Karlsruhe Institute of Technology

Master-Thesis

High Quality Hypergraph Partitioning via Max-Flow-Min-Cut Computations

Tobias Heuer

Advisors: Prof. Dr. Peter Sanders
M. Sc. Sebastian Schlag

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

Date of submission: 22.12.2017

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 22.12.2017

Abstract

In this thesis, we introduce an framework based on *Max-Flow-Min-Cut* computations for improving a balanced direct k-way partition of a hypergraph. Currently, *local search* algorithms based on the *FM* idea [17] are used in all state-of-the-art multilevel hypergraph partitioner. Such *move*-based heuristics have the disadvantage that they only incorporate *local* informations about the problem structure and in case of ties, the quality mainly depends on random choices made within the algorithm [15, 31, 36]. *Flow*-based approaches are not *move*-based and able to find a *global* minimum cut separating to vertices s and t [18]. Therefore, utilizing such techniques as *refinement* heuristic would overcome the drawbacks of the *FM* algorithm.

Our framework is inspired by the work of Sanders and Schulz [45] who successfully integrate a *flow*-based refinement algorithm in a multilevel graph partitioner. We generalize many ideas of them such that they are applicable in the multilevel hypergraph partitioning context. We develop several techniques to sparsify the flow network of a hypergraph, which reduces the resulting problem size on average by a factor of 2 compared to the state-of-the-art representation [33]. Additionally, we show how to configure a flow problem on a subhypergraph such that the quality achievable with a *Max-Flow-Min-Cut* computation is significantly better than with the modeling approach of Sanders and Schulz. Finally, we integrate our work as refinement strategy into the n-level hypergraph partitioner *KaHyPar* [25].

We tested our framework on a large benchmark set with 3222 instances. In comparison to 5 different systems, our new configuration outperforms the tested state-of-the-art hypergraph partitioner on 70% of the instances. In comparison to the latest version of *KaHyPar*, our new approach produces 2.4% better quality while only incurring a performance slowdown by a factor of 2. Moreover, our partitioner has a comparable running time to the direct k-way version of *hMetis* and outperforms it on 82% of the benchmarks.

Acknowledgements

A good mentor is important for staying motivated and to have the feeling to work on something meaningful. Since my bachelor thesis, I work together with Sebastian Schlag in the research area of *Hypergraph Partitioning*. The decision to further work on this topic was not only a matter of interest, it was mainly a decision based on our outstanding interpersonal working relationship. Our intellectual discussions were characterized by the right balance of fun and the necessary seriousness to work towards a common goal. I think here is the right place to thank you for the endless time, which I have spent in your office over the last three years and which have made me a better computer scientist.
Further, I would like to thank my girl friend Alessa Dreixler. To be together with a computer scientist could be sometimes very complicated and exhausting. Especially, if anger and frustration dominated my working day. With her understanding and motivation, I could continue every morning with the same energy and passion as the day before.

Contents

1. Introduction

Hypergraphs are a generalization of graphs, where each (hyper)edge can connect more than two (hyper)nodes. The k-way hypergraph partitioning problem is to partition the vertices of a hypergraph into k disjoint, non-empty blocks such that the size of each block is smaller than $1 + \epsilon$ times the average block size, while the goal is to simultaneously minimize an objective function.
Classical application areas can be found in *VLSI* design, parallelization of the Sparse Matrix-Vector Product and simplifying *SAT* formulas [29, 36, 40]. The goal in *VLSI* design is to partition a circuit into smaller units such that the wires between the gates are as short as possible [9]. A wire can connect more than two gates, therefore a hypergraph models a circuit more accurately than a graph. In *SAT* solving, hypergraph partitioning is used to decompose a formula into smaller subformulas, which can be solved more easily [36].
Hypergraph partitioning is an NP-hard problem [34] and it is even NP-hard to find a good approximation [8]. The most common heuristic used in state-of-the-art hypergraph partitioners is the *multilevel paradigm* [10, 25, 29]. First, a sequence of smaller hypergraphs is generated by contracting a set of matchings between hypernode pairs or clusters in each step (*coarsening phase*). If the hypergraph is small enough, we can use expensive heuristics to *initial partition* it into k blocks. Afterwards, the sequence of smaller hypergraphs is *uncontracted* in reverse order and, at each level, a *local search* heuristic is used to improve the quality of the partition (*refinement phase*).
There exist several *local search* heuristics for improving hypergraph partitions. One algorithm used in the state-of-the-art multilevel hypergraph partitioners is the *Fiduccia-Mattheyses* heuristic (FM). The *FM* algorithm maintains gain values (according to a objective function) of moving a node from its current block to another block [17]. A move is performed, if its gain value is maximum among all possible moves. The *FM* heuristic is generally intuitive, flexible in adapting to different optimization objectives, easy to implement and relatively fast [52].
However, the gain of moving a (hyper)node to another block only depends on the state of the incident (hyper)edges. Therefore, *FM* has no *global* view on the structure of the problem. A move is performed *locally* and *greedily*. Consequently, the algorithm tends to find locally optimal solutions, which quality heavily depends on the initial partition [15]. Therefore, multiple runs are needed to find a solution close to the global optimum. The probability of finding a good approximation of the global optimum significantly drops if we partition large hypergraphs [15]. If we execute *FM* in the multilevel context, we partially solve the problem. A move of an vertex in a *coarsened* hypergraph corresponds to a movement of a subset of the hypernodes on the *original* hypergraph which allows a more effective exploration of the solution space [40]. The quality of the solution then depends more on the quality of the *coarsening* rather than on the *initial partition*.
Further, a move of a node only influences the gain function if the state of an incident hyperedge changes *immediately* after a move. If a hyperedge contains vertices from two different blocks, where only one hypernode is contained in the first and all remainings are in the second block, then a move of that node contributes to the gain if the objective is e.g., *cut* (sum of the weights of hyperedges which contains vertices of more than one block). Especially for large hyperedges, we often have to *move* a sequence of nodes such that a single move of a node finally contributes to the gain. Therefore, the gain of most vertices is equal to zero in such cases [36]. Krishnamurthy [31] points out that the quality in such situations highly depends on random choices made within the algorithm. Therefore, he enhanced the *FM* algorithm with a look-ahead scheme such that in case of ties one can incoroperate *future gains* into the decision [31]. However, the *forecast* is limited by a predefined parameter.

FM-based *local search* algorithms have the above-mentioned disadvantages, because they are *move*-based and only incoroperate *local* informations about the structure of the problem. Finding a balanced global minimum cut of a (hyper)graph is NP-hard, but if we ask for a minimum cut separating two vertices s and t the problem becomes solvable in polynomial time [16]. The well-known *max-flow min-cut* theorem [18] establish an analogy between the maximum flow from a source s to a sink t and the minimum cut separating s and t in a graph. *Flow*-based approaches are not *move*-based and incoroperate the *global* structure of the problem. Therefore, it overcomes the drawbacks of the *FM* algorithm. However, it was overlooked for a long time because it was perceived as computationally expensive and impractical for (hyper)graph partitioning [35].
Sanders and Schulz [45] successfully integrated a *flow*-based refinement algorithm in their multilevel *graph* partitioner. They combine the strength of *flow*-based and *FM local search* by executing both algorithms alternating throughout the multilevel hierarchy. As a result their multilevel graph partitioner produces the best partitions for a wide range of graph partitioning benchmarks. Recently, several algorithms were developed to obtain a balanced bipartition of a hypergraph with *Max-Flow-Min-Cut* computations [35, 41, 51]. A balanced k-way hypergraph partition with such an approach is currently only obtainable by applying the bipartitioning algorithm recursively [51]. The impact of a *flow*-based *local search* algorithm on the solution quality of a multilevel hypergraph partitioner has not been studied yet.

1.1. Problem Statement

Motivated by the successfull integration of a *flow*-based *local search* algorithm in the multilevel graph partitioner *KaFFPa* of Sanders and Schulz [45] to obtain balanced k-way partitions, this thesis investigates the integration of such an approach into the multilevel hypergraph partitioner *KaHyPar* [25].
In the first step, we have to find an appropriate model of a hypergraph as flow network. Afterwards, we want to improve a given bipartition of a hypergraph with a *Max-Flow-Min-Cut* computation by using the flow network of the previous step. This work is the theorectical foundations for developing a *flow*-based *local search* algorithm which works in a multilevel hypergraph partitioner and improves a given balanced k-way partition. The last step is to integrate the framework into the n-level hypergraph partitioner *KaHyPar* [25] and evaluate the performance on a large benchmark set in comparison to different state-of-the-art multilevel hypergraph partitioners. A major goal of this work is to outperform the latest version of *KaHyPar* on most of the benchmark instances and simultaneously ensure that the running time is only within a constant factor slower.

1.2. Contributions

We present several techniques to sparsify the state-of-the-art hypergraph flow network modeling approach proposed by Lawler [33]. Our experiments indicate that maximum flow algorithms are up to a factor of 3 faster with our new network. The theorectical results, which leads to the presented sparsification techniques, are of independent interest and can also be applied on general flow networks. Further, we show how to configure a flow problem on a subhypergraph of H such that a maximum (S, T)-flow yields an improved cut of a given bipartition. We choose S and T in a way such that the value of the cut of H after a *Max-Flow-Min-Cut* computation on a subhypergraph can be calculated with the value of a maximum (S, T)-flow. Our *flow*-based *local search* framework is inspired by algorithmic ideas of Sanders and Schulz [45]. However, we generalize many results of their work such that they are applicable on hypergraph partitioning.

Further, we implement several heuristics which might prevent unpromising *Max-Flow-Min-Cut* computations throughout the multilevel hierarchy and show that they speed-up the framework by factor of 2 while maintaining the quality of the solutions on average. We integrate our *flow*-based *local search* algorithm into the n-level hypergraph partitioner *KaHyPar* and show that *flow*-based refinement in combination with the *FM* algorithm produces the best partitions on a majority of real world benchmarks in comparison to other state-of-the-art hypergraph partitioners. Ccompared to 5 different systems we achieve the best partitions on 70% of 3222 benchmark instances. In comparison to the latest version of *KaHyPar*, our new approach produces solutions that are 2% better on average, while only incurring a slowdown by a factor of 2. Moreover, our partitioner has a comparable running time to the direct k-way version of *hMetis* and outperforms it on 82% of the benchmark instances

1.3. Outline

We first introduce necessary notations and summarize related work in Sections 2 and 3. Afterwards, we describe techniques to sparsify the flow network proposed by Lawler [33] in Section 4. In Section 5 we present our source and sink set modeling approach and describe the integration of our *flow*-based refinement framework into the n-level hypergraph partitioner *KaHyPar*. The experimental evaluation of our algorithm is presented in Section 6. Section 7 concludes this thesis.

2. Preliminaries

2.1. Graphs

Definition 2.1. *A directed weighted graph $G = (V, E, c, \omega)$ is a set of nodes V and a set of edges E with a node weight function $c : V \to \mathbb{R}_{\geq 0}$ and an edge weight function $\omega : E \to \mathbb{R}_{\geq 0}$. An edge $e = (u, v)$ is a relation between two nodes $u, v \in V$.*

Two vertices u and v are *adjacent*, if there exists an edge $(u, v) \in E$. Two edges e_1 and e_2 are *incident* to each other if they share a node. $N(v)$ denotes the set of all *adjacent* nodes of v. The *degree* of a node v is $d(v) = |N(v)|$.

Definition 2.2. *Given a directed graph $G = (V, E)$. A contraction of two nodes u and v results in a new graph $G_{(u,v)} = (V \setminus \{v\}, E')$, where each edge of the form (v, w) or (w, v) in E is replaced with an edge (u, w) or (w, u) in E'.*

A *path* $P = (v_1, \ldots, v_k)$ is a sequence of nodes, where for each $i \in [1, k-1] : (v_i, v_{i+1}) \in E$. A *cycle* is a *path* $P = (v_1, \ldots, v_k)$ with $v_1 = v_k$. A *strongly connected component* $C \subseteq V$ is a set of nodes where for each $u, v \in C$ exists a *path* from u to v. We can enumerate all *strongly connected components* (*SCC*) in a directed graph G with a linear time algorithm proposed by Tarjan [48]. A directed graph G without any *cycles* is called *directed acyclic graph* (*DAG*). On such graphs we can define a *topological order* $\gamma : V \to \mathbb{N}_+$ such that for each $(u, v) \in E : \gamma(u) < \gamma(v)$. A *topological order* of a *DAG* can be found in linear time with Kahn's algorithm [28]. We can transform a general directed graph G into a *DAG* if we contract each *strongly connected component*. All concepts are illustrated in Fig. 1.

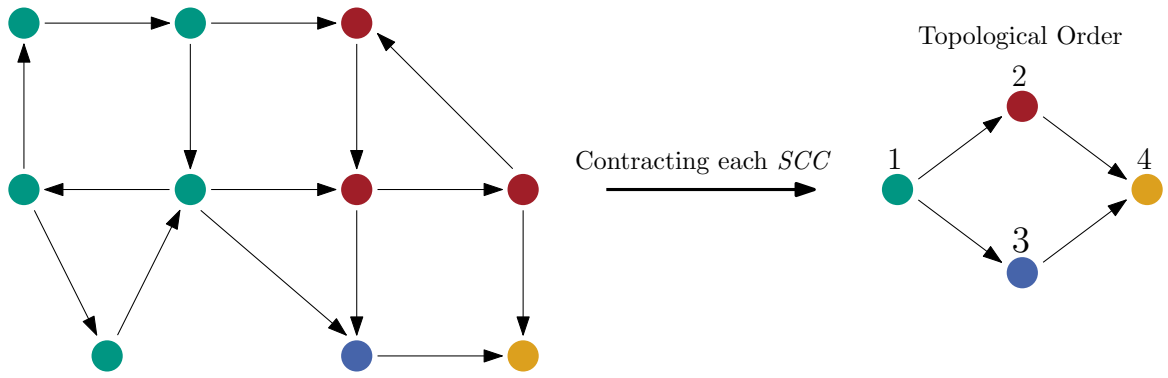

Figure 1: Example of *Strongly Connected Components* in a directed graph and a *Topological Order* of a *Directed Acyclic Graph*. Each *SCC* is marked with the same color.

Definition 2.3. *Let $G_{V'} = (V', E_{V'}, c, \omega)$ be the subgraph of a graph G induced by $V' \subseteq V$ with $E_{V'} = \{(u, v) \in E \mid u, v \in V'\}$.*

2.2. Flows and Applications

Given a graph $G = (V, E, u)$ with capacity function $u : E \to \mathbb{R}_+$ and a source $s \in V$ and a sink $t \in V$, the maximum flow problem is to find the maximum amount of flow from s to t in G. A flow is a function $f : E \to \mathbb{R}_+$, which have to satisfy the following constraints:

(i) $\forall (u, v) \in E : f(u, v) \leq u(u, v)$ (capacity constraint)

(ii) $\forall v \in V \setminus \{s, t\} : \sum_{(u,v)\in E} f(u, v) = \sum_{(v,u)\in E} f(v, u)$ (conservation of flow constraint)

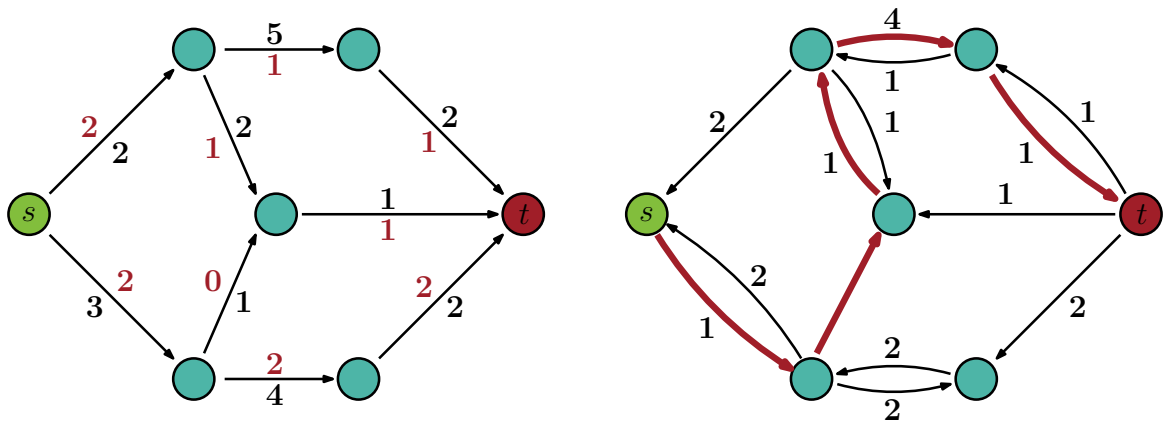

Figure 2: Illustrates concepts related to the maximum flow problem. A flow function f (red values) from s to t of a graph G is shown on the left side. The corresponding *residual graph* G_f with its *residual capacities* (black values) is illustrated on the right side. The red highlighted path is an *augmenting path*.

The capacity constraint restricts the flow on edge (u, v) by its capacity $u(u, v)$. Whereas the conservation of flow constraint ensures that the amount of flow entering a node $v \in V \setminus \{s, t\}$ is the same as leaving a node. The value of the flow is defined as $|f| = \sum_{(s,v)\in E} f(s, v) = \sum_{(v,t)\in E} f(v, t)$. A flow f is maximal, if there exists no other flow f' with $|f'| > |f|$.
Further, we define the *residual graph* G_f and the *residual capacity* r_f of a flow function f on graph G. The *residual capacity* $r_f : V \times V \to \mathbb{R}_+$ is defined as follows:

(i) $\forall (u, v) \in E : r_f(u, v) = u(u, v) - f(u, v)$

(ii) $\forall (u, v) \in E$: If $f(u, v) > 0$ and $u(v, u) = 0$, then $r_f(v, u) = f(u, v)$

For an edge $e = (u, v) \in E$ the residual capacity $r_f(u, v)$ is the remaining amount of flow which can be send over edge e. For each reverse edge $\overleftarrow{e} \notin E$ the residual capacity $r_f(\overleftarrow{e})$ is the amount of flow which is send over e. The *residual graph* $G_f = (V, E_f, r_f)$ is the network containing all $(u, v) \in V \times V$ with $r_f(u, v) > 0$. More formally $E_f = \{(u, v) \in V \times V \mid r_f(u, v) > 0\}$. An *augmenting path* $P = \{v_1, \ldots, v_k\}$ is a path in G_f with $v_1 = s$ and $v_k = t$ [16]. Fig. 2 illustrates all presented concepts.

The *Max-Flow-Min-Cut*-Theorem is fundamental for many applications related to the maximum flow problem [18].

Theorem 2.1. *The value of a maximum (s, t)-flow obtainable in a graph G is equal with the minimum-weight cutset in G separating s and t.*

Let f be a maximum (s, t)-flow in a graph $G = (V, E, \omega)$ with $s \in V$ and $t \in V$. Further, let A be the set containing all $v \in V$, which are *reachable* from s in G_f. A node v is *reachable* from a node u if there exists a path from u to v. Then the set of all cut edges between the bipartition $(A, V \setminus A)$ is a minimum-weight (s, t)-cutset [19]. A can be calculated with a *BFS* in G_f starting from s.
We can solve with maximum flows many related problems like e.g., maximum bipartite-matching, number of edge- or vertex-disjoint paths in a graph or to find a minimum-weight vertex separator. Solutions for those problems sometimes involve a transformation T of the graph G into a flow network $T(G)$, such that the *Max-Flow-Min-Cut*-Theorem is applicable. A problem essential for this work is to find a minimum-weight (s, t)-vertex separator in a graph $G = (V, E, c)$ with $c : V \to \mathbb{R}_+$.

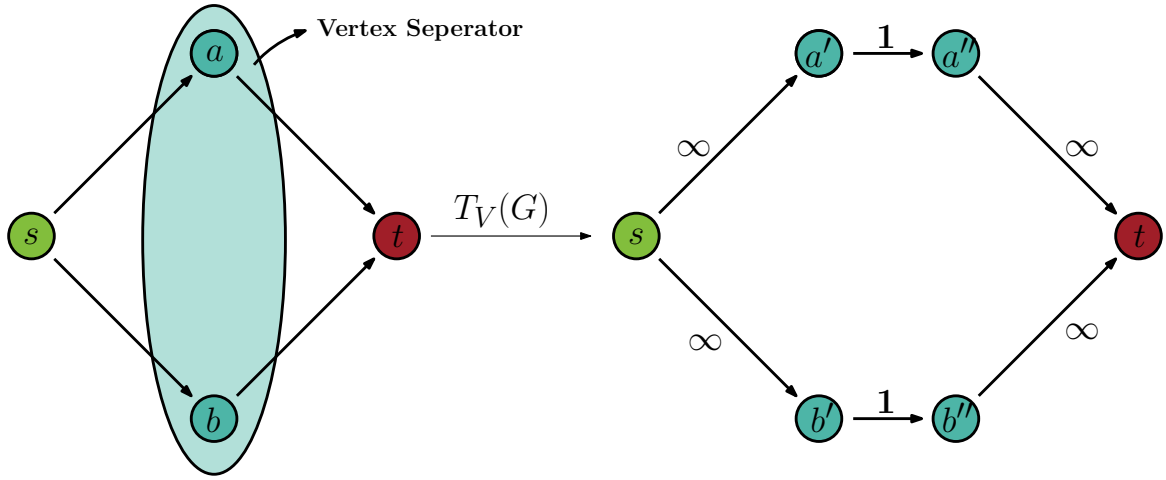

Figure 3: Illustration of the vertex separator problem and the flow network $T_V(G)$ in which we can find a minimum vertex separator.

Definition 2.4. *Let $G = (V, E, c)$ be a graph with $c : V \to \mathbb{R}_+$. $S \subseteq V$ is a vertex separator for non-adjacent vertices $s \in V$ and $t \in V$ if the removal of S from graph G separates s and t (s not* reachable *from t). A vertex separator S is a minimum-weight (s,t)-vertex separator, if for all (s,t)-vertex separators $S' \subseteq V$ follows that $c(S) \leq c(S')$.*

We can calculate a minimum-weight (s,t)-vertex separator with a maximum flow calculation on the following flow network [50]:

Definition 2.5. *Let T_V be a transformation of a graph $G = (V, E, c)$ into a flow network $T_V(G) = (V_V, E_V, u_V)$ (with $u_V : E_V \to \mathbb{R}_+$). T_V is defined as follows:*

(i) $V_V = \bigcup_{v \in V} \{v', v''\}$

(ii) $\forall v \in V$ *add a directed edge* (v', v'') *with capacity* $u_V(v', v'') = c(v)$

(iii) $\forall (u, v) \in E$ *add two directed edges* (u'', v') *and* (v'', u') *with capacity* $u_V(u'', v') = u_V(v'', u') = \infty$.

The vertex separator problem and transformation $T_V(G)$ is illustrated in Fig. 3. Obviously, no edge between two adjacent nodes in G can be in a minimum-capacity (s,t)-cutset of $T_V(G)$, because for all those edges the capacity is ∞. Therefore, the cutset must consist of edges of the form (v', v''). A minimum-weight (s,t)-vertex separator can be calculated by finding a maximum (s,t)-flow of $T_V(G)$ and the corresponding minimum (s,t)-cutset [37].
Given a set of sources S and sinks T. The *multi-source multi-sink* maximum flow problem is to find a maximum flow f from all source nodes $s \in S$ to all sink nodes $t \in T$. We can transform such a problem into a *single-source single-sink* problem by adding two additional nodes s and t. We add a directed edge from s to all source nodes $s' \in S$ and for all sink nodes $t' \in T$ a directed edge to t with capacity $u(s, s') = u(t', t) = \infty$.

2.3. Hypergraphs

Definition 2.6. *An undirected weighted hypergraph $H = (V, E, c, \omega)$ is a set of hypernodes V and a set of hyperedges E with a hypernode weight function $c : V \to \mathbb{R}_{\geq 0}$ and a hyperedge weight function $\omega : E \to \mathbb{R}_{\geq 0}$. A hyperedge e is a subset of V (formally: $\forall e \in E : e \subseteq V$).*

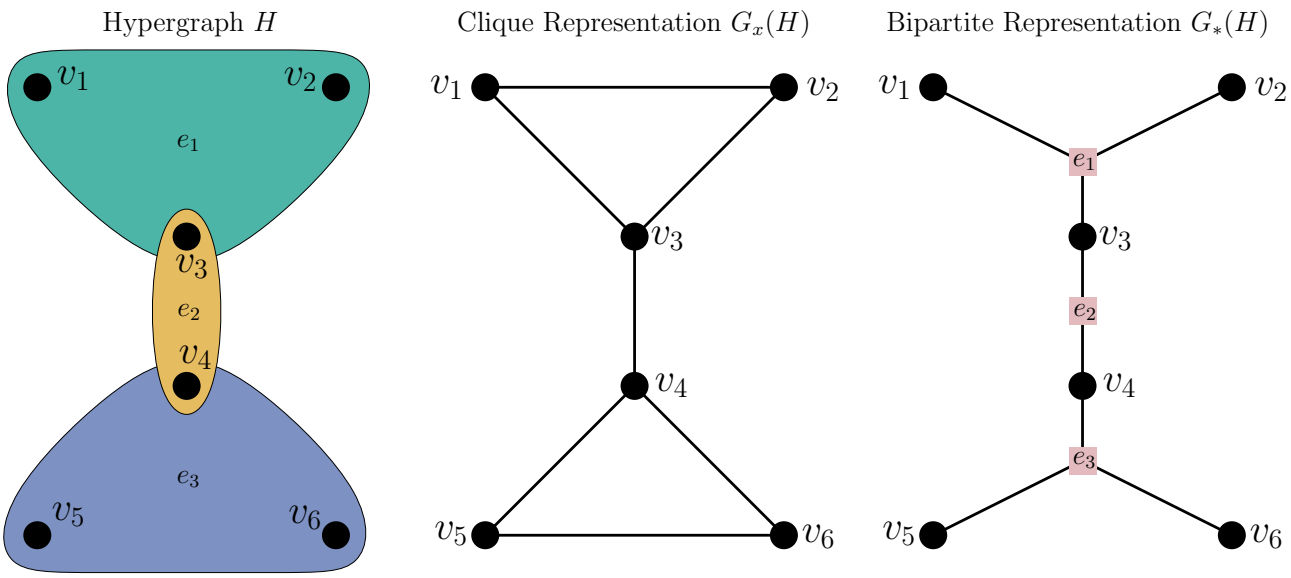

Figure 4: Example of a hypergraph H and its two corresponding graph representations.

A hypergraph generalizes a graph by extending the definition of an edge, which can contain more than two nodes. Hyperedges are also called *nets* and hypernodes are also called *vertices*. A vertex contained in a net is called *pin*. For a subset $V' \subseteq V$ and $E' \subseteq E$ we define

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

A vertex v is *incident* to a hyperedge e if $v \in e$. Two vertices u and v are *adjacent*, if there exists an $e \in E$ such that $u \in e$ and $v \in e$. $I(v)$ denotes the set of all *incident* nets of v. The *degree* of a hypernode v is $d(v) = |I(v)|$. The size of a net e is its cardinality $|e|$.

Definition 2.7. *Let $H_{V'} = (V', E_{V'}, c, \omega)$ be the subhypergraph of a hypergraph H induced by $V' \subseteq V$ with $E_{V'} = \{e \cap V' \mid e \in E : e \cap V' \neq \emptyset\}$.*

Definition 2.8. *Given a subset $A \subseteq V$ of hypergraph $H = (V, E, c, \omega)$. The section hypergraph $H \times A$ is the hypergraph induced by A containing all hyperedges $e \in E$ which are a subset of A. More formally, $H \times A = (A, \{e \mid e \in E : e \subseteq A\}, \omega, c)$.*

A hypergraph $H = (V, E, c, \omega)$ can be represented as an undirected graph. There are two standard transformations, called *clique* and *bipartite* representation [27]. The *clique* graph $G_x(H) = (V, E_x)$ models each net e as a clique between its pins. The *bipartite* graph $G_*(H) = (V \cup E, E_*)$ contains all hypernodes and hyperedges as nodes and connects each net e with an undirected edge $\{e, v\}$ to all its pins $v \in e$. The two transformations are illustrated in Fig. 4.

2.4. Hypergraph Partitioning

Definition 2.9. *A k-way partition of a hypergraph H is a partition of its hypernodes into k disjoint blocks $\Pi = \{V_1, \ldots, V_k\}$ such that $\bigcup_{i=1}^{k} V_i = V$ and $V_i \neq \emptyset$.*

For a k-way partition $\Pi = \{V_1, \ldots, V_k\}$, we define the *connectivity set* of a hyperedge e with $\Lambda(e, \Pi) = \{V_i \in \Pi \mid V_i \cap e \neq \emptyset\}$. The *connectivity* of a net e is $\lambda(e, \Pi) = |\Lambda(e, \Pi)|$. A hyperedge

e is *cut*, if $\lambda(e, \Pi) > 1$. $E(\Pi) = \{e \mid \lambda(e, \Pi) > 1\}$ is the set of all *cut* nets. We say two blocks V_i and V_j are adjacent, if there exists a hyperedge e with $V_i \in \Lambda(e, \Pi)$ and $V_j \in \Lambda(e, \Pi)$. A k-way partition is ϵ-balanced if each block $V_i \in \Pi$ satisfies the *balance constraint* $c(V_i) \leq (1+\epsilon)\lceil \frac{c(V)}{k} \rceil$.

Definition 2.10. *The k-way hypergraph partitioning problem is to find an ϵ-balanced k-way partition Π of a hypergraph H such that a certain objective function is minimized.*

There exists several objective functions in the hypergraph partitioning context. The most popular objective function is the cut metric (especially for *graph partitioning*), which is defined as

$$\omega_H(\Pi) = \sum_{e \in E(\Pi)} \omega(e)$$

The goal is to minimize the weight of all *cut* hyperedges. Another important metric for this work is the $(\lambda - 1)$-metric or *connectivity* metric, which is defined as

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

The idea behind this function is to minimize the *connectivity* of all hyperedges.

Definition 2.11. *We define for a k-way partition $\Pi = \{V_1, \ldots, V_k\}$ of a hypergraph H the quotient graph $Q = (\Pi, E')$ which contains an edge between each pair of adjacent blocks of Π. More formally, $E' = \{(V_i, V_j) \mid \exists e \in E : V_i, V_j \in \Lambda(e, \Pi)\}$*

3. Related Work

3.1. Maximum Flow Algorithms

In Section 2.2 we introduce the concept of flows in a network. We will now present two approaches to solve the maximum flow problem.

3.1.1. Augmenting-Path Algorithms

An *augmenting path* $P = \{v_1, \dots, v_k\}$ is a path in G_f with $v_1 = s$ and $v_k = t$ [16]. Fig. 5 illustrates such a path. Since all $(v_i, v_{i+1}) \in G_f$ it follows that $r_f(v_i, v_{i+1}) > 0$. Therefore, we can increase the flow on all edges (v_i, v_{i+1}) by $\Delta f = \min_{i \in [1,\dots,k-1]} r_f(v_i, v_{i+1})$. It can be shown that f is not a maximum flow if an *augmenting path* exists in G_f [16].
One way to calculate a maximum flow f is to find *augmenting paths* in G_f as long as there exists one. The algorithm was established by Ford and Fulkerson [18] and consists of two phases. First, we search for an *augmenting path* $P = \{v_1, \dots, v_k\}$ from s to t, e.g., with a simple *DFS*. Afterwards, we increase the flow on each edge (v_i, v_{i+1}) by Δf and decrease the flow on each reverse edge (v_{i+1}, v_i) by Δf. If the capacities are integral, the algorithm always terminates. Since we can find an *augmenting path* in G_f with a simple *DFS* in $\mathcal{O}(|V| + |E|)$ and increase the flow on every path by at least one, the running time of the algorithm can be bounded by $\mathcal{O}(|E||f_{max}|)$. We can construct instances, where the running time is $\mathcal{O}(|E||f_{max}|)$ or even the maximum flow $|f_{max}|$ is exponential in the problem size [16].
Edmond and Karp [16] improved Ford & Fulkerson's algorithm by increasing the flow along an *augmenting path* of minimal length. The shortest path from s to t in a graph with unit lengths can be found with a simple *BFS*. It can be shown, that the total number of *augmentations* is $\mathcal{O}(|V||E|)$. The running time of Edmond & Karp's maximum flow algorithm is $\mathcal{O}(|V||E|^2)$. A sample execution of the algorithm is presented in Fig. 5.
Boykov and Kolmogorov proposed a maximum flow algorithm based on *augmenting path* especially designed for applications in computer vision [7]. Their basic idea is to grow two search trees simultanously. One is starting from the source and one from the sink. The two search trees maintains the invariant that all edges in the tree are non-saturated. More formally, for an edge e the residual capacity $r_f(e)$ must be greater than zero. A node is added to one of the two trees if a non-saturated edge exists connecting the node with one of the nodes in the search trees. If the two trees touch at a given node, we found an *augmenting path* from the source to the sink. After we increase the flow along this path, some of the edges in the two search trees are saturated. Therefore, the algorithm tries to restore the search tree invariant by finding a new non-saturated edge for each node which is connected through a saturated edge to the tree. If its not possible, then the node is removed from the tree. The algorithm has no polynomial complexity (worst case $\mathcal{O}(|E||V|^2|f|)$), but it outperforms many state-of-the-art maximum flow algorithms on computer vision benchmarks [7].
An extension of the algorithm of Boykov & Kolmogorov is the *incremental breadth-first search* algorithm [21], which guarantes polynomial running time ($\mathcal{O}(|V|^2|E|)$). The algorithm maintains two distance labels d_s and d_t for each node. For some values D_s and D_t, the source tree contains all nodes up to a distance D_s and the sink tree up to distance D_t. Also they maintain the invariant that $L = D_s + D_t + 1$ is an lower bound for the shortest *augmenting path*. Initially, $d_s(s) = d_t(t) = 0$ and $D_s = D_t = 0$. The algorithm works in passes and in a pass one of the two trees is choosen to grow. Assume, we have choosen the source tree. Each node u contained in the source tree with distance label $d_s(u) = D_s$ is marked as *active*. In a pass all *active* nodes are processed. If a *active* node u is adjacent to a node v not contained in any of the two trees,

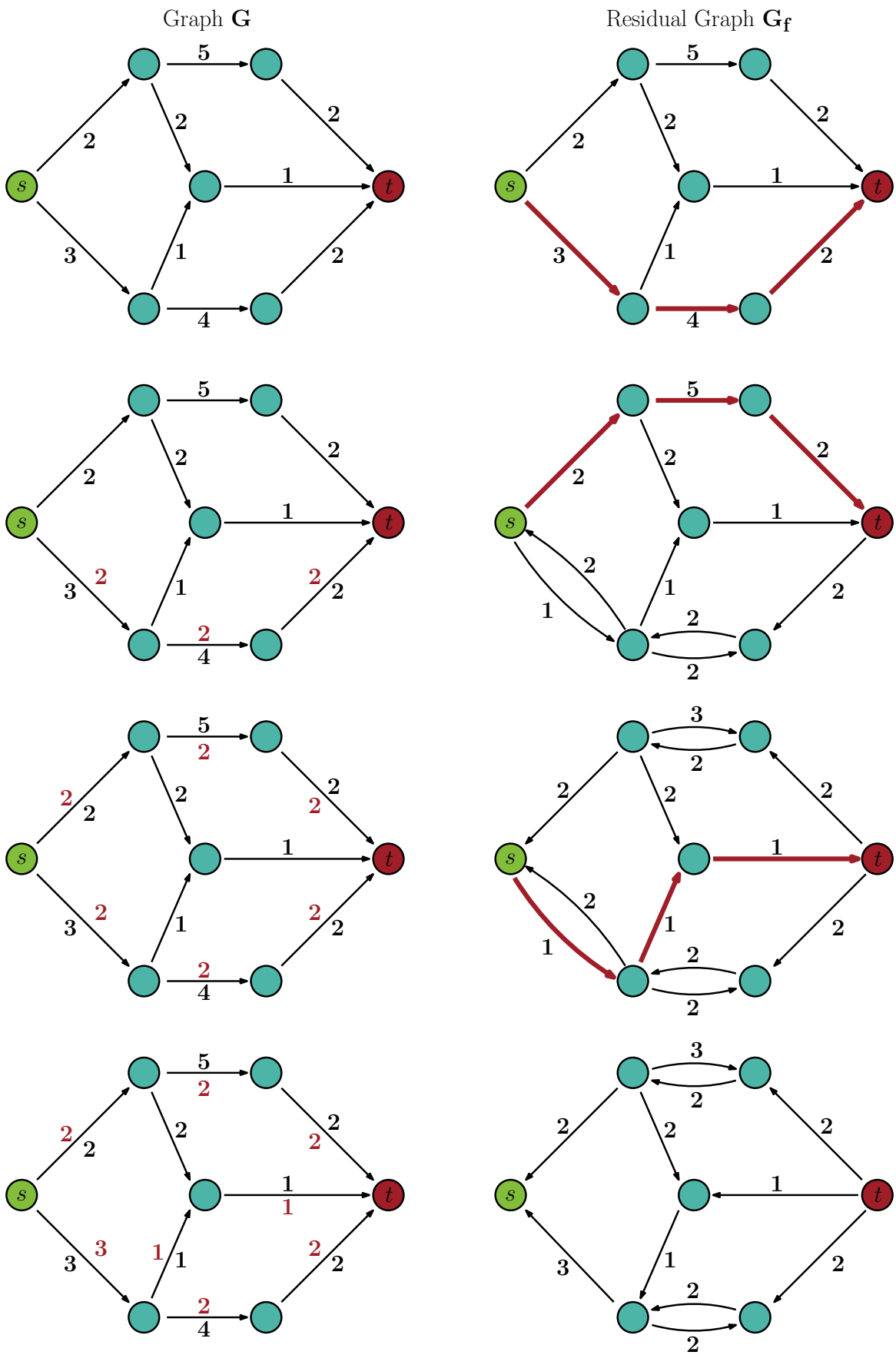

Figure 5: Execution of Edmond & Karps maximum flow algorithm [16]. The network G with its capacities c (black values) and flow f (red values) is illustrated on the left side. The residual graph G_f with its *residual capacities* r_f (black values) is presented on the right side. In each step the current *augmenting path* in G_f is highlighted by a red path.

we add v to the source tree and set $d_s(v) = d_s(u) + 1$. If v is in the sink tree, we have found an *augmenting path*. After *augmenting* along that path some of the nodes are not connected to the tree through a non-saturated edge. For such a node v the adjacency list is scanned and if an adjacent node u exist with $d_s(u) = d_s(v) - 1$ and $r_f(u, v) > 0$, the parent of v is set to u. If such a node is not found, we search for an adjacent node u for which $d_s(u)$ is minimal and $r_f(u, v) > 0$. If such a node is found, we set the parent of v to u and $d_s(v) = d_s(u) + 1$. Otherwise, v is removed from the source tree. If after a pass a node v exists with $d_s(v) = D_s + 1$, D_s is set to $D_s + 1$, otherwise the algorithm terminates.

3.1.2. Push-Relabel Algorithm

Goldberg and Tarjan [23] implemented a maximum flow algorithm not based on finding an *augmenting path* in the *residual graph*. The idea is to maintain a *preflow* during the execution of the algorithm which satisfies the capacity constraints, but only a weakened form of the conservation of flow constraint:

$$\forall v \in V \setminus \{s, t\} : \sum_{u \in V} f(v, u) \leq \sum_{u \in V} f(u, v)$$

The algorithm maintains a *distance labeling* $d : V \to \mathbb{N}$ and an *excess function* $e_f : V \to \mathbb{N}$. The *distance labeling* satisfies the following conditions: $d(s) = |V|$, $d(t) = 0$ and for each $(u, v) \in E_f$, $d(u) \leq d(v) + 1$. We say an residual edge (u, v) is *admissible* if $d(u) = d(v) + 1$. A node v is *active* if $v \notin \{s, t\}$ and $e_f(v) > 0$.
Initially, all *labels* and *excess* values are set to zero except source node s will be set to $d(s) = 1$ and $e_f(s) = \infty$. For each *active* node u the algorithm performs two update operations, called *push* and *relabel*. The first operation pushes flow over each *admissible* edge (u, v). After a *push* $e_f(u) = e_f(u) - \min(e_f(u), r_f(u, v))$ and $e_f(v) = e_f(v) + \min(e_f(u), r_f(u, v))$. If there is no *admissible* edge, a *relabel* operation is performed, which replaces $d(u)$ by $\min_{(u,v) \in E_f} d(v) + 1$. The algorithm terminates, if none of the nodes is *active*. The worst case complexity of the algorithm is $\mathcal{O}(n^3)$. The running time can be reduced to $\mathcal{O}(n^2 \log n)$ with *Dynamic Trees* [23, 47], but this implementation is not practical due to a large hidden constant factor.
The *push-relabel* algorithm is one of the fastest maximum flow algorithms in practice because there exist several speed-up techniques. The first one is the *global relabeling* heuristic which frequently updates the *distance labels* by computing the shortest path in the residual graph from all nodes to the sink [12]. This can be done with a backward *BFS* in linear time. This technique is performed periodically, e.g., after every n relabeling.
The second heuristic is the *gap heuristic* [11, 14]. If at a particular stage of the algorithm there is no node u with $d(u) = g < n$, then for each node v with $g < d(v) < n$ the sink is not reachable anymore. Therefore, we can increase the *distance label* of all those nodes to n. To implement this heuristic, we maintain a linked list of nodes with distance label i.

3.2. Modeling Flows on Hypergraphs

Consider the *bipartite graph* representation $G_*(H)$ of a hypergraph H (see Section 2.3). Hu and Moerder [27] introduce node capacities in $G_*(H)$. Each hyperedge e has a capacity equal to $\omega(e)$ and each hypernode has infinite capacity. Further, they show that a minimum-weight (s, t)-vertex separator in $G_*(H)$ is equal with a minimum-weight (s, t)-cutset of a hypergraph H. Finding such a separator is a flow problem and can be calculated with the flow network $T_L(H)$ presented by Lawler [33]:

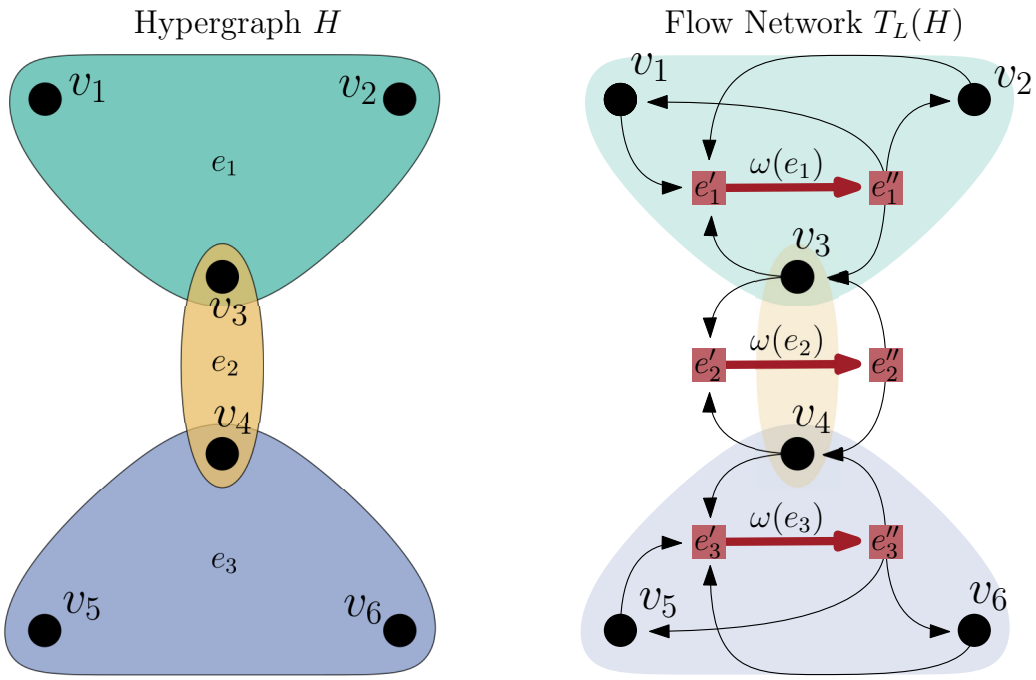

Figure 6: Transformation of a hypergraph into a equivalent flow network by Lawler [33]. Note, capacity of the black edges in the flow network is ∞.

Definition 3.1. *Let T_L be the transformation of a hypergraph $H = (V, E, c, \omega)$ into a flow network $T_L(H) = (V_L, E_L, u_L)$ proposed by Lawler [33]. $T_L(H)$ is defined as follows:*

(i) $V_L = V \cup \bigcup_{e \in E} \{e', e''\}$

(ii) $\forall e \in E$ *we add a directed edge* (e', e'') *with capacity* $u_L(e', e'') = \omega(e)$

(iii) $\forall v \in V$ *and* $\forall e \in I(v)$ *we add two directed edges* (v, e') *and* (e'', v) *with capacity* $u_L(v, e') = u_L(e'', v) = \infty$.

An example of this transformation is shown in Fig. 6. $T_L(H)$ is nearly equivalent to the transformation $T_V(G)$ described in Defintion 2.5 except that we do not have to split the hypernodes $v \in V$. A hypernode cannot be in a minimum-capacity (s,t)-vertex separator because each $v \in V$ has infinity capacity [27]. Therefore, a minimum-capacity (s,t)-cutset of $T_L(H)$ is equal to a minimum (s,t)-vertex separator of $G_*(H)$. The resulting graph $T_L(H)$ has $|V_L| = 2|V|+|E|$ nodes and $|E_L| = 2(\bar{e}+1)|E|$ edges, where $\bar{e}$ is the average size of a hyperedge [43]. Using *Edmond-Karps* maximum flow algorithm (see Section 3.1.1) on flow network $T_L(H)$ takes time $\mathcal{O}(|V|^2|E|^2)$ [33].

A minimum-weight (s,t)-cutset of H can be found by simply mapping the minimum-capacity (s,t)-cutset to their corresponding hyperedges in H (see Section 2.2). The minimum-weight (s,t)-bipartition are all vertices $v \in V$ *reachable* from s in the *residual graph* of $T_L(H)$ and the counterpart are all hypernodes not *reachable* from s.

In Fig. 7 we illustrate the structure of $T_L(H)$ and demonstrate what happens after we *augment* along a path in the Lawler-Network. This figure can be used as a reference if you need an illustration of techniques used in the proofs of Section 4.

In this thesis, we often have to mix up nodes and edges of H and $T_L(H)$. If we use $v \in V_L$, there also exists a corresponding $v \in V$. v can be used in both contexts. For all $e \in E$ there exists two corresponding nodes $e', e'' \in V_L$. e' is called *incoming hyperedge node* and e'' is called *outgoing hyperedge node*. In some cases we need to treat $e', e'' \in V_L$ the same way as their corresponding hyperedge $e \in E \Rightarrow e'_1 \cap e''_2$ or $e''_1 \cap e'_2$ should be the same as $e_1 \cap e_2$.

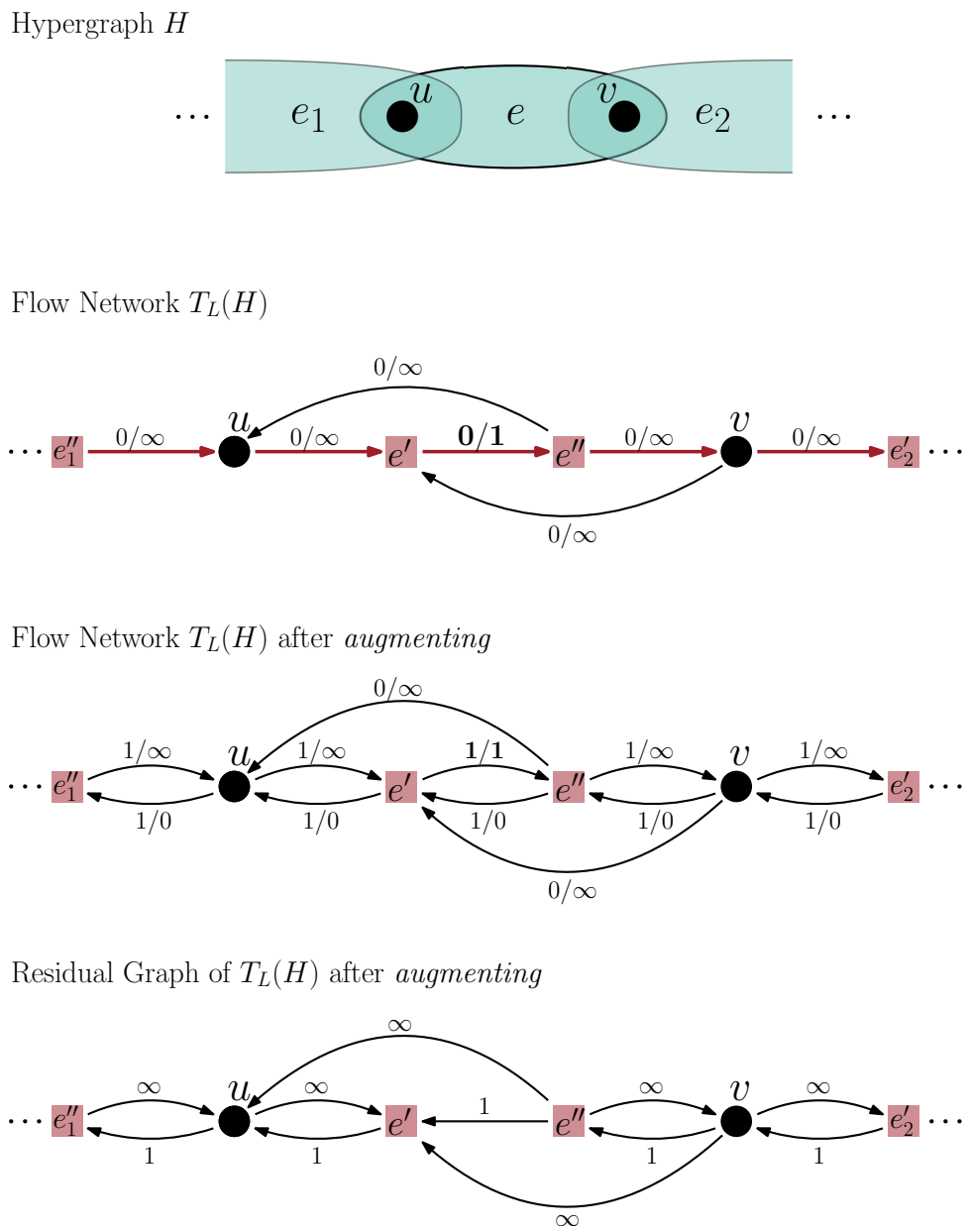

Figure 7: Illustration of the structure of a hyperedge e in $T_L(H)$ and the effect of augmenting along a path in this network. The labeling on an edge represents the flow $f(x, y)$ and the capacity $u(x, y)$ denoted with $f(x, y)/u(x, y)$. The labeling of the edges in the residual graph denotes the residual capacity $r_f(x, y)$. The red highlighted path represents an augmenting path.

3.3. Flow-based Local Search on Graphs

It seems natural to utilize maximum flow computations to improve the cut metric of a given partition of a graph. Lang and Rao [32] use an approach, called *Max-Flow Quotient-cut Improvement* (MQI), to improve the quality of a graph when metrics such as *expansion* or *conductance* are used. For a given bipartition $(S, \bar{S})$, they find the best improvement among all bipartitions $(S', \bar{S}')$ such that $S' \subset S$ by solving a flow problem. Andersen and Lang [4] suggested a flow-based improvement algorithm, called *Improve*, which works similar as MQI, but do not restrict the output of the partition to $S' \subset S$. However, both techniques can not guarantee that the resulting bipartition is balanced and only are applicable for $k = 2$.
Schulz and Sanders [45] integrate flow-based refinement algorithm in their *multilevel graph par-*

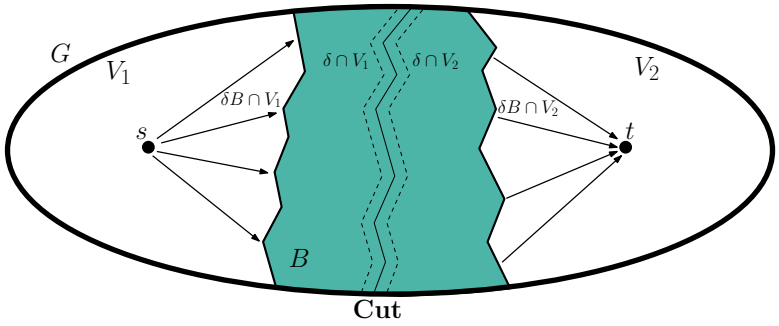

Figure 8: Configuration of a flow problem around the cut of graph G [4].

titioner KaFFPa. In general, they build a flow problem a region B around the cut and connect the *border* of B with the source resp. sink. B is defined in such a way that the flow computation yields a feasible cut according to the *balanced contraint*. Many ideas of this work are used in this thesis and adapted to hypergraphs. Therefore, we will give a detailed description of the concepts and advanced techniques to improve graph partitions.

3.3.1. Balanced Bipartitioning

Let (V_1, V_2) be a balanced bipartition of a graph $G = (V, E, c, \omega)$. Further, $P(v) = 1$, if $v \in V_1$ and $P(v) = 2$, otherwise. We will now explain how a given bipartition can be improved with flow computations. This technique can also be applied on a k-way partition by applying the approach on two adjacent blocks [45].
Let $\delta := \{u \mid \exists (u, v) \in E : P(u) \neq P(v)\}$ be the set of nodes around the cut of G. For a set $B \subseteq V$ we define its border $\delta B := \{u \in B \mid \exists (u, v) \in E : v \notin B\}$. The basic idea is to build a region B around all cut nodes δ of G and connect all nodes in $\delta B \cap V_1$ to the source node s and all nodes in $\delta B \cap V_2$ to the sink node t.
We can construct $B := B_1 \cup B_2$ with two *Breadth First Searches* (*BFS*). One is initialized with all nodes $\delta \cap V_1$ and stops if $c(B_1)$ would exceed $(1 + \epsilon)\frac{c(V)}{2} - c(V_2)$. The second is initialized with all nodes $\delta \cap V_2$ and stops if $c(B_2)$ would exceed $(1 + \epsilon)\frac{c(V)}{2} - c(V_1)$. The two *BFS*s only touches nodes of V_1 resp. $V_2 \Rightarrow B_1 \subseteq V_1$ and $B_2 \subseteq V_2$. The constraints for the weights of B_1 and B_2 guarantes that the bipartition is still balanced after a *Max-Flow-Min-Cut* computation. Connecting s resp. t to all border nodes $\delta B \cap V_1$ resp. $\delta B \cap V_2$ ensures that a non-cut edge not contained in G_B is not a cut edge after assigning the minimum (s, t)-bipartition of subgraph G_B to G. This also yields the conclusion that each minimum (s, t)-cutset in G_B leads to a cut smaller or equal to the old cut of G. All concepts are illustrated in Fig. 8.

3.3.2. Adaptive Flow Iterations

Sanders and Schulz [45] introduce several techniques to improve their basic approach. If the *Max-Flow-Min-Cut* computation on G_B leads to an improved cut, we can apply the method described in Section 3.3.1 again. An extension of this approach is to iteratively adapt the size of the flow problem based on the result of the maximum flow computation. We define $\epsilon' := \alpha\epsilon$ for a $\alpha \geq 1$ and let the size of B depend on ϵ' rather than on ϵ. If we find an improvement on G, we increase α to $\min\{2\alpha, \alpha'\}$ where α' is a predefined upper bound for α. If not, we decrease the size of α to $\max\{\frac{\alpha}{2}, 1\}$. This approach is called *adaptive flow iterations* [45].

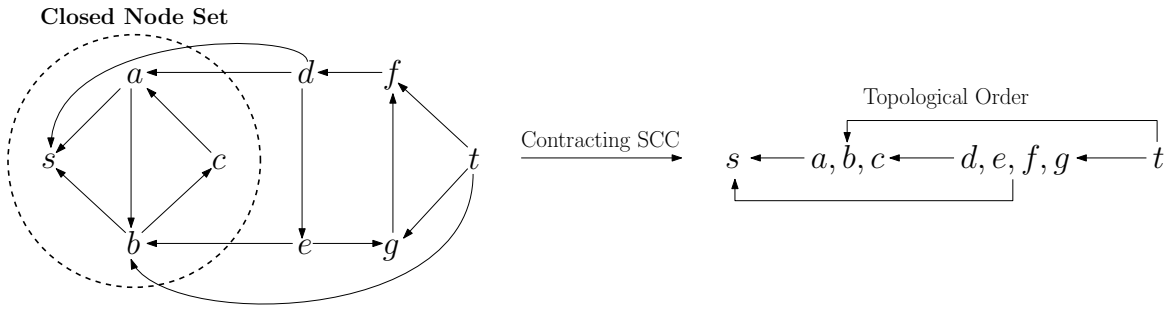

Figure 9: $C = \{s, a, b, c\}$ is a *closed node set* of graph G (left side). After contracting all *Strongly Connected Components*, we can enumerate all *closed node sets* of G by sweeping in reverse topological order over the contracted graph (right side).

3.3.3. Most Balanced Minimum Cut

Picard and Queyranne [42] show that all minimum (s,t)-cutsets are computable with one maximum (s,t)-flow computation. To understand the main theorem and the algorithm to compute all minimum (s,t)-cutsets we need the definition of a *closed node set* $C \subseteq V$ of a graph G.

Definition 3.2. *Let $G = (V, E)$ be a graph and $C \subseteq V$. C is called a closed node set iff the condition $u \in C$ implies that for all edges $(u, v) \in E$ also $v \in C$.*

A *closed node set* is illustrated in Fig. 9. A simple observation is that all nodes on a cycle have to be in the same *closed node set* per definition. Therefore we can contract all *Strongly Connected Components* (SCC) of G with a linear time algorithm proposed by Tarjan [48] and sweep in reverse topological order over the contracted graph to enumerate all *closed node sets*. Note, if we contract all SCCs of G the resulting graph is a *Directed Acyclic Graph* (DAC). Therefore, a topological order exists.
With the Theorem of Picard and Queyranne [42] we can enumerate all minimum (s,t)-cuts of G with one maximum flow computation.

Theorem 3.1. *There is a 1-1 correspondence between the minimum (s,t)-cuts of a graph and the closed node sets containing s in the residual graph of a maximum (s,t)-flow.*

All *closed node sets* in the residual graph of G induce a minimum (s,t)-cutset on G. They can be calculated with the algorithm described above having the residual graph of G as input. The running time of the algorithm is $\mathcal{O}(|V| + |E|)$.
A common problem of the *adaptive flow iteration* approach (see Section 3.3.2) is that using a large α often leads to cuts in G that violate the balanced constraint. We can enumerate all minimum (s,t)-cutsets with one maximum flow computation and therefore have a higher probability to find a feasible partition after a *Max-Flow-Min-Cut* computation. We refer to this method as *Most Balanced Minimum Cut*.

3.3.4. Active Block Scheduling

Active Block Scheduling is a *quotient graph style refinement* technique for k-way partitions [26, 45]. The algorithm is organized in rounds and executes a two-way local improvement algorithm on each adjacent pair of blocks in the *quotient graph* where at least one of both is *active*. Initially all blocks are *active*. A block becomes *inactive* if none of its nodes move in a round. The algorithm terminates, if all blocks are *inactive*.

Fiduccia and Mattheyses [17] introduce a linear time two-way local search heuristic, called *FM* heuristic, which is fundamental for many graph partitioning algorithms. They define the gain $g(v)$ of a node $v \in V$ as the reduction of the cut metric when moving v from its current block to an other block. By maintaining the gains of the nodes in a special data structure, called *bucket queue*, they can find a maximum gain node in constant time. After moving a maximum gain node, they are also able to update the data structure in time equal to the number of adjacent nodes.
The local improvement algorithm (for *Active Block Scheduling*) can either be an *FM* local search or a flow-based approach or even a combination of both as proposed by Sanders and Schulz [45].

3.4. Hypergraph Partitioning

In this Section, we review how most hypergraph partitioners solve the *hypergraph partitioning problem*. The most successful approach is the *multilevel paradigm* [3, 5, 40] which we describe in Section 3.4.1. The results of this thesis is integrated into n-level hypergraph partitioner *KaHyPar*. Therefore, we give a brief overview of implementation details of this framework (see Section 3.4.2).

3.4.1. Multilevel Paradigm

The *multilevel paradigm* is a three phase algorithm to solve the *hypergraph partitioning problem* (see Fig. 10). In the first stage, called *coarsening phase*, vertex matchings or clusterings are calculated which we contract. This process is repeated until a predefined number of hypernodes remains. The sequence of successively smaller hypergraphs is called *levels*. If the hypergraph H is small enough, we can use expensive algorithms to initially partition H into k blocks (*Initial Partitioning*). Afterwards, we *uncontract* each *level* in reverse order of *contraction* by projecting the partition to the next *level*. After *uncontraction* a *refinement* heuristic can be used to improve the quality of the current partition according to an objective function. The most commonly used *refinement* algorithm is the *FM* algorithm [17].

3.4.2. n-Level Hypergraph Partitioning

KaHyPar is a multilevel hypergraph partitioner in its most extreme version, which removes only a single vertex in one *level* of the hierarchy. It seems to be the method of choice for optimizing cut- and the $(\lambda - 1)$-metric unless speed is more important than quality [25]. The framework provides a *direct k-way* [1] and a *recursive bisection* mode, which recursively calculates bipartitions (with *multilevel paradigm*) until the hypergraph is divided into k blocks [46]. *KaHyPar* consists of four phases: *Preprocessing* and the three phases of the *multilevel paradigm*.
In the *preprocessing* step community structures of the hypergraph are detected. The hypergraph is transformed into a bipartite graph $G_*(H)$ (see Section 2.3) and a community detection algorithm is executed which optimizes *modularity* [20, 25]. During the *coarsening phase* contractions are restricted to vertices within the same community. The contraction partners are chosen according to the *heavy-edge* rating function $r(u, v) := \sum_{e \in I(u) \cap I(v)} \frac{\omega(e)}{|e|-1}$ [29]. The function prefers vertices which share a large number of heavy nets with small size. The contraction algorithm works in passes. At the beginning of each pass a random permutation of the vertices is generated and for each vertex u, we determine the contraction partner v according to the *heavy-edge* rating function [46]. A pass ends if each vertex is either considered as representative or contraction partner. The passes are repeated until only $t = 160k$ hypernodes remains.

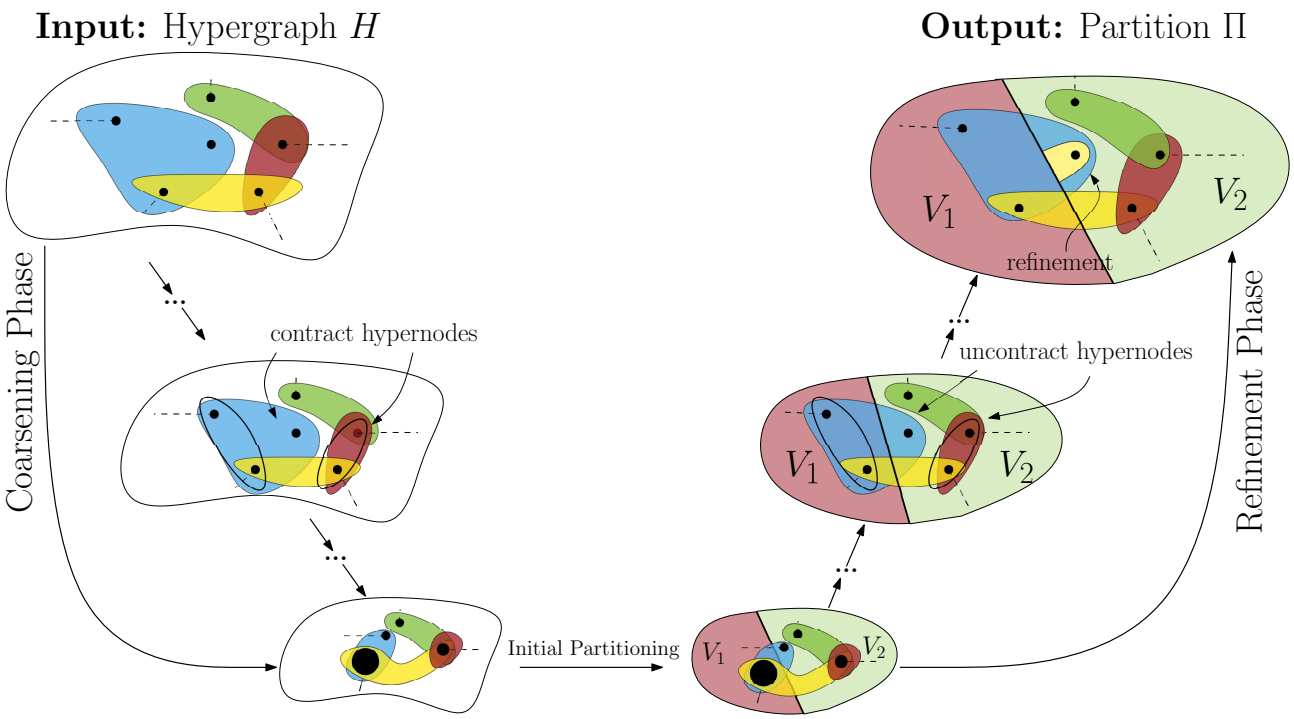

Figure 10: Multilevel Hypergraph Partitioning

The *initial partitioning* uses the *recursive bisection* approach to calculate a k-way partition in combination with a portfolio of initial partitioning techniques [24]. In the *refinement phase*, a localized *FM* search is started [17], initialized with the current uncontracted vertices. The *local search* maintains k *priority queues* (PQ) for each block V_i exactly one [1]. A hypernode v contained in the i-th PQ with gain g means that moving vertex v to block V_i has gain g. After a move, the gains of all adjacent hypernodes are updated with a *delta-gain* update strategy [40]. The recalculation of all gain values at the beginning of a *FM* pass is one of the main bottlenecks of the algorithm [40]. Therefore, Schlag et al. [1, 46] introduce a *gain cache*, which prevents expensive recalculations of the corresponding gain function. The *gain cache* is maintained with *delta-gain* updates in the same way as the *PQs*. Further, the *local search* is stopped, when an improvement during an *FM* pass becomes unlikely. This model is called *adaptive stopping rule* [1]. Sanders and Osipov [39] shows that it is unlikely that *local search* gives an improvement if $p > \frac{\sigma^2}{4\mu^2}$, where p is the number of moves in the current *FM* pass, μ is the average gain, and σ^2 the corresponding variance.

4. Hypergraph Flow Networks

In Section 3.2 we have shown how a hypergraph H can be transformed into a flow network $T_L(H)$ such that each minimum-weight (S,T)-cutset of H is a minimum-capacity (S,T)-cutset of $T_L(H)$ [33]. However, the resulting flow network has significantly more nodes and edges than the original hypergraph. The running time of a maximum (S,T)-flow algorithm depends heavily on the problem size. Therefore, different modeling approaches, which reduce the number of nodes and edges, can have a crucial impact on the running time of the flow algorithm.
We will present techniques to sparsify the flow network proposed by Lawler. First, we will show how *any subset* $V' \subseteq V$ *of hypernodes* could be removed from $T_L(H)$ (see Section 4.1). This approach minimizes the number of nodes, but in some cases, the number of edges can be significantly higher than in $T_L(H)$. The basic idea of this technique can still be applied to remove low degree hypernodes from the Lawler-Network *without* increasing the number of edges (see Section 4.2). Additionally, we show how every hyperedge e of size 2 can be removed by inserting an undirected flow edge between the corresponding nodes (see Section 4.3). Finally, we combine the two suggested approaches into a Hybrid-Network (see Section 4.4).

4.1. Removing Hypernodes via Clique-Expansion

In this Section, we show how all hypernodes of $T_L(H)$ can be removed such that a maximum (S,T)-flow on the new network induce a minimum-weight (S,T)-cutset on H. If a hypernode $v \in V$ occurs in an augmenting path P the previous node in the path must be a hyperedge, either e' or e''. Further, for all $e \in I(v)$ the capacity $u_L(v,e')$ is ∞. Therefore, if we push flow over a hypernode v, coming from a hyperedge, we can redirect the flow to any hyperedge node $e' \in I(v)$ during the whole maximum flow calculation because $u_L(v,e') = \infty$. The following lemma is central to our first sparsifying technique and is illusrated in Fig. 11. Given a graph $G = (V,E)$ we define the two sets $in(u) := \{v \mid (v,u) \in E\}$ and $out(u) := \{v \mid (u,v) \in E\}$ with $u \in V$.

Lemma 4.1 (Shortcut Edges)**.** *Let $G = (V,E,u)$ be a flow network and $u \in V$ a node where all incoming and outgoing edges have capacity equal to ∞. Further, let $G(u) = (V \setminus \{u\}, E_u, u_u)$ be the flow network obtained by removing u and inserting a shortcut edge between each $v \in in(u)$ and $w \in out(u)$ with $u_u(v,w) = \infty$. If f is a maximum (S,T)-flow of G with $|f| < \infty$, then f is equal to a maximum (S,T)-flow f' of $G(u)$ with $u \notin S \cup T$.*

Proof. Let f be a maximum (S,T)-flow of G. We define a maximum (S,T)-flow f' of $G(u)$ as follows:

$$f'(v,w) = \begin{cases} \frac{f(v,u)f(u,w)}{\sum_{w \in out(u)} f(u,w)}, & \text{if } v \in in(u), w \in out(u) \\ f(v,w), & \text{otherwise} \end{cases} \tag{4.1}$$

f' is choosen in such a way that for all $v \in in(u) : \sum_{w \in out(u)} f'(v,w) = f(v,u)$ and for all $w \in out(u) : \sum_{v \in in(u)} f'(v,w) = f(u,w)$. Therefore, f' satisfies the flow conservation constraint and since all capacities are equal to ∞, f' also satisfies the capacity constraint $\Rightarrow f'$ is a valid flow function. Further u is not contained in $S \cup T$ which implies that $|f| = |f'|$.
Let f' be a maximum (S,T)-flow of $G(u)$. We define a maximum (S,T)-flow f of G as follows:

$$\begin{aligned} f(u,w) &= \sum_{x \in in(u)} f'(x,w) \\ f(v,u) &= \sum_{x \in out(u)} f'(v,x) \\ f(x,y) &= f'(x,y) \end{aligned} \tag{4.2}$$

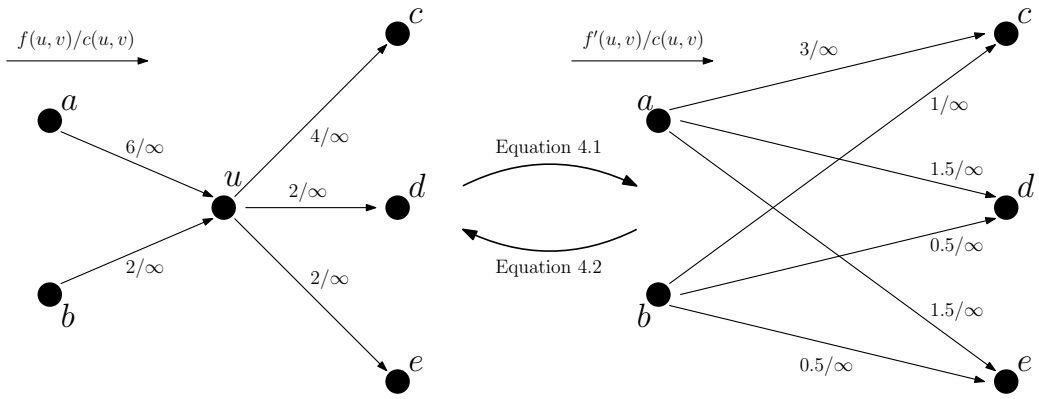

Figure 11: Illustration of Lemma 4.1 and Equation 4.1 and 4.2.

The amount of flow from each $v \in in(u)$ to each $w \in out(u)$ of flow function f' is redirected over u in f. Therefore, f is an valid flow function. Since $u \notin S \cup T$, it follows that $|f| = |f'|$. □

The structure we add between all *incoming* and *outgoing* edges is called *maximum bipartite graph* or *biclique*. In $T_L(H)$ all incoming and outgoing edges of a hypernode v have capacities equal to ∞. For all $e \in I(v)$ there is an edge from v to e' and from e'' to v. Consequently, $in(v) = \bigcup_{e \in I(v)} e''$ and $out(v) = \bigcup_{e \in I(v)} e'$. Therefore, we can remove v and add a *biclique* between $in(v)$ and $out(v)$. A removal of a hypernode induce $d(v)^2$ edges in the new network where $d(v)$ is the degree of hypernode v. However, we can proof that $d(v)(d(v) - 1)$ edges are sufficient to model the problem equivalent. In Appendix B we proof that we can remove a *infinite* weight node v of a graph by adding a clique between all adjacent nodes of v, if we want to find a minimum-weight (s,t)-vertex separator. Using the vertex separator transformation (see Definition 2.5) results in a flow network where a removal of a hypernode induce $d(v)(d(v) - 1)$ instead of $d(v)^2$ edges. The technique is illustrated in Fig. 22. The removed edges are exactly the edges between the same hyperedge nodes. More formally, $\forall e \in I(v)$ we can remove edge (e'', e') from the *biclique* of Lemma 4.1. Therefore, we can construct the following network with Lemma 4.1:

Definition 4.1. *Let T_H be a transformation that converts a hypergraph $H = (V, E, c, \omega)$ into a flow network $T_H(H, V') = (V_H, E_H, u_H)$ with $V' \subseteq V$. $T_H(H, V')$ is defined as follows:*

(i) $V_H = V \setminus V' \bigcup_{e \in E} \{e', e''\}$

(ii) $\forall v \in V'$ *and* $\forall e_1, e_2 \in I(v)$ *with* $e_1 \neq e_2$ *we add a directed edge* (e_1'', e_2') *with capacity* $u_H(e_1'', e_2') = \infty$ *(Lemma 4.1).*

(iii) Let $H' = (V \setminus V', E', c, \omega)$ *be the hypergraph with* $E' = \{e \setminus V' \mid e \in E \ \wedge \ e \setminus V' \neq \emptyset\}$, *then we add all edges of* $T_L(H')$ *to* E_H *with their corresponding capacities.*

An example of the transformation is shown in Fig. 12. We have to proof that a minimum-capacity (S,T)-cutset of $T_H(H, V')$ is equal with a minimum-weight (S,T)-cutset of H. However, we will use the following lemma in the correctness proof.

Lemma 4.2 (Source/Sink Node Removal)**.** *Let $G = (V, E, u)$ be a flow network and f a maximum (S,T)-flow of G with $|f| < \infty$. If $s \in S$ is a source node where all outgoing edges have infinte capacity and $t \in T$ is a sink node where all incoming edges have infinte capacity, then $|f|$ is equal with the amount of a maximum (S', T)-flow f_s of $G(s)$ and a maximum (S, T')-flow f_t of $G(t)$, where $S' = S \setminus \{s\} \cup out(s)$ and $T' = T \setminus \{t\} \cup in(t)$.*

Proof. First we note, that the flow over an incoming edge of a source node $s \in S$ is zero. More formally, $\forall v \in in(s) : f(v, s) = 0$. Edmond and Karp [16] show that we can find a maximum

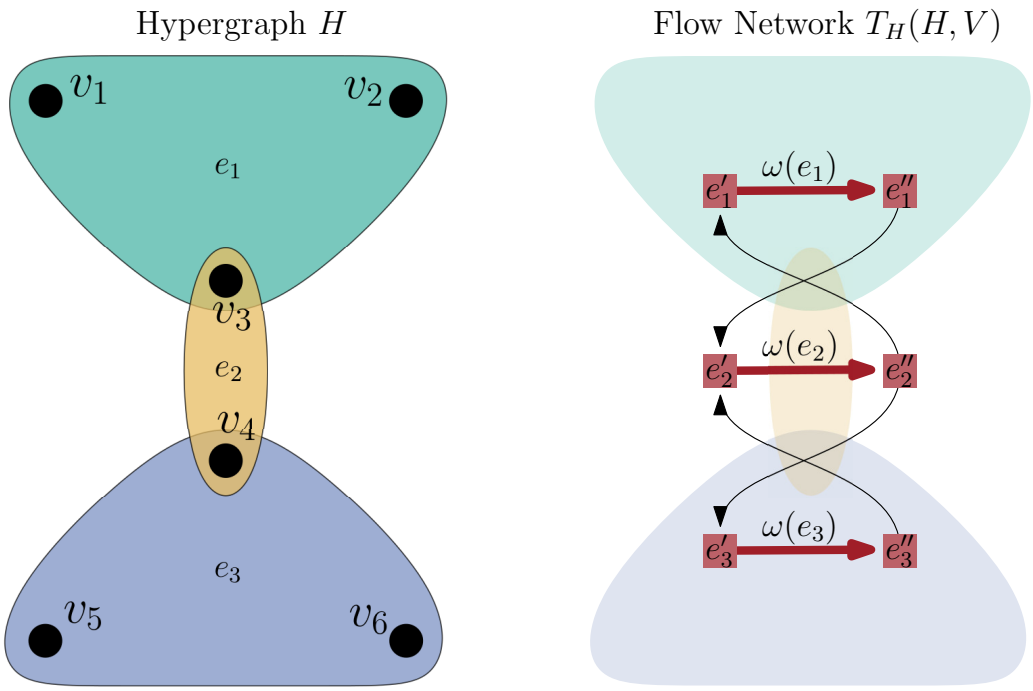

Figure 12: Transformation of a hypergraph H into an equivalent flow network $T_H(H, V)$ by removing all hypernodes of $T_L(H)$. Note, capacity of the black edges in the flow network is ∞.

(s, t)-flow if we augment in each step along a shortest path. Assume we find an augmenting path P which contains an edge (v, s). We can obtain a shorter path if we split P after edge (v, s) and use the second part as augmenting path. Therefore, $f(v, s) = 0$. The same holds for all outgoing edges of a sink node. Consequently, we can remove all incoming resp. outgoing edges of a source resp. sink node.
In Section 2.2 we described how to solve a *multi-source multi sink* flow problem by adding a super source node a and super sink node b to the network and connect a with all sources $s' \in S$ and all sinks $t' \in T$ with b. $\forall s' \in S : u(a, s') = \infty$ and $\forall t' \in T : u(t', b) = \infty$. With Lemma 4.1 follows, that we can remove s from G and insert a directed edge from a to each $v \in out(s)$ (equal to $G(s)$) and $|f| = |f_s|$. The new flow problem corresponds to the *multi-source multi-sink* problem with S' and T as source and sink set. The proof for $G(t)$ is equivalent. □

As a consequence of this lemma, we can remove a source hypernode $v \in S$ of $T_L(H)$ and instead add all incoming hyperedge nodes $e' \in I(v)$ as sources to the flow problem. Because for all incoming resp. outgoing edges of vertices v of $T_L(H)$ the capacity is ∞.

Theorem 4.1. *A minimum-weight (S, T)-cutset of a hypergraph $H = (V, E, c, \omega)$ (with $S, T \subseteq V, S \cap T = \emptyset$) is equivalent with a minimum-capacity (S', T')-cutset of the flow network $T_H(H, V')$ ($V' \subseteq V$) with $S' = S \setminus V' \cup \bigcup_{e \in N(V' \cap S)} \{e'\}$ and $T' = T \setminus V' \cup \bigcup_{e \in N(V' \cap T)} \{e''\}$.*

Proof. Applying Lemma 4.1 and 4.2 on all nodes $v \in V'$ of flow network $T_L(H)$ yields network $T_H(H, V')$ with S' and T' as source and sink sets. A maximum (S, T)-flow f_L of $T_L(H)$ is then equal with a maximum (S', T')-flow f_H of $T_H(H, V')$. Since $|f_L| < \infty$, only edges between hyperedge nodes are contained in a minimum-capacity (S, T)-cutset of $T_L(H)$. Since $|f_L| = |f_H|$, the same holds for a minimum-capacity (S', T')-cutset of $T_H(H, V')$, which is equal with a minimum-weight (S, T)-cutset of H.

□

Consequently, we can find a minimum-weight (S,T)-cutset of H by calculating a minimum-capacity (S',T')-cutset of $T_H(H,V')$. Finally, we have to find the corresponding minimum-weight (S,T)-bipartition. In $T_L(H)$ all hypernodes reachable from source nodes in the residual graph are part of the first and all not reachable are part of the second block of the bipartition. Since we removed all hypernodes $v \in V'$ in our new network, we have to reconstruct the bipartition using the following lemma.

Lemma 4.3 (Reachability of Hypernodes)**.** *Let f be a maximum (S,T)-flow of $T_L(H)$. If a hypernode $v \notin S$ is reachable from a node $s \in S$ in the residual graph of $T_L(H)$, then there must exist at least one net $e \in I(v)$ where e'' is reachable from s in the residual graph of $T_L(H)$.*

Proof. Let A be the set of all nodes reachable from the source nodes S in the residual graph of $T_L(H)$. Assume, if $v \in A$, then $\forall e \in I(v)$ the *outgoing hyperedge node* e'' is not contained in A which implies that all edges (v,e'') are not contained in the residual graph of $T_L(H)$. More formally, $\forall e \in I(v) : r_f(v,e'') = 0$. Otherwise, e'' would be in A because $v \in A$. Since $r_f(v,e'') = f(e'',v) = 0$, there is no flow entering node v and due to the conservation of flow constraint there cannot be any flow leaving node v. Therefore, there is no path from any $s \in S$ to v over a node e', because $\forall e \in I(v) : r_f(e',v) = f(v,e') = 0$ and no path over e'' because $\forall e \in I(v) : e'' \notin A$ Therefore, v is not reachable from any $s \in S$ which is a contradiction to the assumption that $v \in A$. □

Lemma 4.3 gives us an alternative construction for the minimum-weight (S,T)-bipartition of H for both networks $T_L(H)$ and $T_H(H,V')$. Regardless of the flow network, we can calculate a maximum flow on it and define the set E'', which contains all *outgoing hyperedge nodes* e'' *reachable* from a source node $s \in S$ in the *residual graph* of the flow network. Further, $(A := \bigcup_{e \in E''} e,\ V \setminus A)$ is a minimum-weight (S,T)-bipartition of H.

4.2. Low-Degree Hypernodes

The resulting flow network $T_H(H,V)$ proposed in Section 4.1 has significantly fewer nodes than the network $T_L(H)$ proposed by Lawler. On the other hand, the number of edges could be much larger.
Consider a hypernode $v \in V$. We replace v in $T_L(H)$ with a biclique between all e'' and e' which are incident to v. The number of edges added to $T_H(H,V)$ depends on the degree of v. Each vertex $v \in V$ induce $d(v)(d(v)-1)$ edges in $T_H(H,V)$. In $T_L(H)$, a hypernode adds $2d(v)$ edges to the network and add one additional node. A simple observation is that for all hypernodes with $d(v) \leq 3$ the inequality $d(v)(d(v)-1) \leq 2d(v)$ holds. Removing such low degree hypernodes not only reduces the number of nodes, but also the number of edges.
Let $V_d(n) = \{v \in V \mid d(v) \leq n\}$ be the set of all hypernodes with degree smaller or equal n. Then our suggested flow network is $T_H(H, V_d(3))$.

4.3. Removing Graph Hyperedges

If we want to find a minimum-weight (S,T)-cutset of a graph $G = (V,E,\omega)$, we do not have to transform G into an equivalent flow network. We can directly operate on the graph with capacities $u(e) = \omega(e)$ for all $e \in E$ [18]. Hypergraphs are a generalization of graphs, where an edge can consist of more than two nodes. However, a hyperedge e of size 2 can still be interpreted as a graph edge. Instead of modeling those edges as described by Lawler [33] (see hyperedge e_2 in Fig. 6), we can add an undirected flow edge between $v_1, v_2 \in e$ (with $v_1 \neq v_2$) with capacity $u(\{v_1,v_2\}) = \omega(e)$. In the following, we will proof the opposite. We will show that

each undirected graph can be modeled as a directed graph with the same *min-cut* properties. The transformation used in the proof of an undirected to an directed edge will have the same structure as a hyperedge of size two in the Lawler-Network. As a consequence, if we define the network where each hyperedge of size two is modeled with an undirected flow edge, we can use the following lemma to show that both networks have the same value of a maximum (S,T)-flow.

Lemma 4.4 (Transformation of Undirected to Directed Networks)**.** *Let $G = (V, E, u)$ be an undirected flow network with capacity function $u : E \to \mathbb{N}_+$. G can be transformed into a directed graph G' such that the value of a maximum (s,t)-flow f of G is equal with the value of a maximum (s,t)-flow f' of G'. More formally, $|f| = |f'|$.*

Proof. Assume $\forall e \in E : u(e) = 1$. According to Mengers Theorem [37], a maximum (s,t)-flow is then equal with the maximum number of edge-disjoint paths between s and t in a directed graph. This theorem can also be proven for undirected graphs if we replace each undirected edge $e = \{u, v\}$ by five directed edges $(v, x'), (w, x'), (x', x''), (x'', v), (x'', w)$ (see Fig. 13) [37]. Obviously, we can map each set of edge-disjoint paths from s to t from G' to G and vice versa. Therefore, the maximum number of edge-disjoint paths from s to t in G' is then the same as G and therefore, $|f| = |f'|$.
Consider the general case where $\forall e \in E : u(e) \in \mathbb{N}_+$. We can transform the weighted undirected graph G into an unweighted directed multigraph by replacing each undirected edge $e = \{u, v\}$ with $u(e)$ undirected edges of weight 1 (see Fig. 13). Afterwards, we can use the transformation to an unweighted directed multigraph the same way as before. Again, we can apply Menger's Theorem to show that $|f| = |f'|$. Newman [38] showed that there is an one-to-one correspondence between a maximum (s,t)-flow of an unweighted multigraph and its corresponding weighted graph where the weight of each edge (u, v) is the number of parallel edges between u and v of the multigraph. □

As a consequence of the construction of the proof of Lemma 4.4 the weighted directed graph illustrated on the right side of Fig. 13 can be transformed into a single undirected edge with weight $u(\{u, v\}) = u(x', x'')$. Each hyperedge e with $|e| = 2$ has exactly this structure in $T_L(H)$. Therefore, we can construct the following network:

Definition 4.2. *Let T_G be a transformation that converts a hypergraph $H = (V, E, c, \omega)$ into a flow network $T_G(H) = (V_G, E_G, u_G)$. $T_G(H)$ is defined as follows:*

(i) $V_G = V \cup \bigcup_{e \in E : |e| \neq 2} \{e', e''\}$

(ii) $\forall e \in E$ *with* $|e| = 2$ *and* $v_1, v_2 \in e$ $(v_1 \neq v_2)$ *we add two directed edges* (v_1, v_2) *and* (v_2, v_1) *to* E_G *with capacity* $u_G(v_1, v_2) = \omega(e)$ *and* $u_G(v_2, v_1) = \omega(e)$

(iii) Let $H' = (V, E', c, \omega)$ *be the hypergraph with* $E' = \{e \mid e \in E \wedge |e| \neq 2\}$*, then we add all edges of* $T_L(H')$ *to* E_G *with their corresponding capacities.*

An example of transformation $T_G(H)$ is shown in Fig. 14. A hyperedge e of size 2 consists exactly of 4 nodes and 5 edges in $T_L(H)$ (see Fig. 6). The same hyperedge induces 2 nodes and 2 edges in $T_G(H)$ (see Fig. 15).

Theorem 4.2. *A minimum-weight (S,T)-cutset of a hypergraph $H = (V, E, c, \omega)$ (with $S, T \subseteq V, S \cap T = \emptyset$) is equal with a minimum-capacity (S,T)-cutset of the flow network $T_G(H) = (V_G, E_G, u_G)$.*

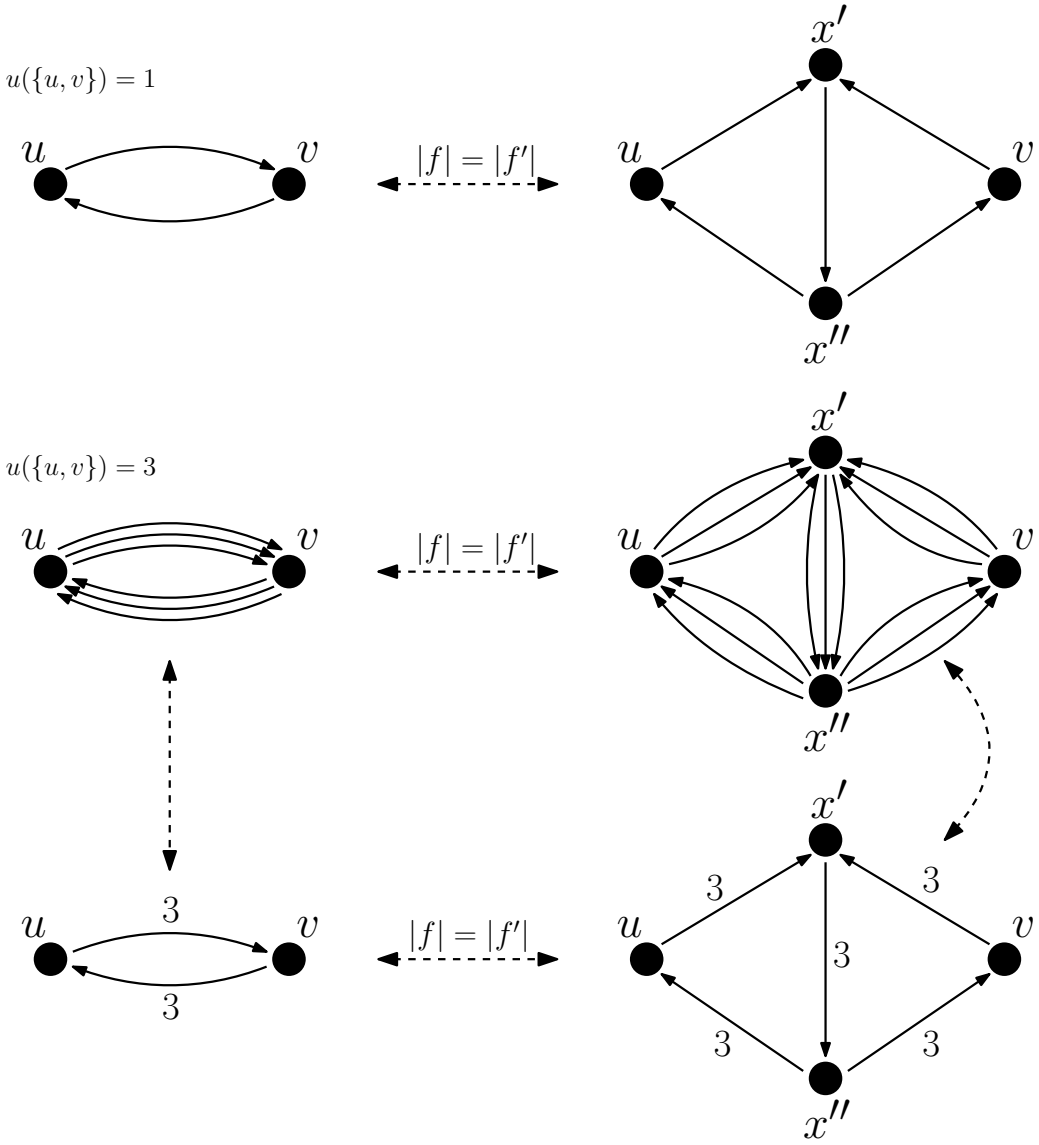

Figure 13: Illustration of the transformation of an unweighted or weighted undirected graph into an unweighted or weighted directed graph. The equivalence of a maximum (s,t)-flow of a unweighted multigraph and their correspondnig weighted graph is a result of a work by Newman [38].

Proof. Consider a hyperedge e with $|e| = 2$ and $u, v \in e$ $(u \neq v)$. The capacity of (u, e'), (v, e'), (e'', u) and (e'', v) is infinity in flow network $T_L(H)$. Before we can apply Lemma 4.4 on all hyperedges e with $|e| = 2$, we have to show how to handle the infinite capacity edges. The flow leaving e' is restricted by $u(e', e'') = \omega(e)$. Therefore, the flow entering e' is restricted by $f(u, e') + f(v, e') \leq u(e', e'') = \omega(e)$. Consequently, $f(u, e') \leq \omega(e)$ and $f(v, e') \leq \omega(e)$. The same holds for $f(e'', u)$ and $f(e'', v)$. Therefore, we can replace each infinite capacity of an edge entering e' or leaving e'' with $\omega(e)$ without changing the value of a maximum (S, T)-flow. We call the capacity adapted network $T_{L'}(H)$.
Applying the transformation of Lemma 4.4 on each undirected edge of $T_G(H)$ results in flow network $T_{L'}(H)$. It follows, that a maximum (S, T)-flow of $T_G(H)$ is equal with a maximum (S, T)-flow of $T_{L'}(H)$ and $T_L(H)$. Consequently, a minimum-capacity (S, T)-cutset of $T_G(H)$ is equal with a minimum-weight (S, T)-cutset of H.

□

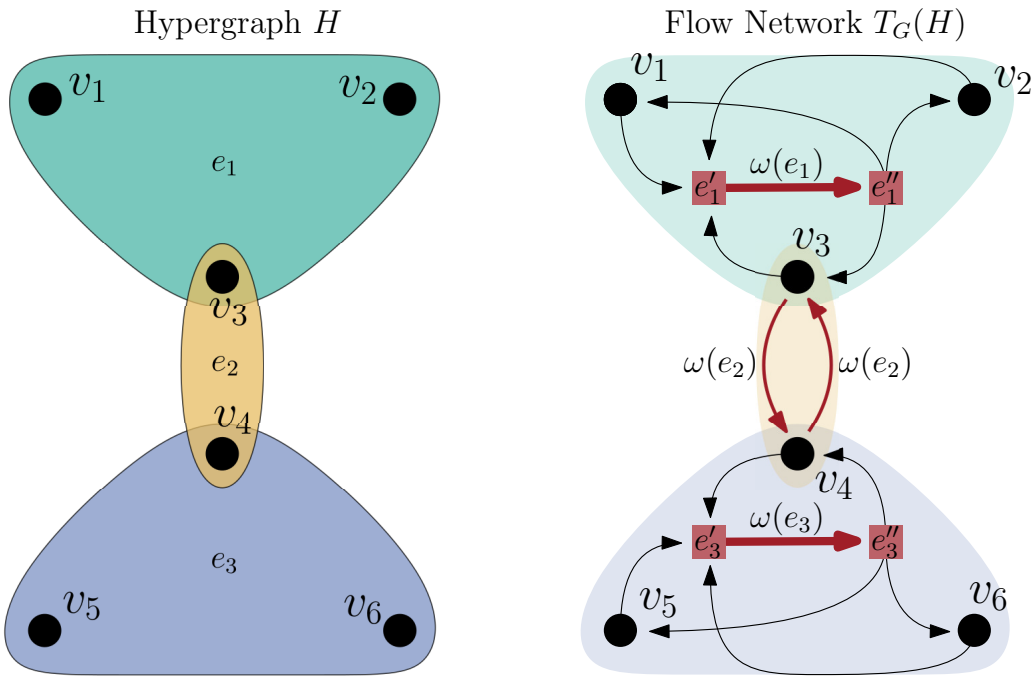

Figure 14: Transformation of a hypergraph into an equivalent flow network by inserting an undirected edge with capacity $\omega(e)$ for each hyperedge of size 2. Note, capacity of the black edges in the flow network is ∞.

A minimum-weight (S, T)-cutset of H can also be calculated with $T_G(H)$. Each edge (v_1, v_2) with $v_1, v_2 \in V$ of the minimum-capacity (S, T)-cutset of $T_G(H)$ can be mapped to their corresponding hyperedge. Since there exists a one-one correspondence between the hypernodes of $T_L(H)$ and $T_G(H)$ the corresponding bipartition are all hypernodes *reachable* from all nodes in S and all not *reachable* from S in the *residual graph* of $T_G(H)$.

4.4. Combining Techniques

The density of a hypergraph $H = (V, E)$ is defined as follows:

$$d := \frac{\overline{d(v)}}{\overline{|e|}} = \frac{|P|/|V|}{|P|/|E|} = \frac{|E|}{|V|}$$

where $\overline{d(v)}$ is the average hypernode degree, $\overline{|e|}$ is the average hyperedge size and $|P|$ are the number of pins. Many real world benchmark instances have either a low or high density. For an example, consider the statistical summary of our benchmark set in Table 8. Hypergraphs with a large density have usually a average hypernode degree significantly greater than the average hyperedge size. Whereas, the opposite behavior can be observed on instances with a low density. High density hypergraphs often have a structure similiar to a graph and low density hypergraphs often have large hyperedges with low degree hypernodes. Of course, there exists instances which are counterexamples to this observation, but for a large majority of real world benchmarks we often find the described behavior.
Currently, we have two different modeling approaches which either perform better on hypergraphs with many low degree hypernodes or small hyperedges. Taking our observation from real-world instances into account means that $T_G(H)$ performs significantly better on high density hypergraphs and $T_H(H, V_d(3))$ on low density hypergraphs. It would be preferable to combine the two approaches into one network which performs best on most instances.

Definition 4.3. *Let T_{Hybrid} be a transformation that converts a hypergraph $H = (V, E, c, \omega)$ into a flow network $T_{Hybrid}(H, V') = (V_{Hybrid}, E_{Hybrid}, u_{Hybrid})$, where $V' = \{v \in V_d(3) \mid \forall e \in I(v) : |e| \neq 2\}$. $T_{Hybrid}(H, V')$ is defined as follows:*

(i) $V_{Hybrid} = V \setminus V' \bigcup_{\substack{e \in E \\ |e| \neq 2}} \{e', e''\}$

(ii) $\forall v \in V'$ *we add a directed edge* (e_1'', e_2'), $\forall e_1, e_2 \in I(v)$ $(e_1 \neq e_2)$ *with capacity* $u_{Hybrid}(e_1'', e_2') = \infty$ *(Lemma 4.1).*

(iii) $\forall e \in E$ *with* $|e| = 2$ *and* $v_1, v_2 \in e$ *we add two directed edges* (v_1, v_2) *and* (v_2, v_1) *with capacity* $u_{Hybrid}(v_1, v_2) = \omega(e)$ *and* $u_{Hybrid}(v_2, v_1) = \omega(e)$ *(Lemma 4.4)*

(iv) $\forall e \in E$ *with* $|e| \neq 2$ *we add a directed edge* (e', e'') *with capacity* $u_{Hybrid}(e', e'') = \omega(e)$ *(same as in* $T_L(H)$*).*

(v) $\forall v \in V \setminus V'$ *we add for each incident hyperedge* $e \in I(v)$ *with* $|e| \neq 2$ *two directed edges* (v, e') *and* (e'', v) *with capacity* $u_{Hybrid}(v, e') = u_{Hybrid}(e'', v) := \infty$ *(same as in* $T_L(H)$*).*

Fig. 15 summarizes all explained transformations of this section. We can prove the correctness of $T_{\text{Hybrid}}(H, V')$ with Lemma 4.1, 4.2 and 4.4 as used in the proof of Theorem 4.1 and 4.2. A minimum-weight (S, T)-cutset of H is equal with a minimum-capacity (S', T')-cutset of $T_{\text{Hybrid}}(H, V')$.
Per definition of $T_{\text{Hybrid}}(H, V')$ we prefer hyperedge removal over hypernode removal. If a hypernode has a degree smaller than or equal to 3, we only remove it, if there is no hyperedge $e \in I(v)$ with $|e| = 2$. The reason for this is that hyperedge removal always removes more nodes and edges than hypernode removal.
The minimum-weight (S, T)-cutset of H can be calculated using the technique described in Section 4.3. Let $(A, V \setminus A)$ be the corresponding bipartition. A is the union of all reachable hypernodes from S' and the union of all reachable *outgoing hyperedge nodes* e'' from S' (see Section 4.1 and Lemma 4.3).

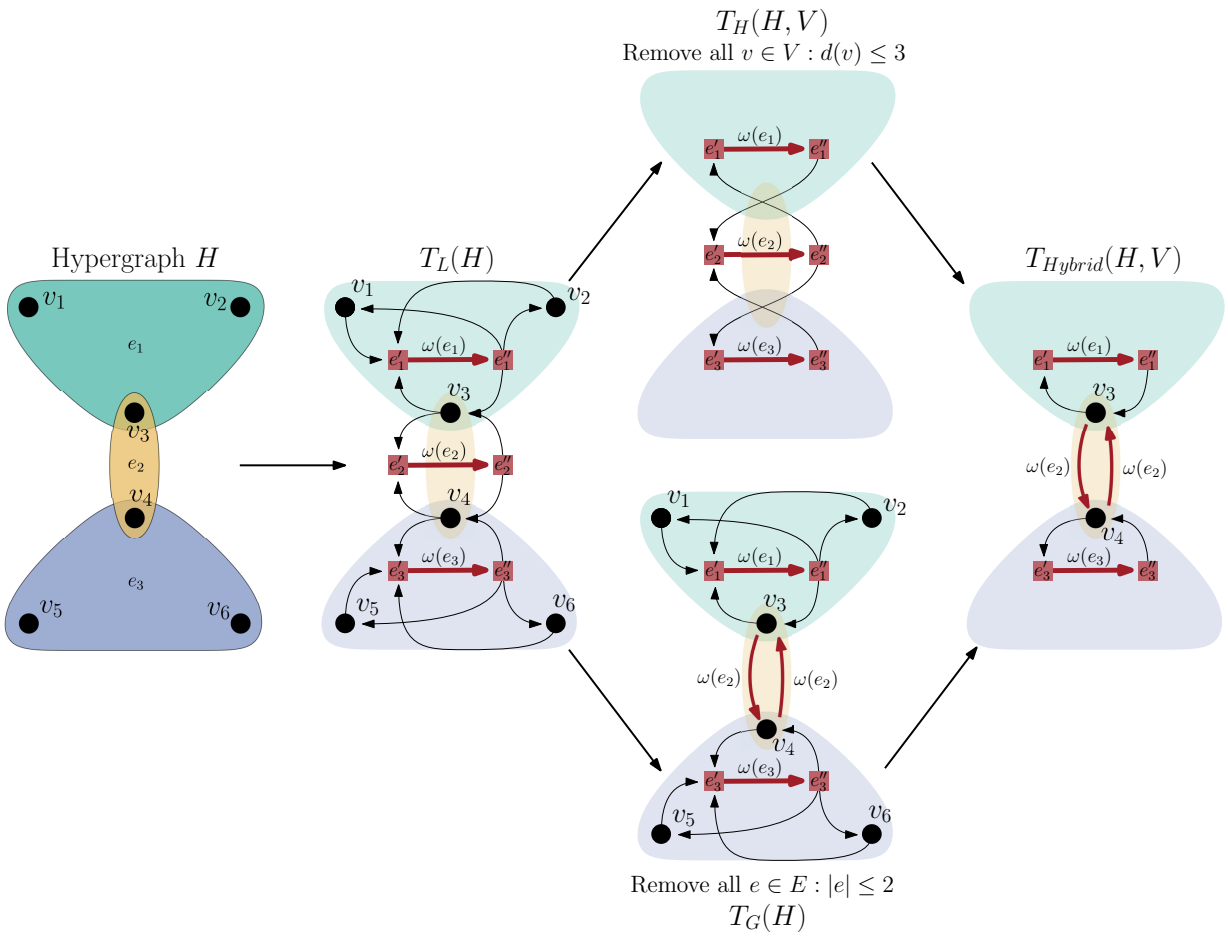

Figure 15: Illustration of all presented techniques to sparsify the flow network of a hypergraph. Transformation from $T_L(H)$ to $T_H(H,V)$ follows with Lemma 4.1. Transformation from $T_L(H)$ to $T_G(H)$ follows with Lemma 4.4.

5. Max-Flow-Min-Cut Refinement Framework

We will now present our direct k-way *flow*-based refinement framework. We use similiar techniques as proposed by Sanders and Schulz [45]. The basic concepts of the framework are illustrated in Fig. 16. The algorithm can be integrated into a *multilevel hypergraph partitioner* by executing the following algorithm in a level of the multilevel hierarchy.
We perform a *flow*-based refinement on two adjacent blocks of a k-way partition $\Pi = \{V_1, \ldots, V_k\}$. The pairwise refinements are embedded into the *active block scheduling* strategy (see Section 3.3.4). The algorithm starts by constructing the quotient graph Q of Π. Afterwards, we iterate over all edges of Q in random order. For each edge (V_i, V_j) of Q, we build a flow problem around the cut of the bipartition induced by V_i and V_j. To construct the flow problem, we use two *BFS*s the first only touches hypernodes of V_i and the second only touches hypernodes of V_j. The *BFS* is initialized with all hypernodes contained in cut hyperedges of the bipartition (V_i, V_j). We embed the pairwise *flow*-based refinement into an *adaptive flow iteration* strategy (as described in Section 3.3.2) which also determines the size of the flow problem. We will denote all hypernodes touched by the two *BFS*s with $V' \subseteq V_i \cup V_j$.
We will use the subhypergraph $H_{V'}$ to construct one of the flow networks proposed in Section 4. We define the corresponding sources S and sinks T of the flow network of $H_{V'}$ in such way that a *Max-Flow-Min-Cut* computation yields an improved k-way partition according to our objective function. After we determine a maximum (S,T)-flow on the flow network, we

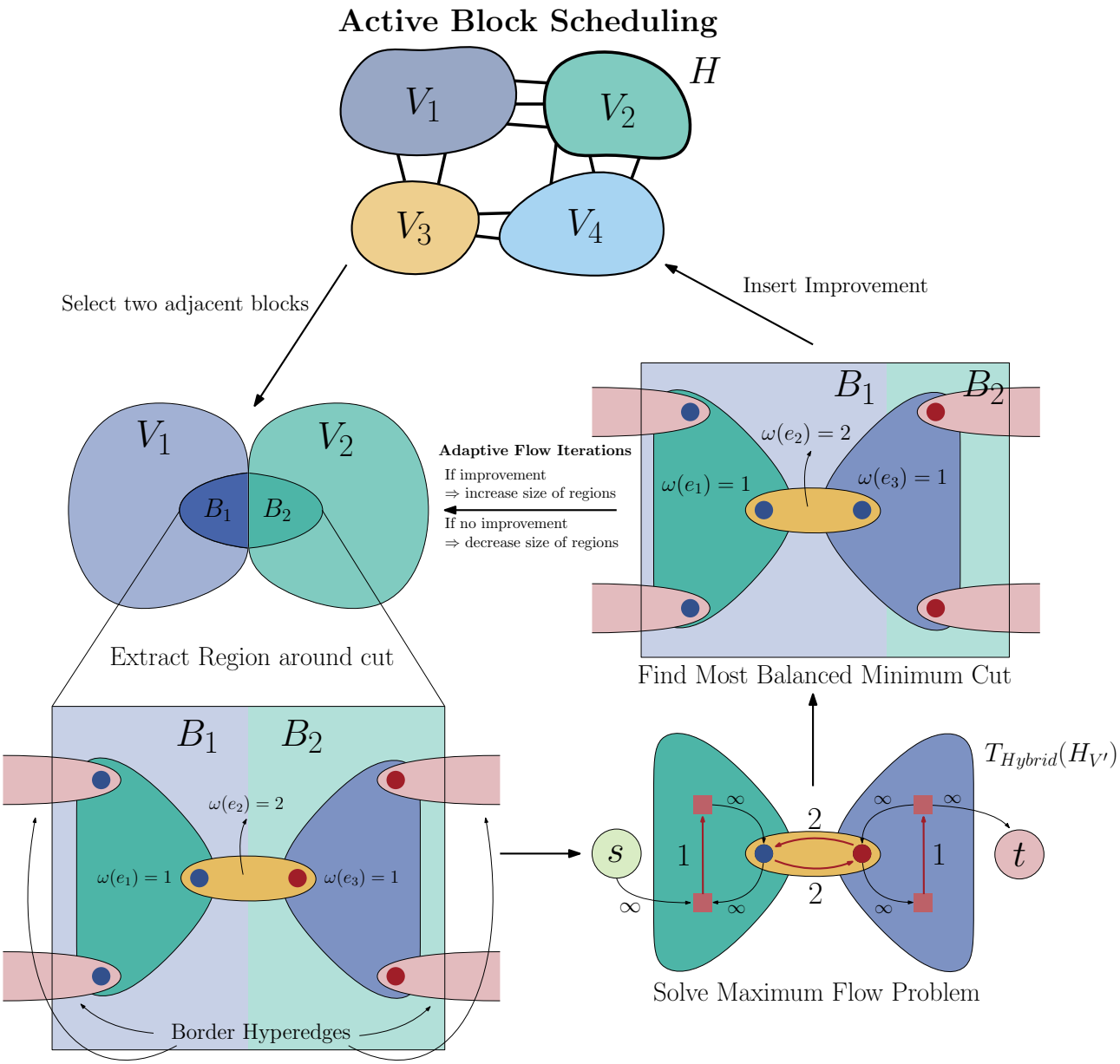

Figure 16: Illustration of our *flow*-based refinement framework for direct k-way hypergraph partitioning.

iterate over all minimum (S, T)-bipartitions of $H_{V'}$ and choose the *Most Balanced Minimum Cut* according to our *balance constraint* (as described in Section 3.3.3).
If a *Max-Flow-Min-Cut* computation yields an improved partition of H, we apply the new partition and execute the algorithm on the same blocks again in which we double the flow problem size according to the *adaptive flow iteration* strategy. If we cannot improve the partition, we decrease the flow problem size twice as small as before and execute the algorithm on the same blocks again. The pairwise *flow*-based refinements stops if the *adaptive flow iteration* scaling parameter α is smaller than 1 (see Section 3.3.2).

5.1. Flow Algorithms

We implement two maximum flow algorithms. One is the *augmenting path* algorithm of Edmond & Karp (EdmondKarp) [16] and the second is the *Push-Relabel* algorithm of Goldberg & Tarjan (GoldbergTarjan) [12, 23]. The EdmondKarp algorithm finds one *augmenting path*

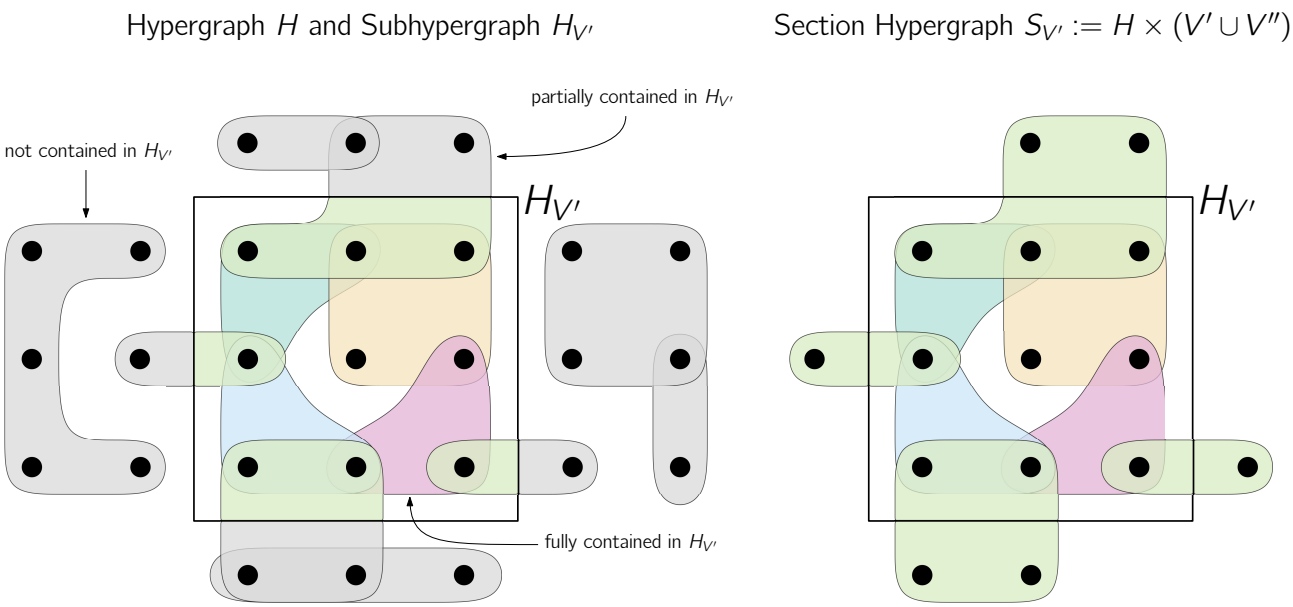

Figure 17: Illustration of the *section hypergraph* $S_{V'}$. Each hyperedge of the the hypergraph H which is fully or partially contained in $H_{V'}$ is fully contained in $S_{V'}$. The nodes not contained in the rectangle of the right figure are part of V''.

with one *BFS* computation in each step. Since we have a *Multi-Source-Multi-Sink* problem, we can find several *augmenting paths* with one *BFS*. After we execute a *BFS* on the residual graph, we search as many as possible edge-disjoint paths in the resulting *BFS*-tree connecting a source s with a sink t. Our Goldberg & Tarjan implementation uses a *FIFO* queue and the *global relabeling* and *gap* heuristic [12].
Further, we integrate two external maximum flow algorithms. The first is from Boykov & Kolmogorov [7] (BoykovKolmogorov[1]) and the second is the *incremental breadth-first search* algorithm of Goldberg et. al [22] (Ibfs[2]). Before we call the external algorithm, we map our internal flow network representation to the one of the external implementation. Afterwards, we map the flow of each edge back to our flow network.

5.2. Source and Sink Configuration

Let Π_1 be the bipartition of a hypergraph $H = (V, E, c, \omega)$. In the following, we show how to configure the source set S and sink set T of the flow network $T_L(H_{V'})$ of a subhypergraph $H_{V'}$ induced by $V' \subseteq V$. The goal is to improve Π_1 with a maximum (S, T)-flow calculation on $T_L(H_{V'})$ (with f as maximum flow) such that after applying the minimum (S, T)-bipartition of $H_{V'}$ too H the resulting bipartition Π_2 has a cut less than or equal to the cut of Π_1. An important concept of this Section will be the definition of the *section hypergraph* (see Defintion 2.8).

Definition 5.1 (Extension of a Subhypergraph)**.** *Given a subset $V' \subseteq V$ of a hypergraph $H = (V, E)$. The section hypergraph $S_{V'} := H \times (V' \cup V'')$ is a hypergraph where V'' contains all pins $u \notin V'$ of hyperedges incident to a vertex $v \in V'$. More formally,*

$$V'' := \bigcup_{\substack{e \in I(V') \\ e \setminus V' \neq \emptyset}} e \setminus V'$$

[1] Available at https://github.com/gerddie/maxflow (Accessed at 14.12.2017)
[2] Available at http://www.cs.tau.ac.il/~sagihed/ibfs/code.html (Accessed at 16.12.2017)

$S_{V'}$ can be seen as an *extension* of subhypergraph $H_{V'}$. Each hyperedge which is partially or fully contained in $H_{V'}$ is fully contained in $S_{V'}$ (see Fig. 17). The source and sink set of $T_L(H_{V'})$ should be chosen in such a way that the two conditions of following problem statement are satisfied:

Problem 5.1. *Given a subhypergraph $H_{V'}$ ($V' \subseteq V$) and bipartition Π_1 of hypergraph H. How should S and T be defined such that after a maximum (S,T)-flow calculation on $T_L(H_{V'})$ (with f as maximum flow) the resulting minimum (S,T)-bipartition Π_2 of H satisfy the following conditions:*

(i) $\omega_H(\Pi_2) \leq \omega_H(\Pi_1)$

(ii) $\Delta_H := \omega_H(\Pi_1) - \omega_H(\Pi_2) = \omega_{S_{V'}}(\Pi_1) - |f| =: \Delta_{H_{V'}}$

The first condition ensures that a *Max-Flow-Min-Cut* computation on $T_L(H_{V'})$ never increases the cut of H. While the second condition allows us to update the cut metric in constant time via $\omega_H(\Pi_2) = w_H(\Pi_1) - \Delta_{H_{V'}}$, instead of having to sum up the weight of all cut hyperedges. Since we have to build the subhypergraph $H_{V'}$ before each maximum flow computation, we can implicitly calculate $\omega_{S_{V'}}(\Pi_1)$.
Note, we define $\Delta_{H_{V'}}$ over the cut of the *section hypergraph* $S_{V'}$. If only hypernodes contained in V' can change its block after a *Max-Flow-Min-Cut* computation then the equality

$$\Delta_H := \omega_H(\Pi_1) - \omega_H(\Pi_2) = \omega_{S_{V'}}(\Pi_1) - \omega_{S_{V'}}(\Pi_2) =: \Delta_{H_{V'}}$$

holds, because all hyperedges partially contained in $H_{V'}$ are fully contained in $S_{V'}$. For example, if a *Max-Flow-Min-Cut* computation on $H_{V'}$ removes a hyperedge e from the cut in $H_{V'}$, but e is still cut in H, then the equality would not hold if we would have defined $\Delta_{H_{V'}}$ over the cut of $H_{V'}$, because $\Delta_{H_{V'}}$ would be equal to 1 and Δ_H equal to 0. Further, if we can show that $|f| = \omega_{S_{V'}}(\Pi_2)$, we simultaneously show that our source and sink set modeling approach satisfies condition (ii) $\Delta_H = \Delta_{H_{V'}}$.
We will now define our source and sink set for the flow network $T_L(H_{V'})$ such that we satisfy the two conditions of Problem 5.1. In the following, $H_{V'}$ is the subhypergraph induced by $V' \subseteq V$ and V'' is the hypernode set as defined in Definition 5.1.

Theorem 5.1. *Let $\Pi_1 = (V_1, V_2)$ be the bipartition of H. The resulting bipartition Π_2 of H of a maximum (S,T)-flow computation on $T_L(H_{V'})$ with*

$$S = \{e' \mid e \in I(V'' \cap V_1)\}$$
$$T = \{e'' \mid e \in I(V'' \cap V_2)\}$$

satisfies the following two conditions:

(i) $\omega_H(\Pi_2) \leq \omega_H(\Pi_1)$

(ii) $\Delta_H = \Delta_{H_{V'}}$

Proof. We will first define S and T for flow network $T_L(S_{V'})$, because for each maximum (S,T)-flow f of $T_L(S_{V'})$ and its corresponding minimum (S,T)-bipartition Π_2 the equality $|f| = \omega_{S_{V'}}(\Pi_2)$ holds due to the *max-flow-min-cut* theorem [18]. Defining a hypernode $v \in V_1$ resp. $v \in V_2$ as source resp. sink means that it cannot change its block after a *Max-Flow-Min-Cut* computation. However, we do not want that a hypernode $v \notin V'$ can change its block after a *Max-Flow-Min-Cut* computation on $T_L(S_{V'})$, because such hypernodes cannot move if we solve a flow problem on subhypergraph $H_{V'}$. Therefore, we define all hypernodes $V'' \cap V_1$ as sources and all hypernodes $V'' \cap V_2$ as sinks (V'' is defined in Defintion 5.1). More formally:

$$S' = V'' \cap V_1$$
$$T' = V'' \cap V_2$$

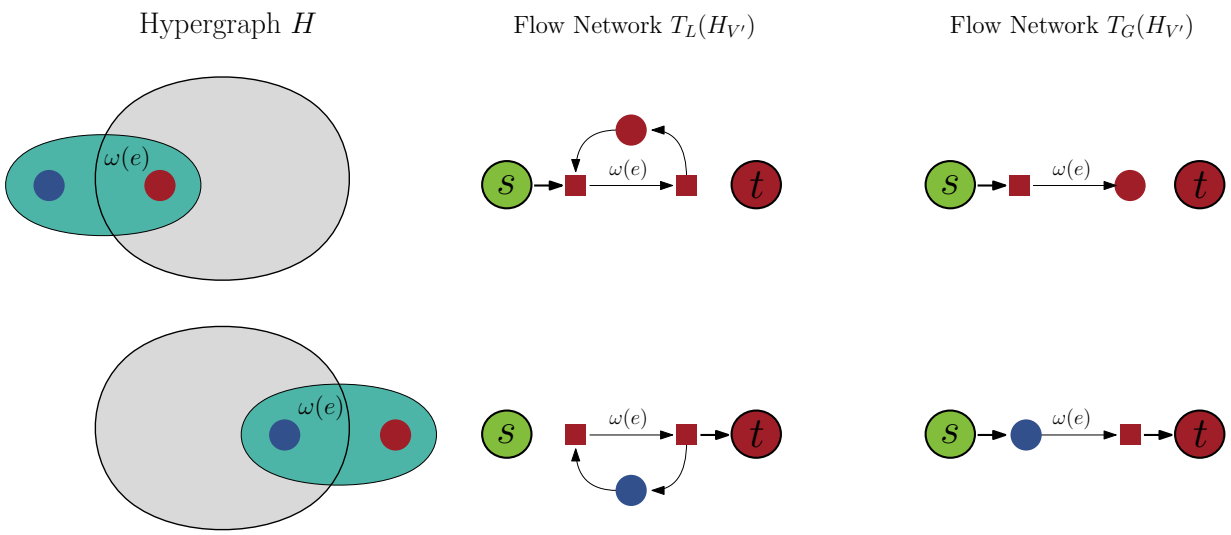

Figure 18: Illustration of modeling hyperedges of size two if the incoming or outgoing hyperedge node is a source or a sink node of the flow problem.

The pins of a hyperedge, which is partially contained in the *section hypergraph* $S_{V'}$ are not able to change its block after a *Max-Flow-Min-Cut* computation, because we define them either as source or a sink. Consequently, no hyperedge partially contained in $S_{V'}$ can change its state from non-cut to cut and therefore it follows with the *max-flow-min-cut* theorem [18] that the inequality $\omega_H(\Pi_2) \leq \omega_H(\Pi_1)$ holds. Since, only nets fully contained in $S_{V'}$ can change its state and $|f| = \omega_{S_{V'}}(\Pi_2)$, it holds that $\Delta_H = \Delta_{H_{V'}}$.
The value of a maximum (S', T')-flow of $T_L(S_{V'})$ is equal with a maximum (S, T)-flow of $T_H(S_{V'}, V'')$ according to Theorem 4.1 with

$$S = S' \setminus V'' \cup \bigcup_{e \in I(V'' \cap S')} \{e'\} \overset{\substack{S' \setminus V'' = \emptyset \\ V'' \cap S' = V'' \cap V_1}}{=} \bigcup_{e \in I(V'' \cap V_1)} \{e'\} = \{e' \mid e \in I(V'' \cap V_1)\}$$

$$T = T' \setminus V'' \cup \bigcup_{e \in I(V'' \cap T')} \{e''\} \overset{\substack{T' \setminus V'' = \emptyset \\ V'' \cap T' = V'' \cap V_2}}{=} \bigcup_{e \in I(V'' \cap V_2)} \{e''\} = \{e'' \mid e \in I(V'' \cap V_2)\}$$

Since each $v \in V''$ is either a source or sink node of $T_L(S_{V'})$, the removal of v does not induce any additional edges in $T_H(S_{V'}, V'')$ (see Lemma 4.2). Therefore, $T_H(S_{V'}, V'') = T_L(H_{V'})$. □

The value of a maximum (S', T')-flow of $T_L(S_{V'})$ and a maximum (S, T)-flow of $T_L(H_{V'})$ are equal. Since S' and T' satisfy conditions (i) and (ii) of our problem statement, also S and T satisfy the two conditions. In general, each hyperedge partially contained in $H_{V'}$, which contains at least one pin $v \notin V'$ of block V_1 resp. V_2 is a source resp. sink node. Furthermore, no hypernode of $T_L(H_{V'})$ is either a source or sink node. Consequently, all hypernodes of V' can change their block after a *Max-Flow-Min-Cut* computation. According to the *max-flow-min-cut* theorem, the value of the cut of bipartition Π_2 is the minimum cut of all possible bipartitions with the restriction that only hypernodes of V' can move.
However, we can model hyperedges of size one more efficiently (see Fig. 18). If the incoming hyperedge node e' is a source node, we can replace the hyperedge of size one with a directed edge (e', v) with $v \in e \cap V'$ and capacity $\omega(e)$. If the outgoing hyperege node e'' is a sink node, we add a directed edge (v, e'') with capacity $\omega(e)$. If e' and e'' is neither a source nor sink node, we can remove the hyperedge from the flow problem.
With the given approach we can optimize the cut metric of a given bipartition of a hypergraph

H. We can transfer those results to improve the connectivity metric of a k-way partition $\Pi = (V_1, \ldots, V_k)$. Let $V' \subseteq V_i \cup V_j$ be a subset of the hypernodes of two adjacent blocks V_i and V_j. If we optimize the cut of subhypergraph $H_{V'}$ we simultaneously optimize the connectivity metric of H. The reduction of the cut of $H_{V'}$ is then equal with the decrease in the connectivity metric of H.

Implications for Graph Partitioning

If we compare our source and sink set modeling approach with the one of Sanders and Schulz [45] (see Section 3.3.1), we can show that with our technique better minimum (S, T)-bipartitions are achievable. They define each node of the graph as source resp. sink, which is adjacent to a node not contained in the flow problem of block V_1 resp. V_2. Consequently, a non-cut edge of the graph partially contained in the flow problem cannot become a cut edge. Therefore, their modeling approach satisfies condition (i) of our problem statement. However, a movement of a node adjacent to a non-cut edge can still improve the cut, if the number of adjacent non-cut edges is smaller than the number of adjacent cut edges. If we interprete a graph as hypergraph, we can use our modeling approach on network $T_L(H_{V'})$ or $T_G(H_{V'})$ with S and T as source and sink set. All nodes incident to a non-cut edge which is partially contained in the flow problem are now able to change their block and the corresponding minimum (S, T)-bipartition is minimum among all possible bipartitions where only nodes of V' can move.

5.3. Most Balanced Minimum Cuts on Hypergraphs

Picard and Queyranne [42] show that all minimum (s, t)-cuts of a graph G are computable with one maximum (s, t)-flow computation by iterating through all *closed node sets* of the residual graph of G.
We can apply the same algorithm on hypergraphs. A minimum-capacity (s, t)-cutset of $T_L(H)$ is equal with a minimum-weight (s, t)-cutset of H. With the algorithm of Section 3.3.3 we can find all minimum-capacity (s, t)-cutsets of $T_L(H)$, which are also minimum-weight (s, t)-cutsets of H. The corresponding minimum-weight (s, t)-bipartitions are all *closed node sets* of the residual graph of $T_L(H)$. However, when we use e.g., $T_H(H, V')$ (see Section 4.1) or $T_{\text{Hybrid}}(H, V')$ (see Section 4.4) as underlying flow network, some hypernodes are removed from the flow problem. However, if an *outgoing* hyperedge node e'' is part of a *closed node set*, than all hypernodes $v \in e$ must be part of it, too, which is a consequence of Lemma 4.3. But the algorithm would become a lot more complicated. Since the algorithm has a linear running time, we decided to use a different approach. We simply reinsert all removed hypernodes with the corresponding edges of the Lawler-Network before we compute the *Most Balanced Minimum Cut*.

Lemma 5.1. *Let $T_L(H) = (V, E_L, u_L)$ be the Lawler-Network and $T_H(H, V') = (V \setminus V', E_H, u_H)$ be the flow network proposed in Section 4.1 of hypergraph $H = (V, E, c, \omega)$ with $V' \subseteq V$. If f is a maximum (S, T)-flow of $T_H(H, V')$, then f is also a maximum (S, T)-flow of the flow network $T = (V, E_L \cup E_H, u)$ with $u(v, w) = u_L(v, w)$, if $(v, w) \in E_L$ and $u(v, w) = u_H(v, w)$, otherwise.*

Proof. The main statement of the lemma is that we can calculate a maximum (S, T)-flow f of $T_H(H, V')$ and then reinsert all removed hypernodes with their corresponding edges of the Lawler-Network. The flow f is also a maximum (S, T)-flow on the resulting flow network T. Assume, that there still exists an *augmenting path* in the residual graph of T. If we remove a hypernode v from the flow network $T_L(H)$, we insert *shortcut* edges between all incident

hyperedges $e \in I(v)$. However, if the *augmenting path* $P = (s, \ldots, e_1'', v, e_2', \ldots, t)$ contains a reinserted hypernode $v \in V'$, we can simply remove it from the path. After removing all $v \in V'$ from P, we obtain a valid path in $T_H(H, V')$. The resulting path contains then the *shortcut* edges (e_1'', e_2') instead of the two inserted edges (e_1'', v) and (v, e_2'). All involved edges have capacity equal to ∞. Therefore, it must be an *augmenting path* in $T_H(H, V')$, which is a contradiction that f is a maximum (S, T)-flow. □

Consequently, if we remove a hypernode from the flow network, we can reinsert it with the corresponding incident edges of the Lawler-Network after a maximum (S, T)-flow computation again. All minimum (S, T)-cutsets of the resulting flow network T will have the same value as all minimum (S, T)-cutsets of $T_H(H, V')$, because the value of a maximum (S, T)-flow is in both networks the same.

5.4. Integration into KaHyPar

Flow Execution Policies

Since *KaHyPar* is an n-level hypergraph partitioner, local searches are executed after each uncontraction of a single vertex (see Section 3.4.2). Using our *flow*-based refinement algorithm in each level would be too expensive. Therefore, we introduce *Flow Execution Policies*, which control total number of *flow*-based refinements throughout the multilevel hierarchy. The first policy is to execute our *flow*-based refinement on each level i where $i = \beta \cdot j$ with $j \in \mathbb{N}_+$ and β as a predefined tuning parameter. Another approach is to simulate a multilevel partitioner with $\log(n)$ hierarchies. A *flow*-based refinement is then executed on each level i where $i = 2^j$ with $j \in \mathbb{N}_+$. Each policy also performs the *active block scheduling* refinement strategy on the last level of the hierarchy. In all remaining levels where no flow is executed, we can use an *FM* algorithm [1, 17, 44] (see Section 3.3.4).

Combining Flow-Based Refinements with the FM algorithm

The *FM* algorithms integrated into *KaHyPar* use a *gain cache* to maintain the gain values of moves througout the multilevel hierarchy. The concept prevents expensive recalculations of gain values if a *FM* local search is instantiated. However, if we use *flow*-based refinement in combination with the *FM* algorithm, we have to ensure that the *gain cache* contains valid entries at each time. Therefore, we undo all changes after a *flow*-based refinement and simulate the moves with the *FM* algorithm to ensure that all entries in the *gain cache* are valid.

Speedup Heuristics

An observation during early experiments was that only a minority of the pairwise refinements based on flows yields an improvement on hypergraph H. Thus, we introduce several rules which might prevent *unpromising* flow executions to speed-up the running time.

(R1) The *acitve block scheduling* refinement strategy is executed in rounds. In each round we use *flows* to improve the bipartition of two adjacent blocks, where one of the two is *active*. Initially, all blocks are *active*. A block becomes *inactive*, if its border does not change in a round. However, we introduce a second criteria when to use *flow*-based refinement on two adjacent blocks. For each pair of adjacent blocks, we count the number of how many

times we found an improvement on these blocks throughout the multilevel hierarchy. The first round of *active block scheduling* is executed as before. In all remaining rounds, we only execute a pairwise *flow*-based refinement, if one of the two blocks is *active* and if we found at least one improvement before on the corresponding blocks.

(R2) If the cut between two adjacent blocks is small (e.g. ≤ 10) we skip the *flow*-based refinement on the blocks except on the last level of the hierarchy.

(R3) If the value of the cut of a minimum (S, T)-bipartition of $H_{V'}$ is the same as the cut before, we stop the *adaptive flow iteration* strategy.

6. Experimental Results

In this Section, we evaluate the performance of our flow-based refinement framework. First, we show the effects of our sparsification techniques of the Lawler-Network [33] on the performance of our maximum flow algorithms. Afterwards, we analyze how the maximum flow problem size influences the solution quality of different configurations of our framework. Finally, we compare the new version of *KaHyPar* with *Max-Flow-Min-Cut* computations against with state-of-the-art hypergraph partitioners.

6.1. Instances

Our full benchmark set consists of 488 hypergraphs from three different application areas. For VLSI design we use instances from the *ISPD98 VLSI Circuit Benchmark Suite* (ISPD98) [2] and add more recent instances of the *DAC 2012 Routability-Driven Placement Contest* (DAC) [49]. We interpret the Sparse Matrix instances of the *Florida Sparse Matrix Collection* (SPM) [13] as hypergraphs using the row-net model [10]. The rows of each matrix are treated as hyperedges and the columns are the vertices of the hypergraph. Our last benchmark type are SAT formulas of the *International SAT Competition 2014* [6]. A common interpretation of a SAT formula as hypergraph is to interpret the literals as vertices and each clause as a net (LITERAL) [40]. Mann and Papp [36] suggested two other hypergraph representation of SAT formulas, called PRIMAL and DUAL. The PRIMAL representation treats each variable as vertex and each clause as hyperedge. The DUAL representation treats each clause as vertex and the variables induces nets containing all clauses where the corresponding variable occurs. A summary of the different instance types is presented in Table 8.
We divide our full benchmark set into two smaller subsets. Our *parameter tuning* benchmark set consists of 25 hypergraphs, 5 of each instance type (except DAC). Additionally, we choose a benchmark subset of 165 instances. On our general experiments we partition each hypergraph into $k \in \{2, 4, 8, 16, 32, 64, 128\}$ blocks and use 10 different *seeds* for each k and an imbalance of $\epsilon = 3\%$.

6.2. System and Methodology

Our experiments run on a single core of a machine consisting of two *Intel Xeon E5- 2670 Octa-Core* processors clocked at 2.6 GHz. The machine has 64 GB main memory, 20 MB L3- and 8 × 256 KB L2-Cache. The code is written in C++ and compiled using g++−5.2 with flags −O3 −mtune=nactive −march=native. We refer to our new implementation of *KaHyPar* with *(M)ax-(F)low-Min-Cut* computations as KaHyPar-MF and the latest configuration with *(C)ommunity-(A)ware* coarsening as KaHyPar-CA.
We compare KaHyPar-MF with the state-of-the-art hypergraph partitioners *hMetis* [29, 30] and *PaToH* [10]. *hMetis* provides a direct k-way (hMetis-K) and recursive bisection (hMetis-R) implementation. Further, we use the default configuration (PaToH-D) and quality preset (PaToH-Q) of *PaToH*. We configure *hMetis* to optimize the *sum-of-external-degree*-metric (SOED) and calculate $(\lambda-1)(\Pi) = \text{SOED}(\Pi) - \text{cut}(\Pi)$. This is also suggested by the authors of *hMetis* [30]. Additionally, we have to adapt the imbalance definition of hMetis-R. An imbalance value of 5 means that the weight of each bisected block is allowed to be between $0.45 \cdot c(V)$ and $0.55 \cdot c(V)$. To ensure that hMetis-R produces a valid ϵ-balanced partition after $\log_2(k)$ bisections we have to adapt ϵ to

$$\epsilon' = 100 \cdot \left(\left((1+\epsilon) \frac{\lceil \frac{c(V)}{k} \rceil}{c(V)} \right)^{\frac{1}{\log_2(k)}} - 0.5 \right)$$

If we evaluate the performance of our hypergraph partitioner, we first summarize the results by calculating the arithmetic mean (or minimum) of a specific metric for the diffrent *seeds* of a hypergraph instance partitioned into k blocks. Afterwards, we calculate the *geometric mean* of all instances to give each instance comparable influence on the final result. To compare the performance of different hypergraph partitioners in more detail we use performance plots introduced in [46]. In the following, we will call a k-way partition of a hypergraph an *instance*. For each algorithm we determine the instance with the minimum cut (of the 10 different *seeds*). Each color in the plot corresponds to one algorithm. The x-axis represents the number of instances and the y-axis represents the quality ratio produced by an algorithm for an instance relative to the partition of the best algorithm. For example, partitioner P_1 produces a partition for an instance X with quality 100 and partitioner P_2 produces a partition for the same instance of quality 105. Then, the y-value for instance X of partitioner P_2 is $1 - \frac{105}{100} = 0.05$ and for partitioner P_1 is $1 - \frac{100}{100} = 0$, which means that the partition of partitioner P_2 for instance X is 5% worse than the best partition produced for instance X. A value of zero indicates that the algorithm produced the best partition. A point close to one indicates that the partition produced by the corresponding partitioner was considerably worse than the partition produced by the best algorithm. Before we insert the points in the grid, we sort them in decreasing order according to the y-values. An algorithm is considered to outperform another algorithm if its corresponding ratio values are below those of the other algorithm. A point with an y-value greater than one corresponds to an infeasible solution that violated the balanced constraint.

6.3. Flow Algorithms and Networks

In the first experiment, we evaluate the effect of our sparsification techniques on the performance of our maximum flow algorithms EdmondKarp, GoldbergTarjan, BoykovKolmogorov and Ibfs. We refer to the Lawler-Network as T_L, which is our baseline flow network. In Section 4 we present several techniques to reduce the number of nodes and edges of T_L. T_H represents our flow network in which we remove all hypernodes with a degree smaller or equal to 3. The network T_G models each hyperedge of size 2 as undirected graph edge between the corresponding pins. Finally, T_{Hybrid} combines both networks.
We evaluate the performance of our maximum flow algorithms on flow problems with $|V'| \in \{500, 1000, 5000, 10000, 25000\}$ hypernodes. The instances are generated by executing *KaHyPar* on our benchmark subset (see Table 7) for $k = 2$ and five different seeds. After an instance is bipartitioned, we generate flow problems with the above-mentioned sizes and execute each possible combination of flow algorithm and flow network on it.
Fig. 19 shows the number of nodes and edges of the resulting flow networks for flow problems with 25000 hypernodes. As expected, T_H reduces the number of nodes more significantly on low hypernode degree instances (Dual) and T_G more on small hyperedge size instances (Primal and Literal). Further, T_{Hybrid} combines the advantages of both networks and reduces the number of nodes and edges of nearly each benchmark type by at least a factor of 2, except on Spm instances. If we compare the sizes of the resulting flow problem, we can observe that instances with a high density ($d = \frac{|E|}{|V|}$), like Primal or Literal, yield large flow problem instances and instances with a low density yield small flow problem instances (see Dual instances).
In Fig. 20 we compare the performance of our maximum flow algorithms on different flow networks. A bar in the plot indicates the speed-up of the corresponding algorithm executed on T_H, T_G or T_{Hybrid} relative to the execution on T_L. The main observation is that the speed-ups are nearly proportional to the reduction of the number of nodes and edges of the corresponding flow network. For example, T_{Hybrid} reduces the size of the flow problems of nearly each benchmark

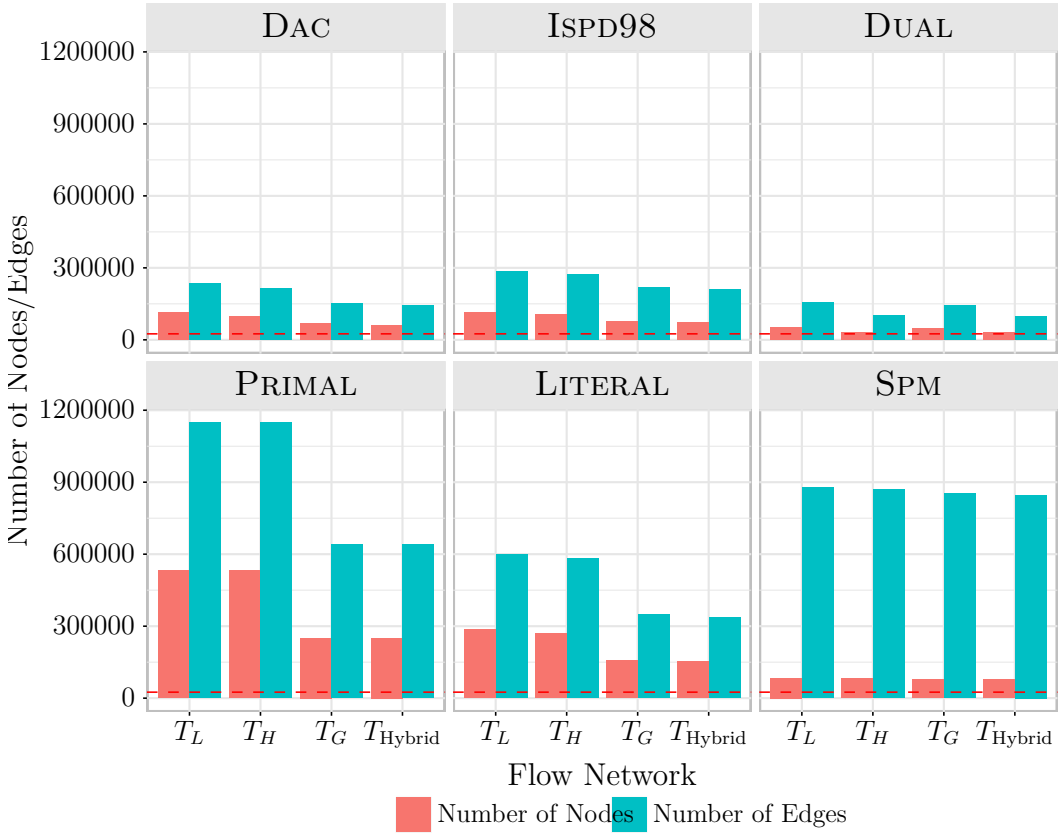

Figure 19: Comparison of the number of nodes and edges induced by flow problems of size $|V'| = 25000$ on our flow network for different benchmark types. The red dashed lines indicates 25000 nodes.

type by at least factor of 2 compared to T_L. Consequently, the speed-ups of our maximum flow algorithms on T_{Hybrid} are approximately to 2 on each benchmark type. Our GoldbergTarjan implementation profits most from our sparsification techniques. The algorithm is around 3 to 4 times faster on T_{Hybrid} than on T_L. Furhter, all algorithms perform best on T_{Hybrid}. In conclusion, our sparsification techniques not only reduces the size of the flow problems, it also significantly speed-ups the running time of several maximum flow algorithms.

In Table 1, we compare the absolute running times of the algorithms on our fastest flow network T_{Hybrid}. The Ibfs algorithm works best on large instances ($|V'| > 1000$). For smaller benchmarks ($|V'| \leq 1000$) BoykovKolmogorov and EdmondKarp are faster than the Ibfs algorithm. However, we are currently not able to use the Ibfs algorithm in our *flow*-based local search algorithm. The data structure of the algorithm is not optimized for multiple executions on different flow networks, because it did not implement an efficient strategy to reuse the allocated memory. The usage of the Ibfs algorithm in our framework leads to memory overflows. However, we currently work on a reimplementation of the algorithm such that it can be used. Therefore, we choose the BoykovKolmogorov maximum flow algorithm in combination with our flow network T_{Hybrid} in the following experiments.

6.4. Configuring the direct k-way Flow-Based Refinement

In this Section, we analyze the quality of our k-way flow-based refinement algorithm with different configurations on our parameter tuning benchmark subset (see Table 6). There are several configurations and tuning parameters which we have to evaluate:

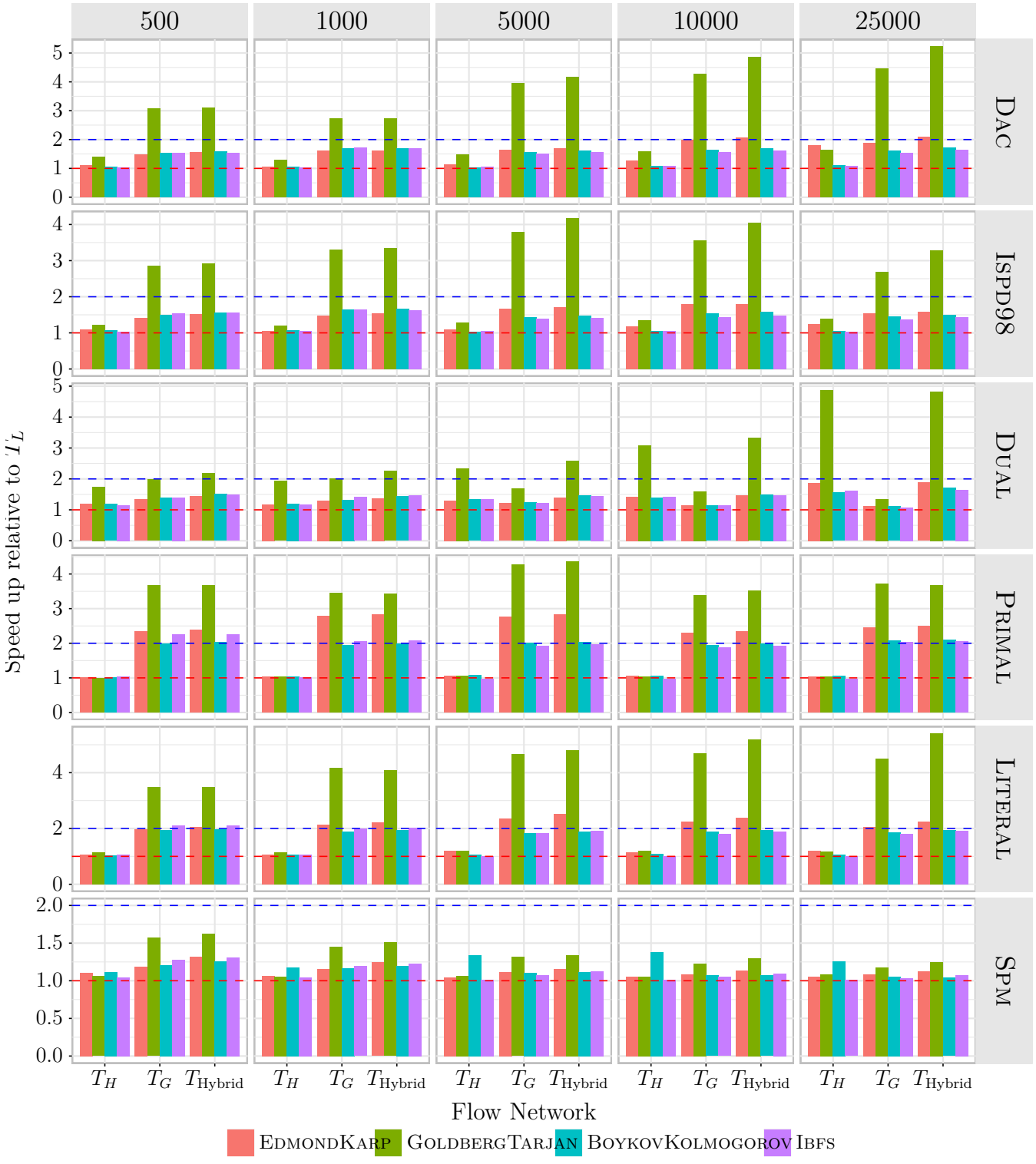

Figure 20: Speed-ups of our flow algorithms on different flow networks relative to execution on T_L for different problem sizes and types. The red resp. blue dashed line indicates a speed-up of 1 resp. 2.

$\lvert V' \rvert$	Ibfs $t[ms]$	BoykovKolmogorov $t[\%]$	GoldbergTarjan $t[\%]$	EdmondKarp $t[\%]$
500	0.81	+1.69	+38.4	**−5.57**
1000	**1.91**	+13.57	+45.44	+15.41
5000	**13.54**	+38.05	+103.94	+111.31
10000	**28.45**	+54.37	+148.3	+186.97
25000	**64.44**	+50.09	+147.82	+160.03

Table 1: Absolute running time of our maximum flow algorithms on flow network T_{Hybrid}. Note, all values in the table are in percentage relative to the running time of the Ibfs algorithm. In each line the fastest variant is marked bold.

- *Max-(F)low-Min-Cut* computations as *local search* algorithm
- *Adaptive Flow Iteration* parameter α' (see Section 3.3.2)
- *(M)ost Balanced Minimum Cut* heuristic (see Section 5.3)
- Combining *Max-(F)low-Min-Cut* computations with *(FM)* refinement

In the following, we denote a configuration e.g. with (+F,-M,-FM) which indicates which heuristic resp. technique is enabled (+) or disabled (−). The meaning of the abbreviations is explained in the enumeration above (see letters inside parentheses). We evaluate each configuration for $k \in \{2, 4, 8, 16, 32, 64, 128\}$, $\alpha' \in \{1, 2, 4, 8, 16\}$ and 10 different seeds ($\epsilon = 3\%$). We execute a *flow*-based *local search* on each level i with $i = 2^j$ ($j \in \mathbb{N}_+$). Further, we use KaHyPar-CA as reference [25] and refer to it as (-F,-M,+FM).

Config.	(+F,-M,-FM)		(+F,+M,-FM)		(+F,+M,+FM)		Constant128	
α'	Avg.[%]	$t[s]$	Avg.[%]	$t[s]$	Avg.[%]	$t[s]$	Avg.[%]	$t[s]$
1	−6.1	13.51	−5.62	14.22	0.23	15.19	0.32	67.38
2	−3.2	16.89	−2.08	18.23	0.74	17.97	0.62	139.21
4	−1.82	22.23	−0.2	24.29	1.21	22.5	1.03	274.6
8	−0.85	31.49	0.98	34.43	1.71	30.58	1.67	558.81
16	−0.2	48.66	1.75	53.23	2.21	45.04	2.44	1220.92
Ref.	(-F,-M,+FM)		6373.88	13.73				

Table 2: Table contains results for different configurations of our flow-based refinement framework for increasing α'. The quality in column *Avg.* is relative to our baseline configuration without the usage of flows.

The results are summarized in Table 2. The values in column *Avg* are improvements relative to our baseline configuration (-F,-M,+FM). The running time are absolute values in seconds. The first observation is that flows as single refinement strategy are not strong enough to outperform the *FM* heuristic. Our strongest configuration with $\alpha' = 16$ is 0.2% worse than the *FM* baseline. The running time scales nearly linear with parameter α'. Enabling the *Most Balanced Minimum Cut* heuristic significantly improves the quality compared to the configuration (+F,-M,-FM). The quality improvements are more significant for large α'. The larger the flow problem size, the larger is the number of different minimum (S, T)-cutsets and this increases the possibility to

Config.	(+F,-M,-FM)		(+F,+M,-FM)		(+F,+M,+FM)	
α'	M1 - Avg.[%]	M2 - Avg.[%]	M1 - Avg.[%]	M2 - Avg.[%]	M1 - Avg.[%]	M2 - Avg.[%]
1	−15.48	−6.1	−15.26	−5.62	0.14	0.23
2	−10.5	−3.2	−10.12	−2.08	0.36	0.74
4	−5.98	−1.82	−5.08	−0.2	0.67	1.21
8	−3.22	−0.85	−1.64	0.98	1.25	1.71
16	−1.52	−0.2	0.51	1.75	1.87	2.21
Ref.	(-F,-M,+FM)		6373.88			

Table 3: Comparison of quality of our framework with different source and sink set modeling approaches. M1 represents the approach of Sanders and Schulz [45] and M2 is our new variant proposed in Section 5.2.

find a feasible solution that respects the balanced constraint. Also it outperforms our baseline *FM* configuration for $\alpha' = 16$ by 1.75%. If we enable *FM* refinement in all levels where no flow is executed, we improve the solution quality by 2.21% (for $\alpha' = 16$). Also, the running time of this variant is faster than all previous flow configurations because we transfer more work to the *FM* refinement. Consequently, a block becomes faster *inactive* during the *active block scheduling* algorithm and this decreases the number of rounds of complete pairwise flow-based refinements. Our best configuration is (+F,+M,+FM) with $\alpha' = 16$. For further experiments, we refer to this variant as KaHyPar-MF.

6.5. Speed-Up Heuristics

At the end of Section 5.4, we presented several heuristics to prevent *unpromising* flow executions during *active block scheduling* ((R1)-(R3)). The main assumption is that only a minority of *Max-Flow-Min-Cut* computations lead to an improvement. To verify this assumption, we execute KaHyPar-MF in combination with different speed-up heuristics on our benchmark subset (see Table 7).
Table 4 summarizes the results of the experiment. The indices of the different variants of KaHyPar-MF describe which speed-up heuristic is enabled. On average, enabling all speed-up heuristics worsen the quality of KaHyPar-MF only by 0.07%. On the other hand, the framework is significantly faster by a factor of 2. In its final configuration KaHyPar-MF$_{(R1,R2,R3)}$ computes partitions with 2.41% better quality than KaHyPar-CA, while only incurring a slowdown of a factor of 2. In the following, we will denote our final configuration KaHyPar-MF$_{(R1,R2,R3)}$ as KaHyPar-MF.

6.6. Comparison with other Hypergraph Partitioner

Finally, we compare our new approach KaHyPar-MF with different state-of-the-art hypergraph partitioner on our full benchmark set. We excluded 194 instances of 3416 either because PaToH-Q could not allocate enough memory or other partitioners did not finish in time. The excluded instances are shown in Table 10.
Fig. 21 summarizes the results of the experiment. KaHyPar-MF produce on ≈ 70% of all benchmark instances the best partitions. It is followed by hMetis-R (14%), hMetis-K (11%),

Variant	Avg.[%]	Min.[%]	$t_{\text{flow}}[s]$	$t[s]$
KaHyPar-CA	7077.2	6820.17	-	29.26
KaHyPar-MF	−2.48	−2.13	51.76	81.02
KaHyPar-MF$_{(\text{R1})}$	−2.41	−2.05	41.21	70.47
KaHyPar-MF$_{(\text{R1,R2})}$	−2.4	−2.04	35.56	64.82
KaHyPar-MF$_{(\text{R1,R2,R3})}$	−2.41	−2.05	26.64	55.9

Table 4: Results of our flow-based refinement framework with different speedup heuristics.

KaHyPar-CA (2.4%), PaToH-Q (1.9%) and PaToH-D (1.4%). Comparing KaHyPar-MF individually with each partitioner, KaHyPar-MF produces better partitions than KaHyPar-CA, hMetis-R, hMetis-K, PaToH-Q, PaToH-Q on 96%, 80%, 82%, 95%, 95% of the benchmark instances. Especially on *VLSI* instances, KaHyPar-MF calculates significantly better partitions than all other hypergraph partitioners (see Dac and Ispd98 in Fig. 21).

Table 13 shows the running time of all partitioner for different benchmark types. The running time of KaHyPar-MF is within a factor of 2 slower than KaHyPar-CA and is comparable to the running time of hMetis-K.

Partitioner	Running Time $t[s]$						
	All	Dac	Ispd98	Primal	Literal	Dual	Spm
KaHyPar-MF	31.98	505.75	21.1	28.71	41.72	45.85	24.19
KaHyPar-CA	16.19	368.97	12.35	14.41	21.56	33.47	10.29
hMetis-R	43.95	446.36	29.03	26.49	49.97	99.55	33.6
hMetis-K	33.47	240.92	23.18	18.65	33.62	69.83	28.2
PaToH-Q	3.36	28.34	1.89	2.8	3.38	5.29	2.81
PaToH-D	0.7	6.45	0.35	0.44	0.55	1.36	0.64
Partitioner	Running Time $t[s]$						
	All	Dac	Ispd98	Primal	Literal	Dual	Spm
KaHyPar-MF	62.24	637.58	22.29	71.63	140.84	106.24	29.61
KaHyPar-CA	31.05	368.97	12.35	32.91	64.65	68.27	13.91
hMetis-R	79.23	446.36	29.03	66.25	142.12	200.36	41.79
hMetis-K	57.86	240.92	23.18	44.23	94.89	125.55	35.95
PaToH-Q	5.89	28.34	1.89	6.9	9.24	10.57	3.42
PaToH-D	1.22	6.45	0.35	1.12	1.58	2.87	0.77

Table 5: Comparing the average running time of KaHyPar-MF with KaHyPar-CA and other hypergraph partitioners.

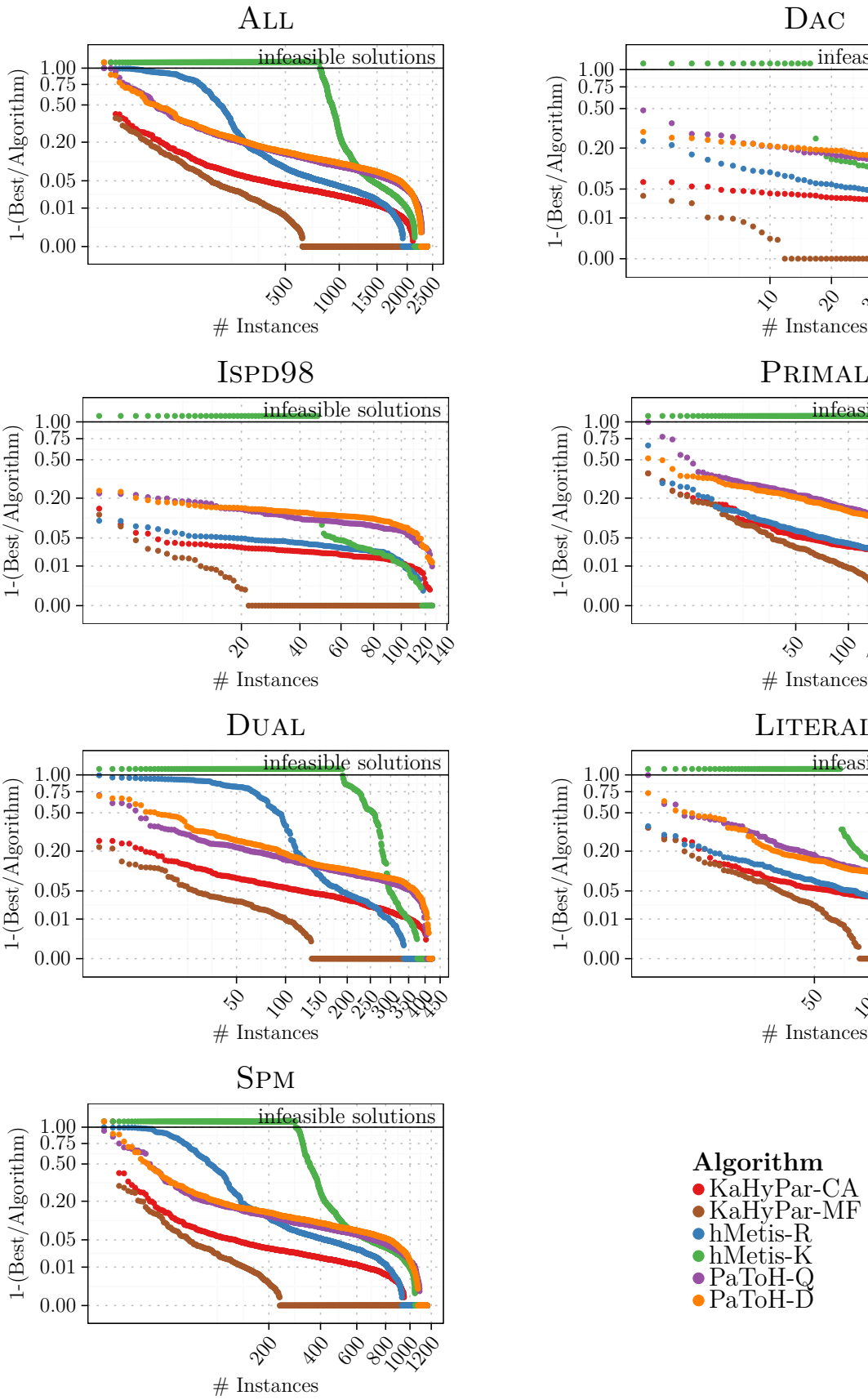

Figure 21: Min-Cut performance plots comparing KaHyPar-MF with KaHyPar-CA and other systems. Plots are explained in Section 6.2.

7. Conclusion

In this thesis, we present a novel *local search* technique based on *Max-Flow-Min-Cut* computations for multilevel hypergraph partitioning. We have integrated our *flow*-based *local search* algorithm into the n-level hypergraph partitioner *KaHyPar* and show that in combination with the *FM* heurisitc our new approach produces the best partitions for a wide range of applications.
On the road to a practical implementation, we introduce several techniques to sparsify the state-of-the-art hypergraph flow network representation [33]. The network consists of many edges with *infinite* capacity. We present several theoretical results, which allows us to remove such edges or even to remove nodes, if all *incoming* and *outgoing* edges have a capacity equal to *infinity*. The results are of independent interest, because they are also applicable on general flow networks with *infinite* capacity edges. As a consequence, we can remove all hypernodes from the flow network of a hypergraph, because all *incoming* and *outgoing* edges have *infinite* capacity in the corresponding flow network [33]. However, the removal of a hypernode in the flow network introduce new edges between all incident nodes, which number depends on the degree of a vertex. We show that we can minimize the number of edges, if the degree of a hypernode is smaller than or equal to 3. Further, we show that each weighted undirected graph can be modeled as a weighted directed graph with the same *min-cut* property. The transformation introduces new nodes and edges in the corresponding weighted directed graph. However, we have shown that the transformation of an undirected edge results in the same structure as the transformation of hyperedges of size 2 in the hypergraph flow network. As a consequence, we can model each hyperedge e with $|e| = 2$ as a undirected flow edge. We combine both techniques in a *Hybrid*-Network and show that maximum flow algorithms are up to a factor of 3 faster compared to the execution on the *Lawler*-Network [33] on real-world hypergraph benchmarks.
Our *flow*-based refinement framework is based on the ideas of Sanders and Schulz [45] (developed for multilevel graph partitioning). However, we generalize many results such that they are applicable on hypergraph partitioning. We show how one can improve a given bipartition of a hypergraph by executing a *Max-Flow-Min-Cut* computation only on a subhypergraph. Further, if we use the *FM* heuristic, the value of the cut after moving a node from one block to another can be calculated with the gain value of the move and the cut before. We show how the source and sink set of the corresponding flow problem can be modeled such that the value of the cut of the improved bipartition can be calculated with the value of a maximum (S, T)-flow. Additionally, we explain how one can find all minimum (s, t)-cutsets with one maximum (s, t)-flow calculation on hypergraphs.
We integrated our framework into the n-level hypergraph partitioner *KaHyPar*. We embed our *flow*-based *local search* algorithm into an *active block scheduling* refinment strategy. The algorithm works in rounds and in each round we start a *flow-based* refinement between each adjacent block. The algorithm terminates if no hypernode changed its block in a round. The flow problem is build around the cut of two adjacent blocks. The sizes of the flow problems are chosen adaptively. If a *flow* computation on two blocks yields an improvement, we increase the flow problem size, otherwise we decrease it. Additionally, we try to automatically balance the partition after *Max-Flow-Min-Cut* computation by using the *Most Balanced Minimum Cut* heuristic. The *active block scheduling* refinement is executed in $\log n$ levels of the multilevel hierarchy. In the remaining levels, where no flow is performed, the classical *FM* heuristic is used to improve the quality of a partition. An observation during implementation was that only a minority of the *Max-Flow-Min-Cut* computations leads to an improvement in quality. Therefore, we implement several speed-up heuristics which prevents the execution of unpromis-

ing pairwise *flow* refinements.
The new configuration KaHyPar-MF produces on 95% of our benchmark instances better partitions than our old configuration KaHyPar-CA. On average the solution quality is 2% better, while only incurring a slowdown by a factor of 2. In comparison with 5 different state-of-the-art hypergraph partitioners, KaHyPar-MF produces on 70% of 3222 benchmark instances the best partitions. Moreover, our partitioner has a comparable running time to the direct k-way version of *hMetis* and outperforms it on 82% of the benchmark instances.

7.1. Future Work

Due to the novelty of the approach, there is a lot of potential in optimizing our basic framework. We made a trade-off between time and quality to obtain a *High-Quality n-level Hypergraph Partitioner* which runs in reasonable time. The quality mainly depends on the number of flow executions throughout the multilevel hierarchy. The number of flow executions, which we can do in reasonable time, depends on the running time of the flow algorithm and the size of the flow problem. Optimizing those two basic building blocks of the framework will allow us to achieve better quality in the same amount of time.
The flow network of a hypergraph proposed by Lawler [33] has a bipartite structure. Because of this structural regularity, there might be other more specialized flow algorithms which run faster on these types of networks. Therefore, a useful work would be to evaluate many different maximum flow algorithms on our benchmark set. Further, one could investigate if it is possible to maintain the whole flow network over the multilevel hierarchy without explicitly setting up the flow network before each flow execution. Also, it would be interesting if information from previous flow calculations can be used to speed-up the current flow calculation. Pistorius [43] described an algorithm which implicitly executes EdmondKarp on a hypergraph using labels on the hypernodes. In our first version of the framework, we also used a similar technique and implicitly executes a flow algorithm on an implicit representation of the underlying network. During experiments, it turned out that the explicit representation was up to a factor of 2-3 faster than the implicit version. We encountered several reasons for that behavior:

(i) Our flow network represents a subhypergraph of the original hypergraph. Iterating over the edges of a node means to iterate also over hypernodes which are not part of the flow problem and therefore have to be ignored.

(ii) There are many different cases when we want to increase the flow along an *augmenting path*.

(iii) Many labels have to be introduced which lead to a large number of main memory accesses.

(iv) Also the implicit flow network is not flexible enough. Developing a new technique to sparsify the flow networ would require a new implementation of the implicit flow network.

In Section **??** and 6.5 we show that with three simple speed up heuristics our *flow*-based refinement framework is up to a factor of 2 faster with comparable quality. Therefore, it would be beneficial to further increase the effectiveness ratio of the flow computation by introducing more heuristics.
It is also possible to further sparsify the flow network. Assume there exists two hypernodes v_1 and v_2 with $d(v_1) = 3$ and $d(v_2) = 4$. Further, $|I(v_1) \cap I(v_2)| = 3$ which means that in each hyperedge e where $v_1 \in e$ also $v_2 \in e$ except for one hyperedge e' where $v_2 \in e'$ and $v_1 \notin e'$. We remove all hypernodes with $d(v) \leq 3$ in our *hybrid* flow network. Consequently, we would remove v_1 and insert a clique between all incident hyperedges. However, v_2 is part of the flow network and induce $2d(v_2) = 8$ edges because $d(v_2) > 3$. Alternatively, we can remove v_2 and expand the clique between all hyperedges of $I(v_1)$ with e'. In that case, we have to insert

an edge from each hyperedge in $I(v_1)$ to e' and vice versa. Since $|I(v_1)| = d(v_1) = 3$ only $2|I(v_1)| = 6$ edges are inserted and we can remove one hypernode. In general, an expansion of a k-clique to a $(k+i)$-clique induced ik edges from the k nodes already contained in the clique to the i new nodes and $i(k+1-1)$ edges from the i new nodes to the k nodes in the clique. If we can remove a hypernode from the flow network by expanding a k-clique between hyperedge nodes to a $(k+i)$-clique, it is beneficial if the following inequality holds

$$ik + i(k+i-1) = i^2 + 2ki - i \leq 2(k+i)$$

The inequality is only satisfied for $i = 1$. In this case, we can exactly remove 2 edges and 1 node from the flow network. A possible algorithm could be to sort the hypernodes according to the degree and for each hypernode store a clique label which indicates between how many incident hyperedges already exist a clique. Afterwards, we iterate over the hypernodes and if we remove a hypernode, we have to update the clique label of all hypernodes in the intersection of the currently inserted clique. We iterate over the hypernodes until none of the hypernodes could be removed anymore. However, we didn't find an efficient implementation of the above-described algorithm. The algorithm requires a fast calculation between the intersection of several hyperedges. An explicit construction of the intersection hypergraph would occupy too much memory.

References

[1] Y. Akhremtsev, T. Heuer, P. Sanders, and S. Schlag. Engineering a direct k-way Hypergraph Partitioning Algorithm. In *2017 Proceedings of the 19th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 28–42. SIAM, 2017.

[2] C. J. Alpert. The ISPD98 Circuit Benchmark Suite. In *Proceedings of the 1998 International Symposium on Physical Design*, pages 80–85. ACM, 1998.

[3] C. J. Alpert and A. B. Kahng. Recent Directions in Netlist Partitioning: a Survey. *Integration, the VLSI journal*, 19(1-2):1–81, 1995.

[4] R. Andersen and K. J. Lang. An Algorithm for Improving Graph Partitions. In *Proceedings of the 19th annual ACM-SIAM symposium on Discrete algorithms*, pages 651–660. Society for Industrial and Applied Mathematics, 2008.

[5] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner. *Graph Partitioning and Graph Clustering*, volume 588. American Mathematical Society, 2013.

[6] A. Belov, M. Heule, D. Diepold, and M. Järvisalo. The Application and the Hard Combinatorial Benchmarks in Sat Competition 2014. *Proceedings of SAT Competition*, pages 81–82, 2014.

[7] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE transactions on pattern analysis and machine intelligence*, 26(9):1124–1137, 2004.

[8] T. N. Bui and C. Jones. Finding Good Approximate Vertex and Edge Partitions is NP-Hard. *Information Processing Letters*, 42(3):153–159, 1992.

[9] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent Advances in Graph Partitioning. In *Algorithm Engineering*, pages 117–158. Springer, 2016.

[10] U. V. Catalyurek and C. Aykanat. Hypergraph-Partitioning-based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Transactions on parallel and distributed systems*, 10(7):673–693, 1999.

[11] B. V. Cherkassky. A Fast Algorithm for Computing Maximum Flow in a Network. *Collected Papers*, 3:90–96, 1994.

[12] B. V. Cherkassky and A. V. Goldberg. On Implementing the Push-Relabel Method for the Maximum Flow Problem. *Algorithmica*, 19(4):390–410, 1997.

[13] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[14] U. Derigs and W. Meier. Implementing Goldberg's Max-Flow-Algorithm — A Computational Investigation. *Mathematical Methods of Operations Research*, 33(6):383–403, 1989.

[15] S. Dutt and W. Deng. Vlsi circuit partitioning by cluster-removal using iterative improvement techniques. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 194–200. IEEE Computer Society, 1997.

[16] J. Edmonds and R. M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM (JACM)*, 19(2):248–264, 1972.

[17] C. M. Fiduccia and R. M. Mattheyses. A Linear-Time Heuristic for improving Network Partitions. In *Papers on Twenty-five years of electronic design automation*, pages 241–247. ACM, 1988.

[18] L. R. Ford and D. R. Fulkerson. Maximal Flow through a Network. *Canadian journal of Mathematics*, 8(3):399–404, 1956.

[19] L. R. Ford Jr and D. R. Fulkerson. *Flows in Networks*. Princeton university press, 2015.

[20] S. Fortunato. Community Detection in Graphs. *Physics reports*, 486(3):75–174, 2010.

[21] A. Goldberg, S. Hed, H. Kaplan, R. Tarjan, and R. Werneck. Maximum flows by incremental breadth-first search. *Algorithms–ESA 2011*, pages 457–468, 2011.

[22] A. V. Goldberg, S. Hed, H. Kaplan, P. Kohli, R. E. Tarjan, and R. F. Werneck. Faster and more dynamic maximum flow by incremental breadth-first search. In *Algorithms-ESA 2015*, pages 619–630. Springer, 2015.

[23] A. V. Goldberg and R. E. Tarjan. A new Approach to the Maximum-Flow Problem. *Journal of the ACM (JACM)*, 35(4):921–940, 1988.

[24] T. Heuer. *Engineering Initial Partitioning Algorithms for direct k-way Hypergraph Partitioning*. PhD thesis, Karlsruher Institut für Technologie (KIT), 2015.

[25] T. Heuer and S. Schlag. Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 75. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[26] M. Holtgrewe, P. Sanders, and C. Schulz. Engineering a scalable High Quality Graph Partitioner. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[27] T. C. Hu and K. Moerder. Multiterminal Flows in a Hypergraph. In T. Hu and E. Kuh, editors, *VLSI Circuit Layout: Theory and Design*, chapter 3, pages 87–93. IEEE Press, 1985.

[28] A. B. Kahn. Topological Sorting of Large Networks. *Communications of the ACM*, 5(11):558–562, 1962.

[29] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel Hypergraph Partitioning: Applications in VLSI Domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, 1999.

[30] G. Karypis and V. Kumar. Multilevel k-way Hypergraph Partitioning. *VLSI design*, 11(3):285–300, 2000.

[31] B. Krishnamurthy. An Improved Min-Cut Algorithm for Partitioning VLSI Networks. *IEEE Transactions on Computers*, (5):438–446, 1984.

[32] K. Lang and S. Rao. A Flow-Based Method for improving the Expansion or Conductance of Graph Cuts. In *IPCO*, volume 4, pages 325–337. Springer, 2004.

[33] E. L. Lawler. Cutsets and Partitions of Hypergraphs. *Networks*, 3(3):275–285, 1973.

[34] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. Springer Science & Business Media, 2012.

[35] H. Liu and D. Wong. Network Flow Based Multi-way Partitioning with Area and Pin Constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(1):50–59, 1998.

[36] Z. Á. Mann and P. A. Papp. Formula Partitioning Revisited. 2014.

[37] K. Menger. Zur Allgemeinen Kurventheorie. *Fundamenta Mathematicae*, 10(1):96–115, 1927.

[38] M. E. Newman. Analysis of Weighted Networks. *Physical review E*, 70(5):056131, 2004.

[39] V. Osipov and P. Sanders. n-Level Graph Partitioning. In *European Symposium on Algorithms*, pages 278–289. Springer, 2010.

[40] D. A. Papa and I. L. Markov. Hypergraph Partitioning and Clustering, 2007.

[41] S. B. Patkar, H. Sharma, and H. Narayanan. Efficient Network Flow Based Ratio-Cut Netlist Hypergraph Partitioning. *WSEAS Transactions on Circuits and Systems*, 3(1):47–53, 2004.

[42] J.-C. Picard and M. Queyranne. On the Structure of all Minimum Cuts in a Network and Applications. *Combinatorial Optimization II*, pages 8–16, 1980.

[43] J. Pistorius and M. Minoux. An Improved Direct Labeling Method for the Max–Flow Min–Cut Computation in Large Hypergraphs and Applications. *International Transactions in Operational Research*, 10(1):1–11, 2003.

[44] L. A. Sanchis. Multiple-Way Network Partitioning. *IEEE Transactions on Computers*, 38(1):62–81, 1989.

[45] P. Sanders and C. Schulz. Engineering Multilevel Graph Partitioning Algorithms. In *ESA*, volume 6942, pages 469–480. Springer, 2011.

[46] S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz. k-way Hypergraph Partitioning via n-Level Recursive Bisection. In *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 53–67. SIAM, 2016.

[47] D. D. Sleator and R. E. Tarjan. A Data Structure for Dynamic Trees. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 114–122. ACM, 1981.

[48] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[49] 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*, pages 774–782. ACM, 2012.

[50] D. B. West et al. *Introduction to Graph Theory*, volume 2. Prentice hall Upper Saddle River, 2001.

[51] H. H. Yang and D. Wong. Efficient Network Flow Based Min-Cut Balanced Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(12):1533–1540, 1996.

[52] Z. Zhao, L. Tao, and Y. Zhao. An Effective Algorithm for Multiway Hypergraph Partitioning. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 49(8):1079–1092, 2002.

A. Benchmark Instances

A.1. Parameter Tunning Benchmark Set

Type	Num	min $\|V\|$	Avg.$\|V\|$	max $\|V\|$	min $\|E\|$	Avg.$\|E\|$
Ispd98	5	32498	49049	69429	34826	52202
Primal	5	53919	90467	163622	245440	414577
Literal	5	96430	141622	283720	140968	323388
Dual	5	100384	297768	1070757	34317	85669
Spm	5	12328	34129	74104	12328	34129
Type	max $\|E\|$	Avg.$\|e\|$	Med.$\|e\|$	Avg.$d(v)$	Med.$d(v)$	Avg.$\frac{\|E\|}{\|V\|}$
Ispd98	75196	3.79	2	4.04	3.57	1.06
Primal	629461	2.56	2.3	11.74	6.54	4.58
Literal	629461	2.56	2.3	5.85	3.25	2.28
Dual	229544	8.05	6.03	2.32	2	0.29
Spm	74104	20.91	19.92	20.91	17.87	1

Table 6: Statistical summary of the parameter tunning instances.

A.2. Benchmark Subset

Type	Num	min $\|V\|$	Avg.$\|V\|$	max $\|V\|$	min $\|E\|$	Avg.$\|E\|$
Dac	5	522482	708389	917944	511685	697951
Ispd98	10	53395	110344	210613	60902	119535
Primal	30	7729	141143	1613160	29194	632173
Literal	30	15458	281238	3226318	29194	632173
Dual	30	29194	632173	6429816	7729	141143
Spm	60	11028	64765	1000005	4371	59589
Type	max $\|E\|$	Avg.$\|e\|$	Med.$\|e\|$	Avg.$d(v)$	Med.$d(v)$	Avg.$\frac{\|E\|}{\|V\|}$
Dac	898001	3.37	2	3.32	3.18	0.99
Ispd98	201920	3.87	2.08	4.2	3.67	1.08
Primal	6429816	2.58	2.2	11.54	7.39	4.48
Literal	6429816	2.58	2.2	5.79	3.78	2.25
Dual	1613160	11.54	7.39	2.58	2.2	0.22
Spm	1000005	16.25	12.95	14.95	12.58	0.92

Table 7: Statistical summary of the benchmark subset instances.

A.3. Full Benchmark Set

Type	Num	min $\|V\|$	Avg.$\|V\|$	max $\|V\|$	min $\|E\|$	Avg.$\|E\|$
Dac	10	522482	888090	1360217	511685	876629
Ispd98	18	12752	59801	210613	14111	64240
Primal	92	7502	111371	1621762	28770	649991
Literal	92	15004	221981	3226318	28770	649991
Dual	92	28770	649991	13378617	7502	111371
Spm	184	10000	56930	9845725	163	52709
Type	max $\|E\|$	Avg.$\|e\|$	Med.$\|e\|$	Avg.$d(v)$	Med.$d(v)$	Avg.$\frac{\|E\|}{\|V\|}$
Dac	1340418	3.41	2	3.37	3.27	0.99
Ispd98	201920	3.83	2.05	4.11	3.52	1.07
Primal	13378617	2.74	2.31	16.01	8.12	5.84
Literal	13378617	2.74	2.31	8.03	3.65	2.93
Dual	1621762	16.01	8.12	2.74	2.31	0.17
Spm	6920306	15.72	12.15	14.56	10.99	0.93

Table 8: Statistical summary of the full benchmark set instances.

A.4. Excluded Test Instances

Hypergraph	2	4	8	16	32	64	128
10pipe-q0-k.dual				△	△	△	❍△
10pipe-q0-k.primal	□	□	□	□	□	□	□
11pipe-k.dual	△	❍△	❍△	❍△	❍△	❍△	❍△
11pipe-k				❍	❍	❍	❍
11pipe-k.primal	□	□	□	□	□	□	❍□
11pipe-q0-k.dual					△	❍△	❍△
11pipe-q0-k.primal	□	□	□	□	□	□	□
9dlx-vliw-at-b-iq3.dual							△
9dlx-vliw-at-b-iq3.primal	□	□	□	□	□	□	□
9vliw-m-9stages-iq3-C1-bug7.dual	△	●❍△	●❍△	●❍△	●❍△	●❍△	●❍△
9vliw-m-9stages-iq3-C1-bug7	△	△	●❍△	●❍△	●❍△	●❍□△	●❍□△
9vliw-m-9stages-iq3-C1-bug7.primal	△	△		△	❍△	❍△	❍△
9vliw-m-9stages-iq3-C1-bug8.dual	△	●❍△	●❍△	●❍△	●❍△	●❍△	●❍△
9vliw-m-9stages-iq3-C1-bug8	△	△	●❍△	●❍△	●❍△	●❍□△	●❍□△
9vliw-m-9stages-iq3-C1-bug8.primal	△	△		△	❍△	❍△	❍△
blocks-blocks-37-1.130-NOTKNOWN.dual		❍	●❍	●❍	●❍	●❍	●❍△
blocks-blocks-37-1.130-NOTKNOWN		□	□	□	□	□	□
blocks-blocks-37-1.130-NOTKNOWN.primal	□	□	□	□	□	□	□
E02F20.dual							❍
E02F22.dual						❍	❍
openstacks-p30-3.085-SAT.primal	□	□	□	□	□	□	□
openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30-3.025-NOTKNOWN.primal	□	□	□	□	□	□	□
openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30-3.085-SAT.primal	□	□	□	□	□	□	□

q-query-3-L100-coli.sat.dual							△
q-query-3-L150-coli.sat.dual						△	△
q-query-3-L200-coli.sat.dual					△	△	△
q-query-3-L80-coli.sat.dual							△
transport-transport-city-sequential-25nodes-1000size-3degree-100mindistance-3trucks-10packages-2008seed.030-NOTKNOWN.dual							△
transport-transport-city-sequential-25nodes-1000size-3degree-100mindistance-3trucks-10packages-2008seed.050-NOTKNOWN.primal	□			□			□
velev-vliw-uns-2.0-uq5.dual			△	△	△	△	△
velev-vliw-uns-2.0-uq5.primal	□	□	□	□	□	□	□
velev-vliw-uns-4.0-9.dual					△	△	△
velev-vliw-uns-4.0-9.primal	□	□	□	□	□	□	□
192bit	□			□			
appu						❍	❍
ESOC	□	□			□	❍□	□
human-gene2					❍△	❍△	❍△
IMDB				△	△	△	△
kron-g500-logn16		△	△	△	△	❍△	❍△
Rucci1					□		
sls	□	□	□	❍□	❍□	❍□	❍□
Trec14							❍

△ :	KaHyPar-CA/KaHyPar-MF exceeded time limit
● :	hMetis-R exceeded time limit
❍ :	hMetis-K exceeded time limit
□ :	PaToH-Q memory allocation error

Table 10: Instances excluded from the full benchmark set evaluation.

B. Removing Infinite Weight Nodes of the Vertex Separator Problem

Finding a minimum-weight (s,t)-cutset of a hypergraph can be reduced to the problem of finding a minimum-weight (s,t)-vertex separator of the bipartite graph representation of the hypergraph. The weight of the hyperedge nodes is the weight of the corresponding hyperedge and the weight of all hypernodes is *infinity*. We can calculate a minimum-weight (s,t)-vertex separator with a maximum (s,t)-flow calculation on the Lawler-Transformation $T_L(H)$. In Section 4.1 we have shown how to remove a hypernode v from $T_L(H)$ by adding a biclique between all outgoing hyperedges e'' and all incoming hyperedges e' with $e \in I(v)$. The correctness of the transformation follows with Lemma 4.1. The inserted biclique induce exactly $d(v)^2$ edges where $d(v)$ is the degree of hypernode v. Since the underlying problem is to find a minimum-weight (s,t)-vertex separator, we can prove that $d(v)(d(v)-1)$ edges are sufficient to model the problem equivalent. The problem is illustrated in Fig. 22. In the following, we show that we can remove the red edges which are exactly the edges between two equivalent hyperedge nodes.

Lemma B.1. *Let $G = (V, E, c)$ be an undirected graph where a node v exists with $c(v) = \infty$. We can find a minimum-weight (s,t)-vertex separator of G (weight must be smaller than ∞) with graph G' which is the graph without node v and having a clique between all adjacent nodes of v.*

Proof. We will show that each (s,t)-vertex separator of G is also a (s,t)-vertex separator of G' and vice versa. Since, the weight of each node in both network is the same, a minimum-weight (s,t)-vertex separator of G is also a minimum-weight (s,t)-vertex separator of G' and vice versa. We will call an edge of G', which is part of the inserted clique, a *shortcut* edge.
Let $V' \subseteq V$ be a (s,t)-vertex separator of G with $c(V') < \infty$. Assume after removing all $u \in V'$ from G' there exists still a path from s to t. If the path not contains any *shortcut* edge, the same path must connect s and t in G because we have removed all $u \in V'$ from G'. This is a contradiction that V' is a vertex separator of G. Assume the path contains a *shortcut* edge (u,w) of G'. Because $c(V') < \infty$ and $c(v) = \infty$, it follows that $v \notin V'$. We can replace (u,w) with two edges (u,v) and (v,w) and obtain a path connecting s and t in G. This is also a contradiction that V' is a vertex separator of G.
The same argumentation holds, if we want to show that each (s,t)-vertex separator of G' is a (s,t)-vertex separator of G. If the path P connecting s and t in G contains the removed node v with edges $(u,v) \in P$ and $(v,w) \in P$ we can replace the two edges with the *shortcut* edge (u,w) of G'. The resulting path connects s and t in G'. This is a contradiction that V' is a vertex separator of G'. □

Lemma B.1 can be used to remove the *infinity* capacity edges between the same hyperedge nodes (see red edges in Fig. 22). Instead of removing a hypernode v of $T_L(H)$ with Lemma 4.1, we can apply Lemma B.1 on a hypernode v of the bipartite graph representation of the hypergraph. Afterwards, we can use the vertex separator transformation (see Defintion 2.5) to obtain a flow network in which the value of a maximum (s,t)-flow is equal with the weight of a minimum-weight (s,t)-vertex separator. Both networks on the right side of Fig. 22 are equivalent, but a removal of a hypernode now induce $d(v)(d(v)-1)$ edges instead of $d(v)^2$ edges in the new network.

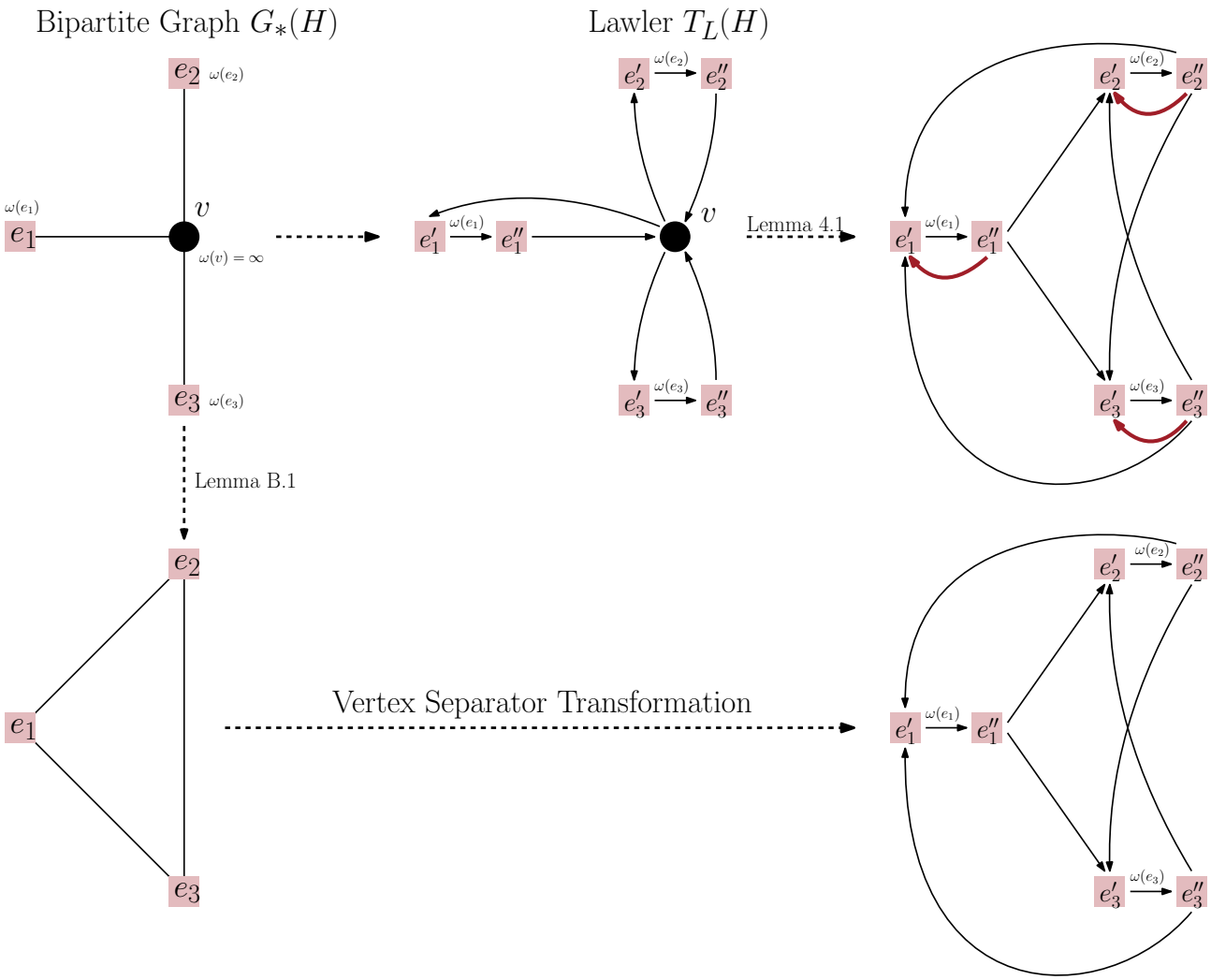

Figure 22: Illustration of the technique to remove a hypernode such that the removal induce $d(v)(d(v)-1)$ edges instead of $d(v)^2$ edges in the corresponding flow network. The edges without an explicit capacity are *infinite* capacity edges. The transformation on the top of the figure illustrates the technique presented in Section 4.1. Using Lemma B.1 and the vertex separator transformation (see Defintion 2.5) results in a equivalent flow network without the red highlighted edges.

C. Detailed Flow Network and Algorithm Evaluation

Instance	$\|V'\|$	GoldbergTarjan T_{Hybrid} $t[ms]$	GoldbergTarjan T_G $t[\%]$	GoldbergTarjan T_H $t[\%]$	GoldbergTarjan T_L $t[\%]$	EdmondKarp T_{Hybrid} $t[\%]$	EdmondKarp T_G $t[\%]$	EdmondKarp T_H $t[\%]$	EdmondKarp T_L $t[\%]$
All	500	**0.77**	+7.12	+50.83	+64.67	+46.57	+51.3	+200.56	+256.9
	1000	**2.2**	+5.01	+55.31	+67.38	+26.02	+29.76	+153.55	+206.75
	5000	28.6	+5.76	+50.26	+69.07	**−3.49**	+6.78	+99.42	+156.17
	10000	81.64	+7.85	+42.73	+63.8	**−13.48**	+4.83	+71.78	+132.82
	25000	167.56	+14.8	+37.09	+69.27	**−4.7**	+30.35	+82.25	+172.98
Dac	500	**0.19**	+4.71	+41.66	+55.51	+79.35	+79.75	+298.08	+454.1
	1000	**0.46**	+0.05	+52.89	+59.87	+97.75	+97.69	+315.35	+437.36
	5000	**4.39**	+3.42	+51.14	+69.34	+39.03	+47.39	+294.76	+480.9
	10000	**14.27**	+3.52	+61.62	+106.02	+12.8	+27.75	+245.63	+446.98
	25000	**44.44**	+11.51	+16.04	+108.9	+63.4	+91.42	+424.27	+752.95
Ispd98	500	**0.37**	+7.37	+38.77	+52.08	+53.92	+57.79	+270.94	+351.04
	1000	**0.87**	+4.8	+46.05	+54.38	+41.35	+43.38	+295.43	+374.43
	5000	**10.51**	+1.69	+54.66	+70.69	+40.07	+53.53	+353.41	+483.79
	10000	**41.82**	+1.02	+51.86	+80.43	+84.99	+109.85	+456.33	+649.18
	25000	**271.64**	+2.4	+26.77	+57.93	+109.79	+156.16	+398.97	+589.37
Dual	500	**0.14**	+7.67	+21.17	+43.25	+84.81	+103.57	+129.67	+302.37
	1000	**0.29**	+6.19	+17.48	+37.63	+94.91	+117.11	+127.92	+338.95
	5000	**1.94**	+14.75	+7.02	+39.36	+109.27	+221.2	+130.41	+439.71
	10000	**4.19**	+26.87	+3.37	+46.59	+131.25	+383.05	+148.38	+668
	25000	**13.02**	+68.6	+0.5	+88.47	+181.61	+907.16	+179.14	+1259.26
Primal	500	3.6	+1.45	+134.06	+137.82	−30.48	**−30.82**	+157.29	+154.56
	1000	10.65	+1.42	+173.18	+182.59	−38.78	**−39.26**	+103.91	+109.21
	5000	184.72	+1.88	+166.77	+181.9	**−60.88**	−60.24	+62.12	+70.09
	10000	290.9	+1.66	+122.2	+133.96	**−55.38**	−53.76	+50.44	+56.67
	25000	1002.32	+1.43	+142.23	+148.6	−56.54	**−57.04**	+53.74	+59.62
Literal	500	**0.94**	+4.42	+95.12	+106.26	+23.37	+24.05	+273.61	+330.97
	1000	3.06	+3.54	+107.09	+120.36	−12.02	**−13.64**	+215.13	+260.24
	5000	38.66	+6.67	+110.25	+152.32	**−29.59**	−27.69	+180.27	+238.17
	10000	122.71	+6.57	+106.82	+138.04	**−39.43**	−32.93	+160.96	+214.77
	25000	266.94	+10.03	+87.48	+124.75	**−36.29**	−23.41	+191.07	+243.8
Spm	500	**0.98**	+11.65	+19.8	+32.33	+104.15	+111.46	+213.59	+232.13
	1000	**3.23**	+7.7	+16.29	+24.21	+65.42	+71.86	+135.55	+149.33
	5000	**54.77**	+3.81	+10.65	+15.63	+9.93	+12	+38.83	+47.28
	10000	217.75	+4.41	+7.62	+13.47	**−24.96**	−20.83	−7.96	−2.78
	25000	229.09	+3.88	+7.4	+12.9	**−20.77**	−15.81	−8.61	−0.81

Table 11: Running time comparison of maximum flow algorithms on different flow networks. Note, all values in the table are in percentage relative to Goldberg-Tarjan on flow network T_{Hybrid}. In each line the fastest variant is marked bold.

D. Effectiveness Tests for Flow Configurations

To evaluate the effectiveness of our configurations presented in Section 6.4 we give each configuration the same amount of time to produce as many as possible partitions of a hypergraph H for a given k. We define $t_{H,k}$ which is the maximum partition time of a configuration to partition H in k blocks. If we execute a configuration on a hypergraph H for a given k and α' the time to produce as many as possible partitions is restricted by $3t_{H,k}$. We sum up the partition times during execution and if that sum plus the current average partition time would exceed $3t_{H,k}$ we perform the next run with a certain probability such that the expected running time is $3t_{H,k}$. The effectiveness tests were proposed by Sanders and Schulz [45]. The results of the tests mirrors our results of Section 6.4.

Config.	(+F,-M,-FM)	(+F,+M,-FM)	(+F,+M,+FM)
α'	Avg.[%]	Avg.[%]	Avg.[%]
1	−15.48	−15.22	0.14
2	−10.53	−10.11	0.36
4	−6	−5.13	0.66
8	−3.24	−1.68	1.24
16	−1.7	0.44	1.82
Ref.	(-F,-M,+FM)	6374.42	

Table 12: Table contains results of the effectiveness test for different configurations of our flow-based refinement framework for increasing α'. The quality in column *Avg.* is relative to our baseline configuration without the usage of flows.

E. Detailed Speedup Heuristic Evaluation

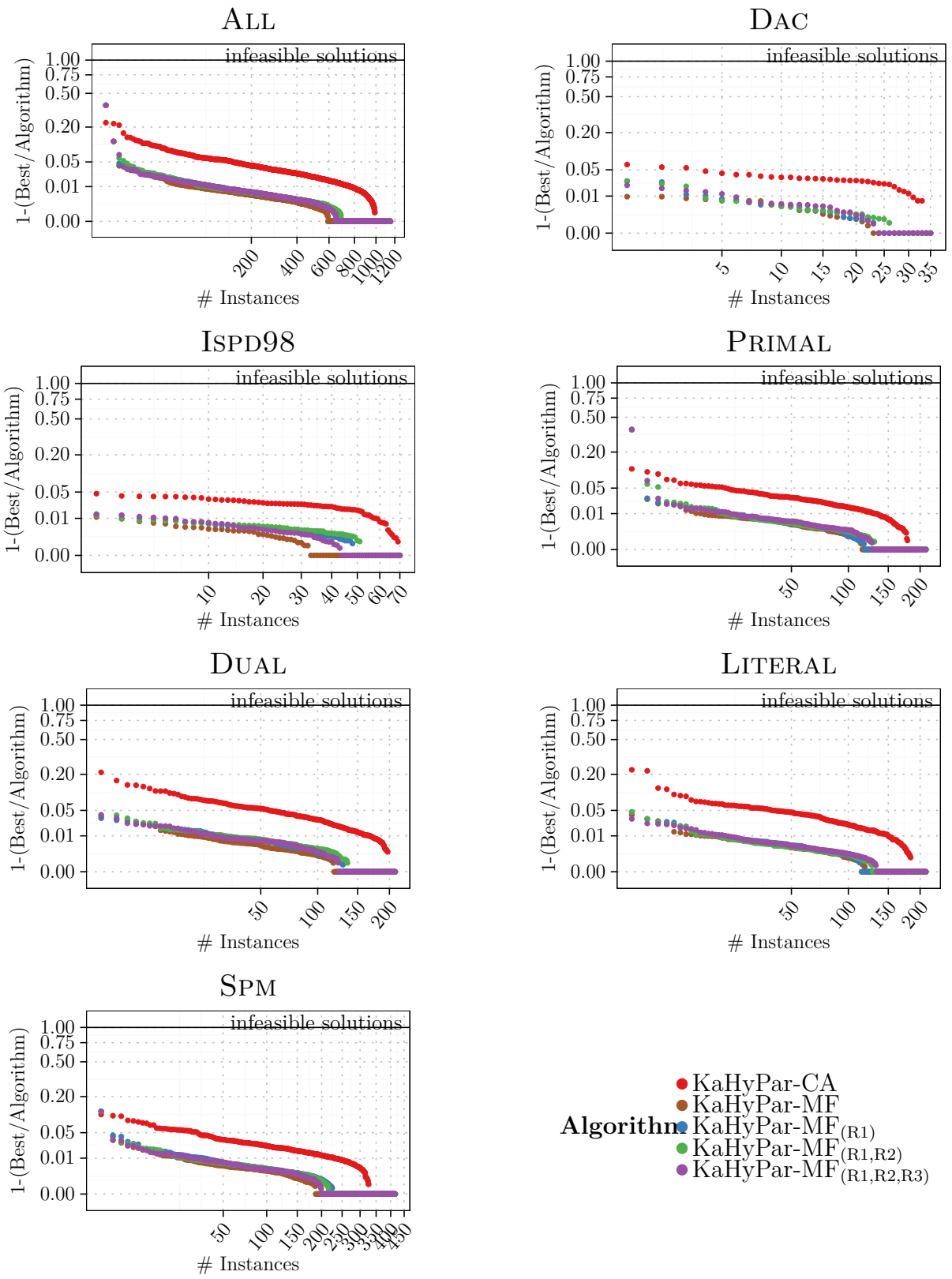

Figure 23: Min-Cut performance plots comparing KaHyPar-MF with KaHyPar-CA. The plots are explained in Section 6.2.

Partitioner	Running Time $t[s]$						
	All	Dac	Ispd98	Primal	Literal	Dual	Spm
KaHyPar-CA	29.26	343.4	21.57	36.44	56.49	58.75	11.31
KaHyPar-MF	81.02	610.48	69.84	107.72	164.68	127.09	33.81
KaHyPar-MF$_{(R1)}$	70.47	526.26	55.56	91.1	136.17	113.87	30.62
KaHyPar-MF$_{(R1,R2)}$	64.82	503.04	47.31	84.65	123.63	101.41	28.95
KaHyPar-MF$_{(R1,R2,R3)}$	55.9	452.27	39.03	71.64	105.2	89.07	25.24

Table 13: Comparing the average running time of KaHyPar-MF with KaHyPar-CA.

F. Detailed Comparison with other Systems

Partitioner	Average $\lambda - 1$						
	All	Dac	Ispd98	Primal	Literal	Dual	Spm
KaHyPar-MF	5162.37	17480.7	5644.61	9458.4	7006.68	2135.09	5096.84
KaHyPar-CA	2.3	3.11	2.2	1.66	2.55	3.5	1.96
hMetis-R	17.58	3.64	1.62	−0.17	1.44	52.25	18.96
hMetis-K	15.24	8.46	1.38	1.61	3	33.27	18.48
PaToH-Q	8.31	13.57	7.92	10.2	10.61	8.82	6.8
PaToH-D	15.09	23.75	15.08	14.53	16.72	19.34	12.89

Table 14: Comparison of average $(\lambda - 1)$ metric of KaHyPar-MF with KaHyPar-CA and other systems on different benchmark types. The results are in percentage relative to KaHyPar-MF.

Partitioner	Average $\lambda - 1$						
	$k = 2$	$k = 4$	$k = 8$	$k = 16$	$k = 32$	$k = 64$	$k = 128$
KaHyPar-MF	851.74	2411.97	4242.11	6496.94	9698.86	13836.24	19502.84
KaHyPar-CA	2.18	2.54	2.64	2.56	2.36	2.04	1.67
hMetis-R	30.3	19.48	18.43	17.69	14.46	12.65	8.67
hMetis-K	29.42	15.51	13.42	13.97	12.84	11.53	8.94
PaToH-Q	10.55	7.29	7.78	8.3	8.45	8.37	7.24
PaToH-D	14.57	15.41	16.1	15.91	15.31	14.87	13.33

Table 15: Comparison of average $(\lambda - 1)$ metric of KaHyPar-MF with KaHyPar-CA and other systems for different values of k. The results are in percentage relative to KaHyPar-MF.

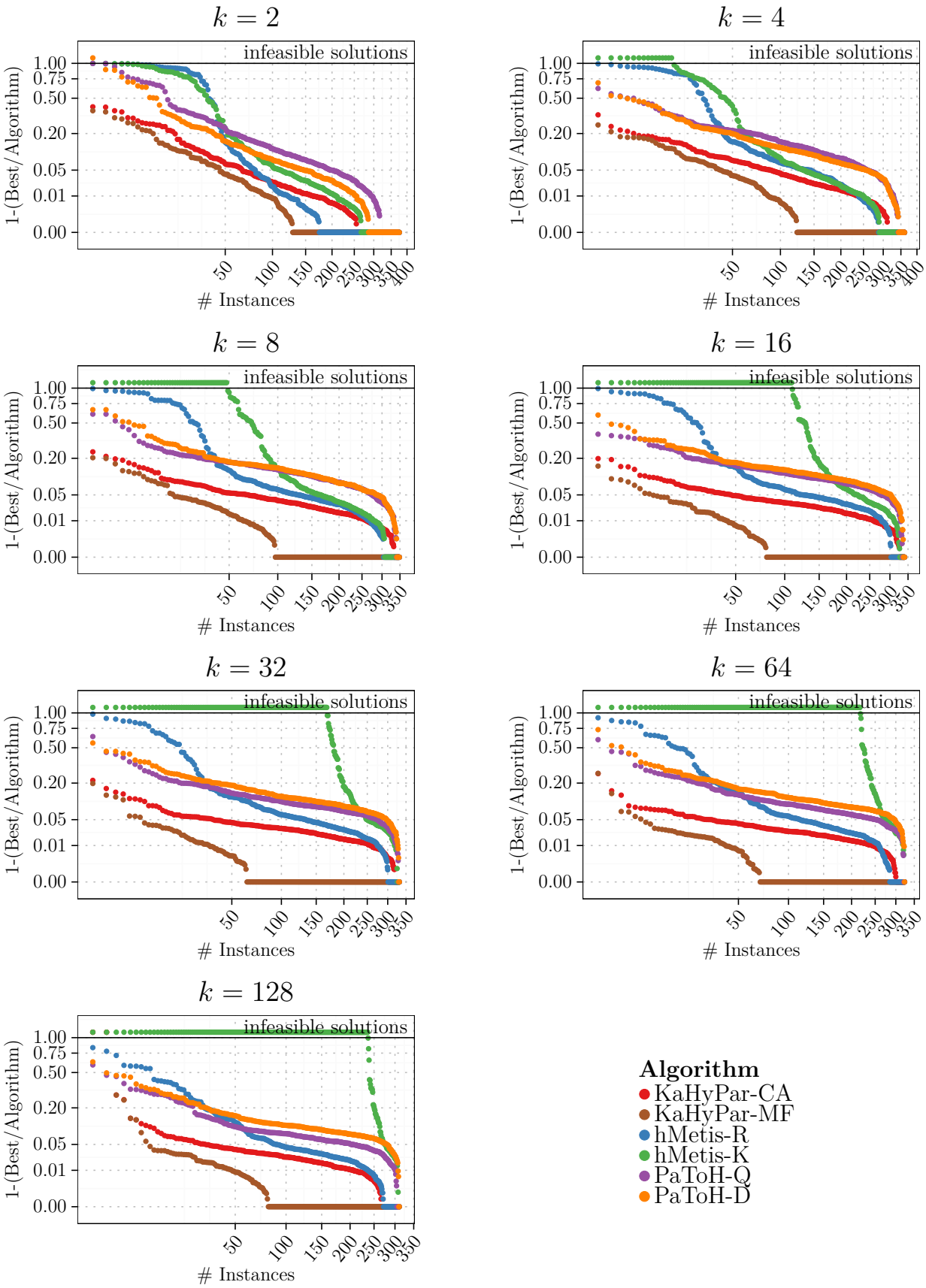

Figure 24: Min-Cut performance plots comparing KaHyPar-MF with KaHyPar-CA and other systems for different values of k.

Partitioner	Running Time $t[s]$						
	$k = 2$	$k = 4$	$k = 8$	$k = 16$	$k = 32$	$k = 64$	$k = 128$
KaHyPar-MF	15.5	23.07	28.7	34.28	41.38	49.23	56.91
KaHyPar-CA	8.23	9.73	12.25	15.93	21.09	28.53	38.49
hMetis-R	19.39	31.7	41.88	50.65	58.05	66.79	74.93
hMetis-K	17.76	19.87	23.83	30.11	42.14	62.52	88.47
PaToH-Q	1.39	2.29	3.17	3.98	4.59	5.37	5.91
PaToH-D	0.31	0.49	0.66	0.82	0.95	1.09	1.2

Table 16: Comparing the average running time of KaHyPar-MF with KaHyPar-CA and other systems for different values of k.