (see KaMinPar-ILP S-64). When we however solve multiple ILPs in parallel, we can observe a significant speed up. In the version with 64 threads, the geometric mean running time is over 30 times faster than that of the sequential version.

As expected, the quality is nevertheless similar, since the ILPs do not conflict with each other (see Figure 11 right). Note that the quality when executing the ILP solver in parallel is also almost identical.

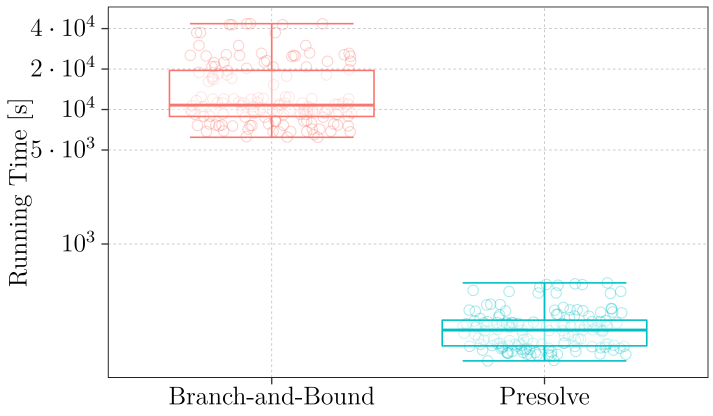


Figure 12: Comparison of the running times for the presolve and branch-and-bound phase of the ILP solver.

Solver Statistics. The ILP solver can reduce the size of an ILP before starting the branch-and-bound algorithm. To do so, it searches for multiple inequalities that together can be replaced by a simpler one. This step is called *presolve*. It is beneficial for the running times of an ILP if a high amount of work is done in the presolve phase. We evaluated the times of our ILP optimizations inside the presolve and the branch-and-bound phase. The results can be seen in Figure 12. We see that the time inside the branch-and-bound algorithm significantly outweighs the time for the presolve phase. This is partly because the latter is in general faster than the former. Moreover it also shows that our ILPs do not leave many possibilities for reducing the size during presolving. This has a negative impact on the overall running times.

6.3 Comparison of Different Refinement Algorithms

Now, we compare our ILP-based refinement algorithm in its final configuration with the other algorithms described in Section 5. Recall that we presented two variants of the Ugander-Backstrom algorithm: the original variant (UB) for unweighted graphs and our extension to weighted instances (UB$_w$). We evaluated both variants on our large benchmark set A. The solution quality is similar for both versions. In terms of running time, UB has a slight advantage over UB$_w$ (geometric mean running time of 64s vs 73s). This is however expected as in the former we use the Ugander-Backstrom algorithm only on the top levels and resign to label propagation otherwise, while in the latter we use both algorithms on all levels. Moreover UB$_w$ has a few outliers with over 1000 seconds of execution time. We expect that this is due to UB$_w$ using ILPs instead of simple linear programming. We therefore used the original version for our comparison with the other algorithms as the quality is comparable while it is more reliable considering execution time. For all other algorithms, we used the default parameters as stated in Section 5.

We ran our experiments on the large benchmark set A with $k \in \{1000, 1200, 1400, 1600\}$ and $\varepsilon \in \{0.01, 0.001\}$. We first present the results for $\varepsilon = 0.01$. Afterwards, we shortly outline the difference that arise when reducing ε to 0.001.

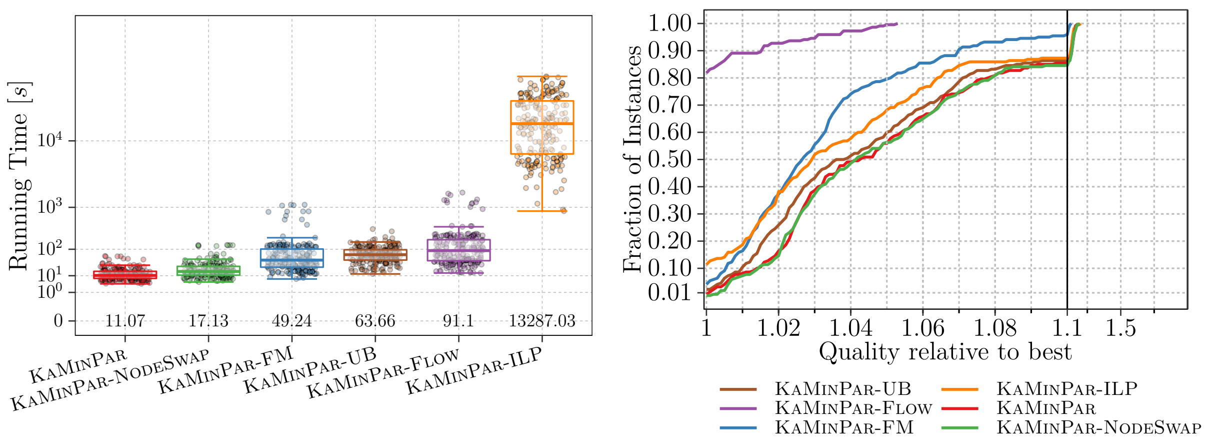


Figure 13: Comparison of running times (left) and solution quality (right) between all algorithms for $k \in \{1000, 1200, 1400, 1600\}$ and $\varepsilon = 0.01$.

Running Time. We start with the evaluation of the running times. Figure 13 shows box plots for the running times per instance for each of the algorithm. The numbers indicate the geometric mean running time. We can see that KaMinPar in its original version using size-constrained label propagation is the fastest algorithm (geometric mean running time of 11s). This is however expected, as all other algorithms invoke label propagation before their own refinement. The second fastest is the NodeSwap algorithm (17s). It is only slightly slower than label propagation. The FM-algorithm has the next lower geometric mean running time (49s). Note however that some instances have a running time of up to 1000 seconds. The Ugander-Backstrom Algorithm (UB) is on average slightly slower (64s), however we see less variance in the running times. Flow-based refinement is the second slowest algorithm (91s) with peaks up to over 1000 seconds per instance. The slowest algorithm is the ILP refinement with an average running time of over 10 000 seconds. We see that using ILPs for refinement is several orders of magnitude slower than all other methods.

Solution Quality. We now evaluate the solution quality of the algorithms in a pairwise fashion. The results are shown in Figure 14 and refer to Appendix B for further pairwise comparisons. Hereby, we start with the two fastest algorithms, the original KaMinPar configuration and the NodeSwap algorithm. We see that almost no difference in quality exists. Hence we prefer the label propagation algorithm from the original version as it is faster. When we now compare this algorithm with the FM-algorithm, we see that the latter produces higher quality solutions. For over 80% of the instances it produces better solutions and in the median they have 2% lower edge cuts. The FM-algorithm finds also better solutions than the Ugander-Backstrom algorithm (UB), although running times of the latter are slightly higher. In comparison with label propagation, UB produces slightly better solutions. In the median, the instances have a 0.5% lower cut. The algorithm with the next highest running times is flow-based refinement (Flow). This algorithm clearly outperforms the FM-algorithm. For 90% of the instances flow-based refinement finds the better solutions with in the median 2.5% lower edge cuts. Flow-based refinement as well as the FM-algorithm outperform our ILP-based refinement by 3% and 1% in the median, respectively. However, ILP-based refinement has slightly higher quality than UB. The former has in the median 0.5% lower edge cuts.

To conclude, flow-based refinement shows the most promising results. It produces the best solutions while having moderate running times. The FM-algorithm is faster but shows also worse quality, however still better than all other algorithms. The NodeSwap and UB algorithms show no promising results as they are slower than label propagation while having similar quality. Further, the quality of the ILP refinement algorithm does not justify its running time, as FM and flow-based refinement both produce better solutions while being three orders of magnitude faster.

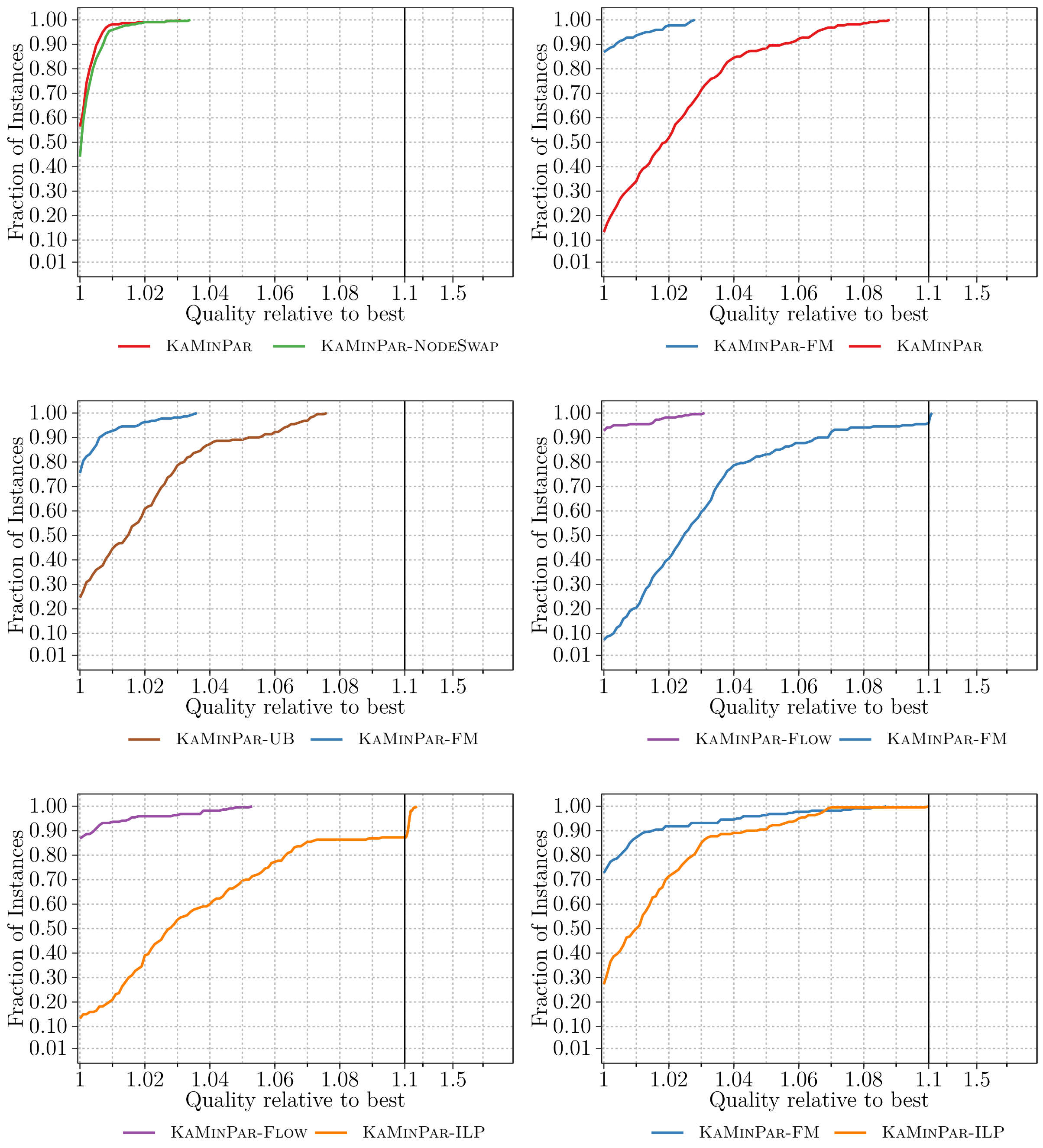


Figure 14: Pairwise comparisons of the solution quality for $\varepsilon = 0.01$ on the benchmark set A.

Running Time and Solution Quality for Small ε. We further compared the refinement algorithms when using $\varepsilon = 0.001$ instead of $\varepsilon = 0.01$, i.e., with a stricter limit on the maximum block weights. The results can be seen in Figure 15 with further pairwise comparisons in Appendix B. Note that we performed the experiments on the smaller benchmark set B, due to limited computational resources. We can see that the running times decreased for all algorithms, however the relative difference between each of them does not significantly change. The ILP refinement remains clearly the slowest algorithm. In terms of quality, we can see that the difference between flow-based refinement and the other algorithms shrinks (see Figure 15 right). Especially the ILP refinement relatively improved. The median difference decreased from almost 3% to 1.5% when comparing them directly. Compared to the FM-algorithm, it could also slightly improve in terms of quality. It now finds for 40% instead of 30% of the instances a better solution. Interestingly, the node swapping algorithm achieves in this case better solutions than label propagation, unlike for $\varepsilon = 0.01$. We expect that this is due to the fact that when many blocks are close to the weight limit, it is hard to find individual moves that respect the balance constraint. Swapping nodes can however move nodes more easily with respect to the balance.

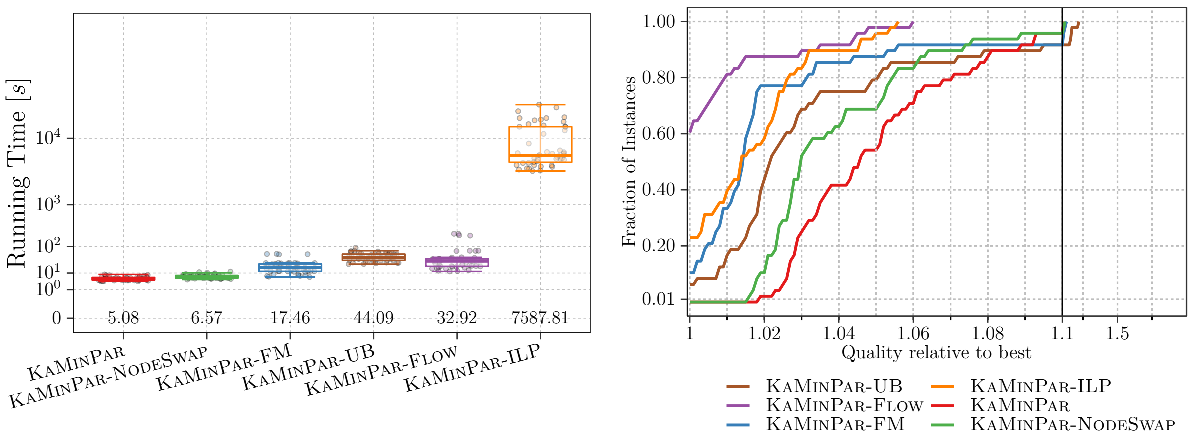


Figure 15: Comparison of running times (left) and solution quality (right) between all algorithms for a tighter balance constraint, i.e., $\varepsilon = 0.001$.

7 Conclusion

In this work, we presented a novel refinement algorithm based on integer linear programming and several other advanced refinement heuristics. We implemented the algorithms in the KaMinPar [20] framework. Then, we evaluated the solution quality and running times for partitioning with large k.

Our ILP-based refinement algorithm constructs an ILP from a set of move candidates. Optimizing it results in the subset of moves that maximize the gain when executed. We introduced constraints that properly predict if two moves are conflicting when executed in parallel. Therefore, the ILP always predicts the accurate gain. We introduced two different strategies to find promising move candidates, i.e., moves with high potential to decrease the cut: A random selection strategy and one that clusters the quotient graph and looks for the moves with the highest gain within each cluster. The former strategy did not provide promising results, hence we used the latter strategy for the comparison with other heuristics. Our algorithm allows parallelization with good speedups, because each cluster can be refined in parallel without conflicts.

We compared our refinement algorithm with size-constraint label propagation [34], multi-try k-way FM local search [39], the Ugander-Backstrom algorithm [46], flow-based refinement [18] and a refinement algorithm based on swapping groups of nodes between blocks [16]. The first algorithm is used in the original version of KaMinPar. Multi-try k-way FM local search is a variant of the FM-algorithm, where individual local searches are started from random border nodes. The UB-algorithm uses linear programming to decide which prefix of the highest gaining nodes should be moved between blocks. Flow-based refinement creates a flow network between a block pair and maximizes the flow to minimize the induced cut.

Our experiments with $1000 \leq k \leq 1600$ and $\varepsilon = 0.01$ show that flow-based refinement provides the overall best solution quality. This algorithm creates the best solutions for over 80% of the instances in our benchmark set with 55 graphs in comparison to all other algorithms. Furthermore, the running times are still reasonable (geometric mean running time 91s). The k-way FM-algorithm is almost twice as fast in the mean (49s) while producing the second best results with 2.5% worse quality in the median compared to the flow-based approach. The overall fastest algorithms are label propagation (11s) and the node swapping strategy (17s). Both achieve similar, but overall the worst quality. However, all other algorithms use label propagation as preprocessing step in their refinement. Our ILP-based refinement algorithm is three orders of magnitude slower (13 287s) and still the quality is slightly worse than that of k-way FM. The solution quality of the UB-algorithm is between label propagation and our ILP-based refinement and it runs slightly faster (64s) than flow-based refinement. In total, flow-based refinement and k-way FM are looking promising in terms of solution quality while label propagation or node swapping provide fast execution times.

When we use a more strict balance constraint, i.e., $\varepsilon = 0.001$, we see that the quality of the ILP-based and node swapping refinement improves relative to the other algorithms. However, flow-based refinement provides still the highest quality over all. Furthermore, the relative running times are similar to $\varepsilon = 0.01$.

7.1 Future Work

Flow-based refinement and the FM-algorithm have shown promising results, hence it might be worth exploring more in these directions. For flow-based refinement, we used the implementation from Mt-KaHyPar [18] that was developed for hypergraphs instead of graphs and small values for k. Thus, we have to transform the graphs into hypergraphs before refining which creates much more overhead than necessary. Therefore, exploring implementations that are specifically tailored for graphs and large k might reduce its running times. Further, we did not examine parallelism for algorithms other than our ILP-based approach. Parallel implementations of for example the k-way FM-algorithm could be worth exploring.

Although we achieved unpromising results for ILP-based refinement in this work, we still see approaches for further research. One might try different *move selection strategies*. We have seen that the chosen strategy can significantly impact the quality. Therefore, different approaches might improve the quality. Further, we believe that the strength of ILP-based refinement lies in situations with strict balance constraints. The results for small ε support this assumption. Hence, it might be worth exploring *perfectly balanced partitioning*. Further, one might evaluate the performance for small values of k. Lastly, for parallelization we have made use of the fact that our refinement algorithm improves only small regions of the graph at once, whereby each region can be processed independently. This might be a useful property for applying our algorithm in *distributed memory* settings.

References

[1] Y. Akhremtsev, P. Sanders, and C. Schulz. “High-Quality Shared-Memory Graph Partitioning”. In: *IEEE Transactions on Parallel and Distributed Systems* 31.11 (2020), pp. 2710–2722. ISSN: 1045-9219. DOI: 10.1109/TPDS.2020.3001645.

[2] C. Aykanat, B. B. Cambazoglu, F. Findik, and T. Kurc. “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. ISSN: 07437315. DOI: 10.1016/j.jpdc.2006.05.005.

[3] D. A. Bader, H. Meyerhenke, P. Sanders, D. Wagner, and editors. *Graph Partitioning and Graph Clustering: 10th DIMACS Implementation Challenge Workshop*. American Mathematical Society and Center for Discrete Mathematics and Theoretical Computer Science, 2012.

[4] E. Beale. *Mathematical Programming in Practice*. London: Sir Isaac Pitman & Sons Ltd., 1968.

[5] C.-E. Bichot and P. Siarry. *Graph Partitioning*. London and Hoboken: ISTE Ltd and John Wiley & Sons, Inc, 2011. ISBN: 978-1-84821-233-6.

[6] T. N. Bui and C. Jones. “Finding Good Approximate Vertex and Edge Partitions is NP-hard”. In: *Information Processing Letters* 42.3 (1992), pp. 153–159. ISSN: 00200190. DOI: 10.1016/0020-0190(92)90140-Q.

[7] A. Buluc, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. *Recent Advances in Graph Partitioning*. Nov. 13, 2013. URL: http://arxiv.org/pdf/1311.3144v3.

[8] Ü. V. Çatalyürek et al. *More Recent Advances in (Hyper)Graph Partitioning*. May 26, 2022. URL: http://arxiv.org/pdf/2205.13202v3.

[9] G. B. Dantzig. “On the Significance of Solving Linear Programming Problems with Some Integer Variables”. In: *Econometrica* 28.1 (1960), p. 30. ISSN: 00129682. DOI: 10.2307/1905292.

[10] E. D. Dolan and J. J. Moré. “Benchmarking optimization software with performance profiles”. In: *Mathematical Programming* 91.2 (2002), pp. 201–213. ISSN: 0025-5610. DOI: 10.1007/s101070100263.

[11] S. Dutt. “New faster Kernighan-Lin-type graph-partitioning algorithms”. In: *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*. IEEE Comput. Soc. Press, 1993, pp. 370–377. ISBN: 0-8186-4490-7. DOI: 10.1109/ICCAD.1993.580083.

[12] C. M. Fiduccia and R. M. Mattheyses. “A Linear-Time Heuristic for Improving Network Partitions”. In: *19th Design Automation Conference*. IEEE, 1982, pp. 175–181. ISBN: 0-89791-020-6. DOI: 10.1109/DAC.1982.1585498.

[13] L. R. Ford and D. R. Fulkerson. “Maximal Flow Through a Network”. In: *Canadian Journal of Mathematics* 8 (1956), pp. 399–404. ISSN: 0008-414X. DOI: 10.4153/CJM-1956-045-5.

[14] D. Funke, S. Lamm, U. Meyer, P. Sanders, M. Penschuck, C. Schulz, D. Strash, and M. v. Looz. *Communication-free Massively Distributed Graph Generation*. Oct. 20, 2017. URL: http://arxiv.org/pdf/1710.07565v3.

[15] M. R. Garey, D. S. Johnson, and L. Stockmeyer. “Some simplified NP-complete problems”. In: *Proceedings of the sixth annual ACM symposium on Theory of computing - STOC ’74*. Ed. by R. L. Constable, R. W. Ritchie, J. W. Carlyle, and M. A. Harrison. New York, New York, USA: ACM Press, 1974, pp. 47–63. DOI: 10.1145/800119.803884.

[16] L. Gottesbüren and M. Hamann. *Deterministic Parallel Hypergraph Partitioning*. 2021. URL: http://arxiv.org/pdf/2112.12704v1.

[17] L. Gottesbüren, M. Hamann, S. Schlag, and D. Wagner. *Advanced Flow-Based Multilevel Hypergraph Partitioning*. 2020. DOI: 10.4230/LIPIcs.SEA.2020.11.

[18] L. Gottesbüren, T. Heuer, and P. Sanders. *Parallel Flow-Based Hypergraph Partitioning*. Jan. 5, 2022. URL: http://arxiv.org/pdf/2201.01556v1.

[19] L. Gottesbüren, T. Heuer, P. Sanders, and S. Schlag. “Scalable Shared-Memory Hypergraph Partitioning”. In: *2021 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. Ed. by M. Farach-Colton and S. Storandt. Philadelphia, PA: SIAM, 2021, pp. 16–30. ISBN: 978-1-61197-647-2. DOI: 10.1137/1.9781611976472.2.

[20] L. Gottesbüren, T. Heuer, P. Sanders, C. Schulz, and D. Seemaier. *Deep Multilevel Graph Partitioning*. 2021. DOI: 10.5445/IR/1000138388.

[21] Gurobi Optimization LLC. *Gurobi Optimizer*. URL: https://www.gurobi.com/ (visited on 07/03/2022).

[22] B. Hendrickson and R. Leland. “A Multi-Level Algorithm For Partitioning Graphs”. In: *Proceedings of the 1995 ACM/IEEE Supercomputing Conference ; Supercomputing ’95 : San Diego, California, USA December 3 through 8, 1995*. Ed. by S. Karin. ACM, 1995. ISBN: 0-89791-816-9. DOI: 10.1145/224170.224228.

[23] A. Henzinger, A. Noe, and C. Schulz. “ILP-Based Local Search for Graph Partitioning”. In: *ACM Journal of Experimental Algorithmics* 25 (2020), pp. 1–26. ISSN: 1084-6654. DOI: 10.1145/3398634.

[24] T. Heuer, P. Sanders, and S. Schlag. “Network Flow-Based Refinement for Multilevel Hypergraph Partitioning”. In: *ACM Journal of Experimental Algorithmics* 24 (2019), pp. 1–36. ISSN: 1084-6654. DOI: 10.1145/3329872.

[25] I. Kabiljo, B. Karrer, M. Pundir, S. Pupyrev, and A. Shalita. “Social hash partitioner”. In: *Proceedings of the VLDB Endowment* 10.11 (2017), pp. 1418–1429. ISSN: 2150-8097. DOI: 10.14778/3137628.3137650.

[26] R. Kannan and C. L. Monma. “On the Computational Complexity of Integer Programming Problems”. In: *Optimization and operations research*. Ed. by R. Henn. Vol. 157. Lecture Notes in Economics and Mathematical Systems. Berlin and Heidelberg: Springer, 1978, pp. 161–172. ISBN: 978-3-540-08842-4. DOI: 10.1007/978-3-642-95322-4_17.

[27] N. Karmarkar. “A new polynomial-time algorithm for linear programming”. In: *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. Ed. by R. DeMillo. ACM Conferences. New York, NY: ACM, 1984, pp. 302–311. ISBN: 0897911334. DOI: 10.1145/800057.808695.

[28] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. “Multilevel Hypergraph Partitioning: Applications in VLSI Domain”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 7.1 (1999), pp. 69–79. ISSN: 1063-8210. DOI: 10.1109/92.748202.

[29] G. Karypis and V. Kumar. “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs”. In: *SIAM Journal on Scientific Computing* 20.1 (1998), pp. 359–392. ISSN: 1064-8275. DOI: 10.1137/S1064827595287997.

[30] G. Karypis and V. Kumar. “Multilevel k-way Partitioning Scheme for Irregular Graphs”. In: *Journal of Parallel and Distributed Computing* 48.1 (1998), pp. 96–129. ISSN: 07437315. DOI: 10.1006/jpdc.1997.1404.

[31] G. Karypis and V. Kumar. “Parallel Multilevel series k-Way Partitioning Scheme for Irregular Graphs”. In: *SIAM Review* 41.2 (1999), pp. 278–300. ISSN: 0036-1445. DOI: 10.1137/S0036144598334138.

[32] B. W. Kernighan and S. Lin. “An Efficient Heuristic Procedure for Partitioning Graphs”. In: *Bell System Technical Journal* 49.2 (1970), pp. 291–307. ISSN: 00058580. DOI: 10.1002/j.1538-7305.1970.tb01770.x.

[33] J. Leskovec and A. Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. 2014. URL: http://snap.stanford.edu/data.

[34] H. Meyerhenke, P. Sanders, and C. Schulz. “Partitioning Complex Networks via Size-Constrained Clustering”. In: *Experimental Algorithms*. Ed. by D. Hutchison et al. Vol. 8504. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 351–363. ISBN: 978-3-319-07958-5. DOI: 10.1007/978-3-319-07959-2_30.

[35] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. “Partitioning Graphs to Speedup Dijkstra’s Algorithm”. In: *ACM Journal of Experimental Algorithmics* 11 (2007). ISSN: 1084-6654. DOI: 10.1145/1187436.1216585.

[36] V. Osipov and P. Sanders. *n-Level Graph Partitioning*. Apr. 23, 2010. URL: http://arxiv.org/pdf/1004.4024v1.

[37] U. N. Raghavan, R. Albert, and S. Kumara. “Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks”. In: *Physical Review E* 76 (2007). DOI: 10.1103/PhysRevE.76.036106.

[38] L. A. Sanchis. “Multiple-way network partitioning”. In: *IEEE Transactions on Computers* 38.1 (1989), pp. 62–81. ISSN: 00189340. DOI: 10.1109/12.8730.

[39] P. Sanders and C. Schulz. “Engineering Multilevel Graph Partitioning Algorithms”. In: *Algorithms - ESA 2011*. Ed. by C. Demetrescu and M. M. Halldórsson. Vol. 6942. Lecture Notes in Computer Science. Berlin and Heidelberg: Springer, 2011, pp. 469–480. ISBN: 978-3-642-23718-8. DOI: 10.1007/978-3-642-23719-5_40.

[40] P. Sanders and C. Schulz. “Think Locally, Act Globally: Highly Balanced Graph Partitioning”. In: *Experimental Algorithms*. Ed. by D. Hutchison et al. Vol. 7933. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 164–175. ISBN: 978-3-642-38526-1. DOI: 10.1007/978-3-642-38527-8_16.

[41] S. Schlag, T. Heuer, L. Gottesbüren, Y. Akhremtsev, C. Schulz, and P. Sanders. “High-Quality Hypergraph Partitioning”. In: *ACM Journal of Experimental Algorithmics* (2022). ISSN: 1084-6654. DOI: 10.1145/3529090.

[42] K. Schloegel, G. Karypis, and V. Kumar. “Graph Partitioning for High-Performance Scientific Simulations”. In: *Sourcebook of Parallel Computing*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc, 2003, pp. 491–541. ISBN: 1558608710.

[43] C. Schulz. “High Quality Graph Partitioning”. PhD thesis. Karlsruhe: Karlsruher Institut für Technologie, 2013. DOI: 10.5445/IR/1000035713. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000035713.

[44] H. D. Simon and S.-H. Teng. “How Good is Recursive Bisection?” In: *SIAM Journal on Scientific Computing* 18.5 (1997), pp. 1436–1445. ISSN: 1064-8275. DOI: 10.1137/S1064827593255135.

[45] Top 500. *The List.* 2022. URL: https://www.top500.org/project/introduction/ (visited on 07/13/2022).

[46] J. Ugander and L. Backstrom. “Balanced label propagation for partitioning massive graphs”. In: *Proceedings of the sixth ACM international conference on Web search and data mining - WSDM ’13.* Ed. by S. Leonardi, A. Panconesi, P. Ferragina, and A. Gionis. New York, New York, USA: ACM Press, 2013, p. 507. ISBN: 9781450318693. DOI: 10.1145/2433396.2433461.

[47] University of Milano Laboratory of Web Algorithms. *Datasets.* URL: http://law.di.unimi.it/datasets.php.

[48] N. Viswanathan, C. Alpert, C. Sze, Z. Li, and Y. Wei. “The DAC 2012 Routability-Driven Placement Contest and Benchmark Suite”. In: *Proceedings of the 49th Annual Design Automation Conference - DAC ’12.* Ed. by P. Groeneveld, D. Sciuto, and S. Hassoun. New York, USA: ACM Press, 2012, pp. 774–782. ISBN: 9781450311991. DOI: 10.1145/2228360.2228500.

[49] C. Walshaw and M. Cross. “JOSTLE: parallel multilevel graph-partitioning software–an overview”. In: *Mesh partitioning techniques and domain decomposition techniques.* Vol. 10. 2007, pp. 27–58. URL: https://chriswalshaw.co.uk/papers/fulltext/walshawmpdd07.pdf.

[50] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. “Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms”. In: *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing - SC ’07.* Ed. by B. Verastegui. New York, USA: ACM Press, 2007, pp. 1–12. ISBN: 9781595937643. DOI: 10.1145/1362622.1362674.

A Benchmark Set Statistics

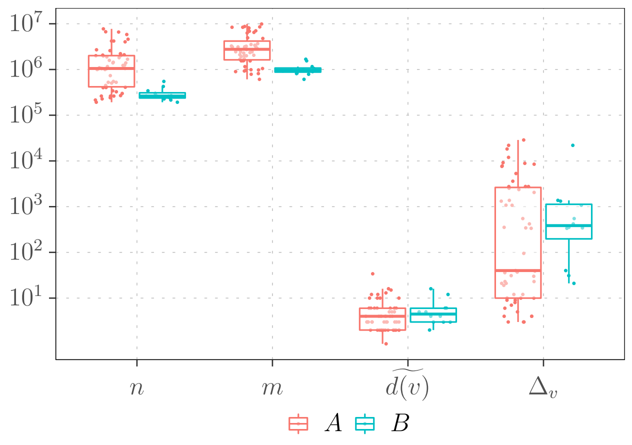


Figure 16: Statistical evaluation of the number of nodes n, number of edges m, median node degree $\widetilde{d(v)}$ and maximum node degree Δ_v in our benchmark sets A and B.

B Detailed Comparison of Different Refinement Algorithms

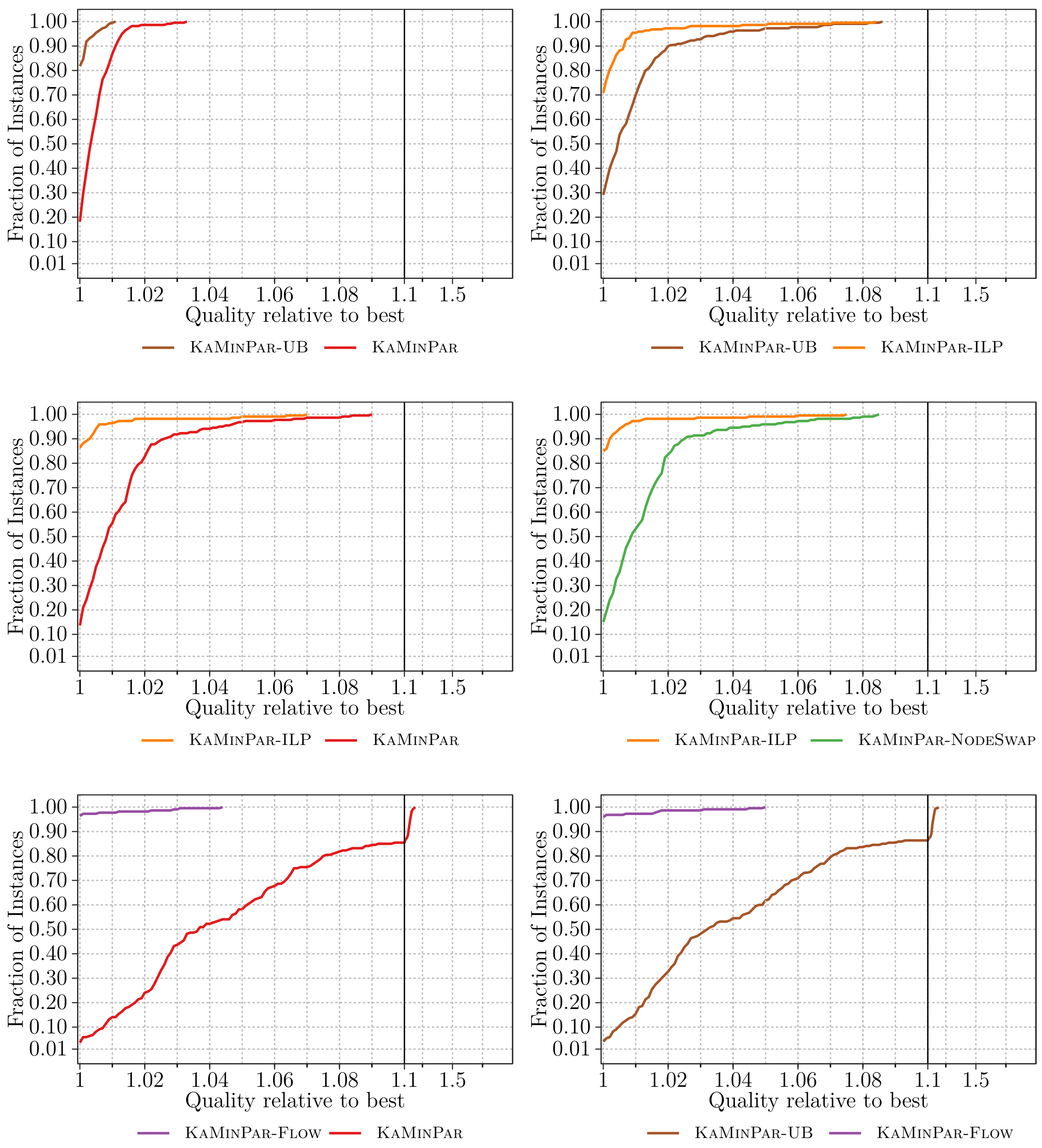


Figure 17: Additional pairwise comparisons of the solution quality for $\varepsilon = 0.01$ on the benchmark set A.

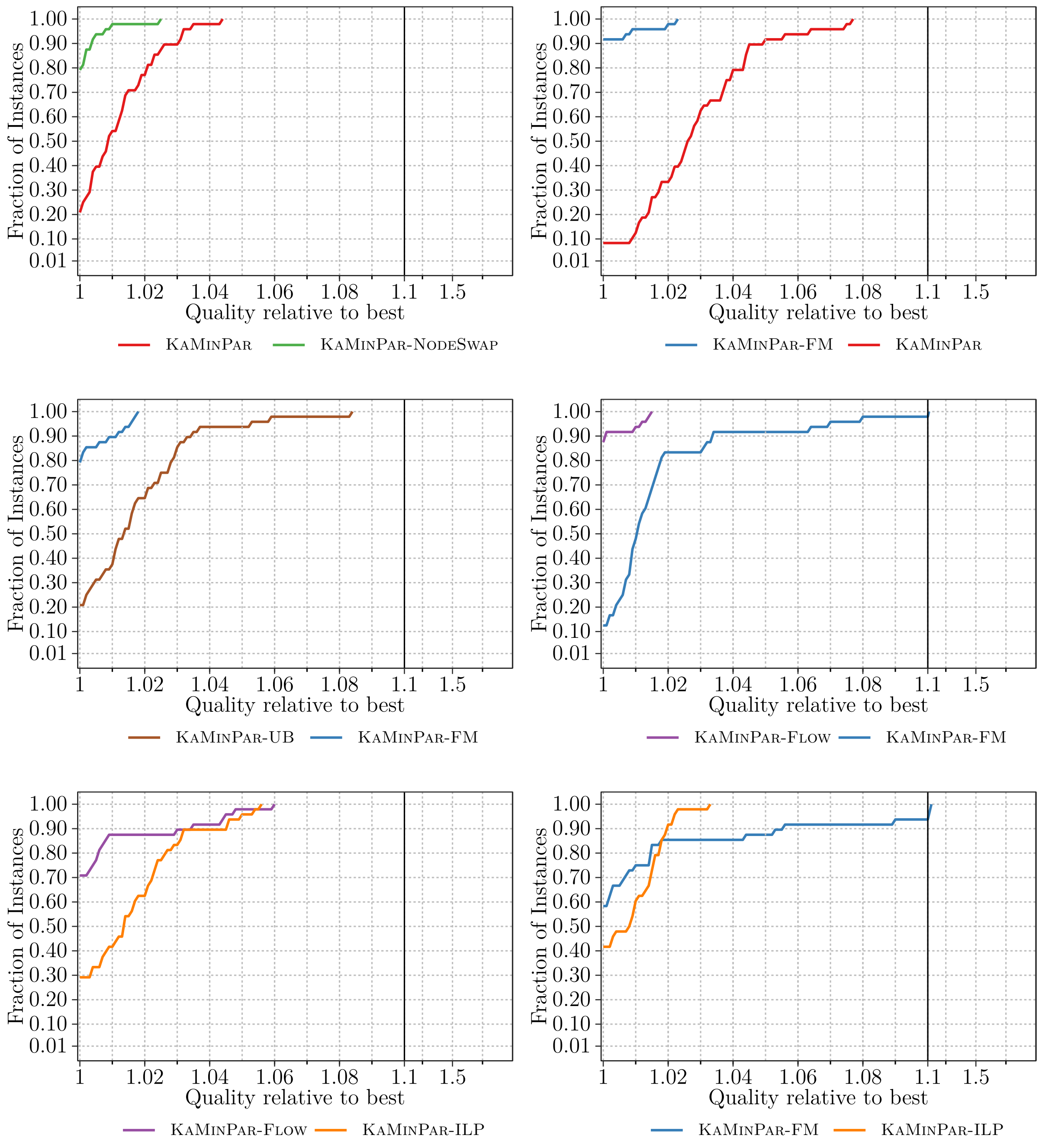


Figure 18: Pairwise comparisons of the solution quality of all algorithms on the benchmark set A for a stricter balance constraint, i.e., $\varepsilon = 0.001$.