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

Currently, algorithms based on the *FM* idea [15] are the only practical heuristics to improve a k-way partition in a multilevel hypergraph partitioner. However, they are often criticized for their limited ability to lookahead [42]. It might be more beneficial to move a hypernode with small gain, because it will induce many good moves later. We present an alternative *local search* approach based on *Max-Flow-Min-Cut* computations. The framework is inspired by the work of Sanders and Schulz [36] who successfully showed that *flow*-based refinement in combination with the *FM* algorithm significantly improve the quality of partitions in a multilevel graph partitioner. In this work, we develop different modeling techniques of a hypergraph as flow network and show how to improves the connectivity metric of a k-way partition by building a flow problem on a subset of the vertices. We integrated the framework in the hypergraph partitioner *KaHyPar* by applying and adapting the basic framework of [36]. On our large benchmark set with 3222 instances our new configuration outperforms all state-of-the-art hypergraph partitioner on 70% of the instances. In comparison to the latest configuration of *KaHyPar* our new approach produces 2% better quality by a performance slowdown only by a factor of 2.

Zusammenfassung

Algorithmen basierend auf der *FM*-Idee [15] sind zur Zeit die einzigsten praktischen Heuristiken, um eine k-teilige Partitionierung in einem *Multilevel Hypergraph Partitioner* zu verbessern. Jedoch werden sie oft kritisiert für ihre limitierte Eigenschaft vorrauszuschauen [42]. Zum Beispiel könnte es von Vorteil sein ein Knoten mit geringem *Gain* zu verschieben, weil er vielleicht später viele bessere Verschiebungen induziert. Wir präsentieren einen alternativen *Lokale Suche* Ansatz basierend auf *Max-Flow-Min-Cut* Berechnungen. Das Framework ist inspiriert durch die Arbeit von Sanders und Schulz, welche gezeigt haben, dass ein *flow*-basierter Ansatz in Kombination mit dem *FM*-Algorithmus signifikant die Qualität von Partitionierungen in einem *Multilevel Graph Partitioner* verbessert [36]. In dieser Arbeit entwickeln wir verschiedene Modellierungstechnicken eines Hypergraphen als Flussnetzwerk und zeigen wie die *connectivity metric* einer k-teiligen Partitionierung verbessert werden kann, indem ein Flussproblem auf einer Teilmenge der Knoten aufgebaut wird. Wir haben das Framework in den *Hypergraph Partitioner KaHyPar* integriert, indem wir das Framework von [36] übernommen und angepasst haben. Auf unserem großen *Benchmark Set* mit 3222 Instanzen erzielt unsere neue Konfiguration auf 70% der Instanzen eine bessere Qualität als die meisten *State-of-the-Art* Partitionierer. Im Vergleich zu der letzten Konfiguration von *KaHyPar* erreichen wir mit unserem Ansatz 2% bessere Qualität mit nur doppelt so langer Laufzeit.

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 serious 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 satisfies a lower and upper bound, while we simultaneously want to minimize an objective function.
Classical application areas can be found in *VLSI* design, parallelization of the *Sparse Matrix-Vector Product* and simplifying *SAT* formulas [25, 30, 32]. 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 [8]. A wire can connect more than two gates, therefore a hypergraph models a circuit more accurate than a graph. In *SAT* solving hypergraph partitioning is used to decompose a formula into smaller subformulas, which can be solved easier [30]. Beneath the traditional application areas hypergraph partitioning can also be found in more vivid areas like *Warehouse Planning*. A warehouse consists of several storage spaces where products can be placed. If a list of previous orders is available, we can interpret the products as vertices and the orders as hyperedges. If we partition the hypergraph into k blocks, where k is the number of storage spaces, we can place products in the warehouse such that products are close to each other if they are often ordered together.
Hypergraph partitioning is an NP-hard problem [29] and it is even NP-hard to find a good approximation [7]. The most common heuristic used in state-of-the-art hypergraph partitioner is the *multilevel paradigm* [9, 21, 25]. First, a sequence of smaller hypergraphs is calculated by contracting a set of hypernode pairs in each step (*coarsening phase*). If the hypergraph is small enough, we can use expensive heuristics to *initial partition* the hypergraph 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 a partition (*refinement phase*).
There exists several *local search* heuristics for improving a partition of a hypergraph, but only the *FM* algorithm leads to a practical performance of a multilevel hypergraph partitioner for large benchmarks [32]. In general, the *FM* heuristics maintains gain values (according to the objective function) of moving a node from its current block to another block [15]. A move is performed, if its gain value is maximum among all possible moves. The algorithm can be implemented in linear time. The main disadvantage of the algorithm is its limited ability to lookahead [42]. E.g., it might be more beneficial to move a hypernode with a small gain because it will induce many good moves later. Therefore, the algorithm tends to find locally optimal solutions.
Sanders and Schulz [36] successfully integrated a *flow*-based refinement algorithm in their multilevel graph partitioner. It is well known that a maximum (s,t)-flow calculation yields to a minimum (s,t)-cutset on graphs [16]. Their general approach was to extract a subgraph around the cut and configure the source and sink sets of the flow problem such that a maximum flow calculation on the subgraph leads to a smaller cut on the original graph. In combination with the *FM* heuristic, their *local search* algorithm can find out of locally optimal solutions and produces the best partitions for a wide range of graph partitioning benchmarks.

1.1. Problem Statement

Currently, there are no competitive alternatives to the *FM* heuristic as *local search* algorithm for a multilevel hypergraph partitioner. Sanders and Schulz [36] showed that *flow*-based approaches could be used in a multilevel graph partitioner to obtain high-quality partitions. Their algorithm is a generic framework, which basic ideas can be applied one-to-one on hypergraphs. However, several key challenges remain.

First, we have to find an appropriate model of a hypergraph as flow network. Each maximum (s,t)-flow on this model should induce a minimum (s,t)-cutset on the hypergraph. Afterwards, the model should be used to improve the cut of a given bipartition by executing a flow problem on a subset of the hypernodes. Therefore, the sources and sinks must be configured to satisfy the above-formulated constraint.
The framework should be integrated into the n-level hypergraph partitioner *KaHyPar*. *KaHyPar* is a multilevel hypergraph partitioner in its most extreme version by only contracting two vertices in one level of the multilevel hierarchy [1, 21, 37]. In the *refinement phase*, *n-local searches* are instantiated. Therefore, the most challenging part is to implement the framework in such a way that we obtain high-quality partitions and simultaneously ensure that the performance reduction is within a constant factor.

1.2. Contributions

We present several sparsifying techniques of the state-of-the-art hypergraph flow network modeling approach proposed by Lawler [28]. Our experiments indicate that maximum flow algorithms are up to a factor of 3 faster with our new network. Further, we show that the source and sink sets of the resulting flow network of a subhypergraph of an already partitioned hypergraph can be configured more flexible than on graphs. More precisely, applying the approach of Sanders and Schulz [36] directly on hypergraphs results in a minimum (S,T)-cutset greater or equal as with our new technique. We integrate the framework of [36] into *KaHyPar* and show that *flow*-based refinement in combination with the *FM* algorithm produces on a majority of a wide range of real-world benchmarks the best-known partitions in comparison to other state-of-the-art hypergraph partitioners. In numbers, compared to 5 different systems we achieve on 70% of 3222 benchmark instances the best-known partitions. In comparison to the latest quality preset of *KaHyPar* our new approach produces on average 2% better partitions and is only slower by a factor of 2.

1.3. Outline

We first introduce necessary notations and summarize related work in Section 2 and 3. Afterwards, we describe sparsifying techniques of the flow network proposed by Lawler [28] in Section 4. In Section 5 we present our optimized source and sink set modeling approach and describe the integration of our *flow*-based refinement framework into the n-level hypergraph partitioner *KaHyPar*. The evaluation of our new flow network proposed in Section 4 and framework proposed in Section 5 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. $I(v)$ denotes the set of all *adjacent* nodes of v. The *degree* of a node v is $d(v) = |I(v)|$.

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* $P = (v_1, \dots, v_k)$ 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* $P = (v_1, \dots, v_k)$ 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 [39]. 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 [24]. We can transform a general directed graph G into a *DAG* if we contract each *strongly connected component*. All concepts are illustrated in Figure 1.

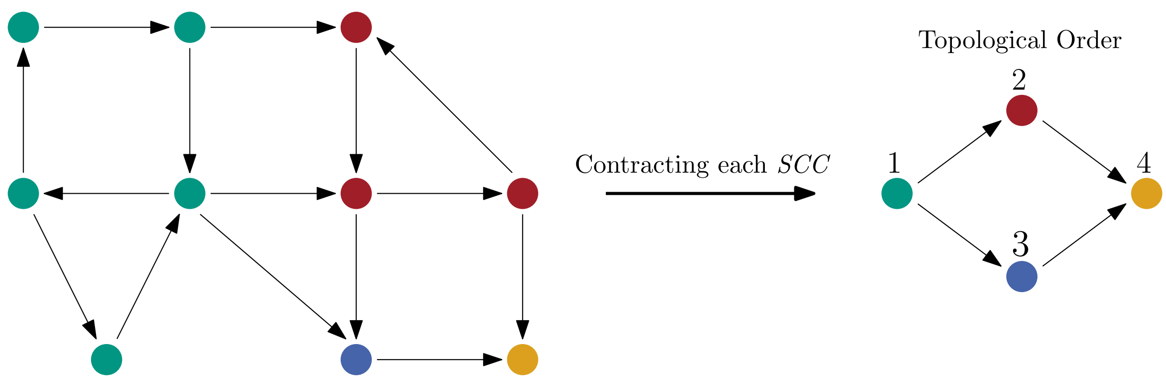

Figure 1: Example of *strongly connected components* of a directed graph and a *topological order* on a *directed acyclic graph*. Each *SCC* is marked in the same color.

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 $G = (V, E, c)$ with capacity function $c : E \to \mathbb{R}_+$ and a source $s \in V$ and a sink $t \in V$. The maximum flow problem is about finding 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) $\forall (u, v) \in E : f(u, v) \leq c(u, v)$ (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)

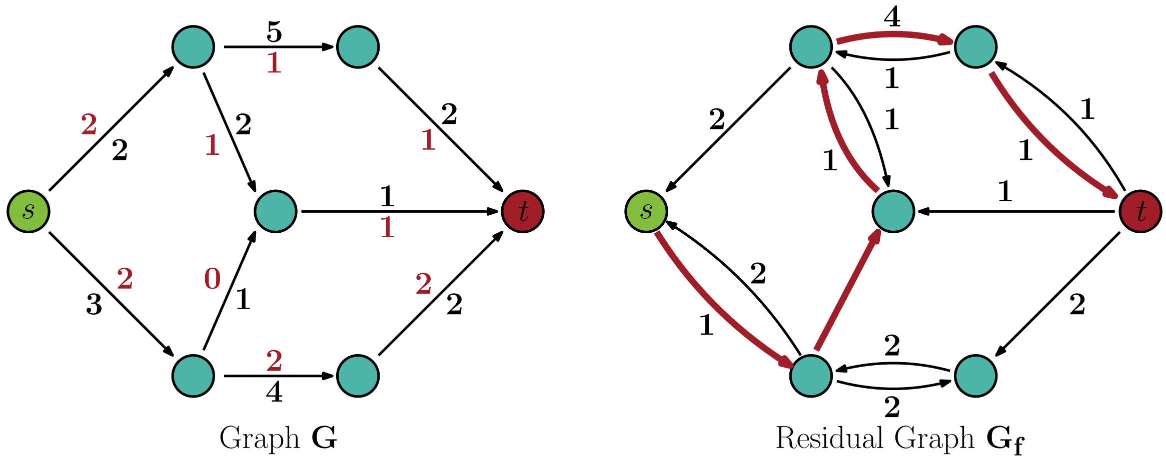

Figure 2: Figure illustrates concepts related to the maximum flow problem. A valid flow f (red values) from s to t on a graph G is shown on the left side. The corresponding *residual graph* G_f with its *residual capacities* (black values) is illustrated on the right side. The red highlighted path represents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

The capacity constraint restricts the flow on edge (u, v) by its capacity $c(u, v)$. 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) $\forall (u, v) \in E : r_f(u, v) = c(u, v) - f(u, v)$

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

For a 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 formal $E_f = \{(u, v) \in V \times V \mid r_f(u, v) > 0\}$. Figure 2 illustrates all presented concepts.

The *Max-Flow-Min-Cut*-Theorem is fundamental for many applications related to the maximum flow problem [16].

Theorem 2.1. *The value of a maximum (s, t)-flow obtainable in a graph G is equal with the weight of the minimum 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 [17]. A can be calculated with a *BFS* in G_f starting from s.
From this analogy, many solutions for related problems arose. Samples are listed below:

(i) Maximum Bipartite-Matching

(ii) Minimum-Weight Vertex Separator

(iii) Number of Edge-Disjoint Paths

(iv) Number of Vertex-Disjoint Paths

Solutions for those problems sometimes involves 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

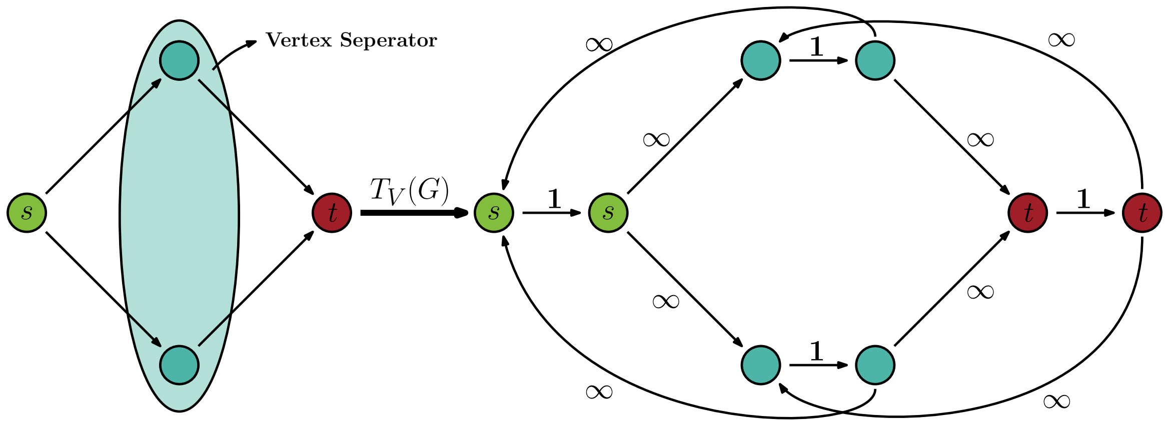

Figure 3: Illustration of the vertex separator problem and the transformation $T_V(G)$ in which we can find a minimum vertex separator with maximum flow computation.

for this work is to find a minimum-weight (s,t)-vertex separator in a graph $G = (V, E, c)$ with $c : V \to \mathbb{R}_+$.

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 [41]:

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

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

The vertex separator problem and transformation $T_V(G)$ is illustrated in Figure 3. Obviously, no edge between two adjacent nodes 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 with the following algorithm:

(i) Calculate a maximum flow of $T_V(G)$

(ii) Find the corresponding minimum (s,t)-cutset.

(iii) Map each cut edge (v', v'') to the corresponding node v in G

Given a set of sources S and sinks T. The *multi-source multi-sink* maximum flow problem is about finding 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 $c(s, s') = c(t', t) = \infty$.

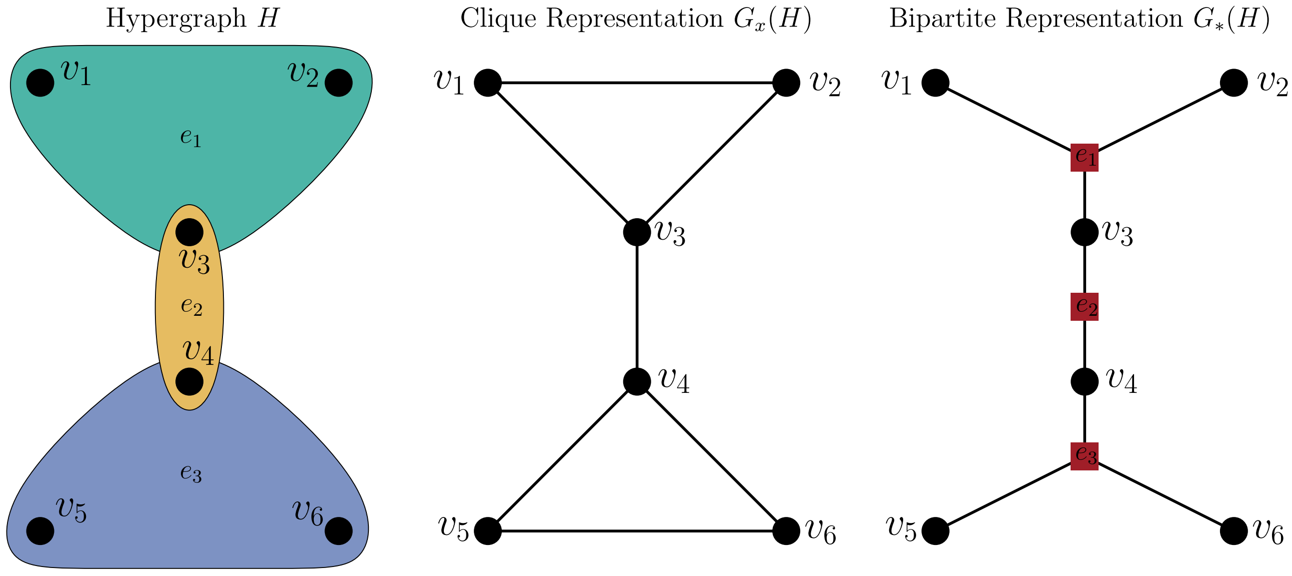

Figure 4: Example of a hypergraph H and its two corresponding graph representations.

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:* $\forall e \in E : e \subseteq V$*).*

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 the hypernodes of a net are called *pins*. 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 $u, v \in e$. $I(v)$ 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 the 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* $E_{V'} = \{e \cap V' \mid e \in E : e \cap V' \neq \emptyset\}$.

A hypergraph $H = (V, E, c, \omega)$ can be represented as an undirected graph. There are two standard transformations, called *clique* and *bipartite* representation [23]. 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 Figure 4.

2.4. Hypergraph Partitioning

Definition 2.8. *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 $\Pi = \{V_1, \dots, V_k\}$, 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 $V_i, V_j \in \Lambda(e, \Pi)$.

Definition 2.9. *For a k-way partition $\Pi = \{V_1, \dots, V_k\}$ of a hypergraph H the quotient graph $Q = (\Pi, E')$ is an undirected graph containing an edge between each pair of adjacent blocks of Π. More formal, $E' = \{(V_i, V_j) \mid \exists e \in E : V_i, V_j \in \Lambda(e, \Pi)\}$*

We say 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, which should either be minimized or maximized. 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 sum 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.

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 algorithms 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 $v_k = t$ [14]. Figure 2 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 [14].
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 [16] 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.
Edmond and Karp [14] improved Ford & Fulkerson 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 by a simple *BFS* calculation. 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 Figure 5.

3.1.2. Push-Relabel Algorithm

Goldberg and Tarjan [19] implemented the first 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 for s, $d(s) = 1$ 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* [19, 38], but this implementation is not practical due to a large hidden constant factor.

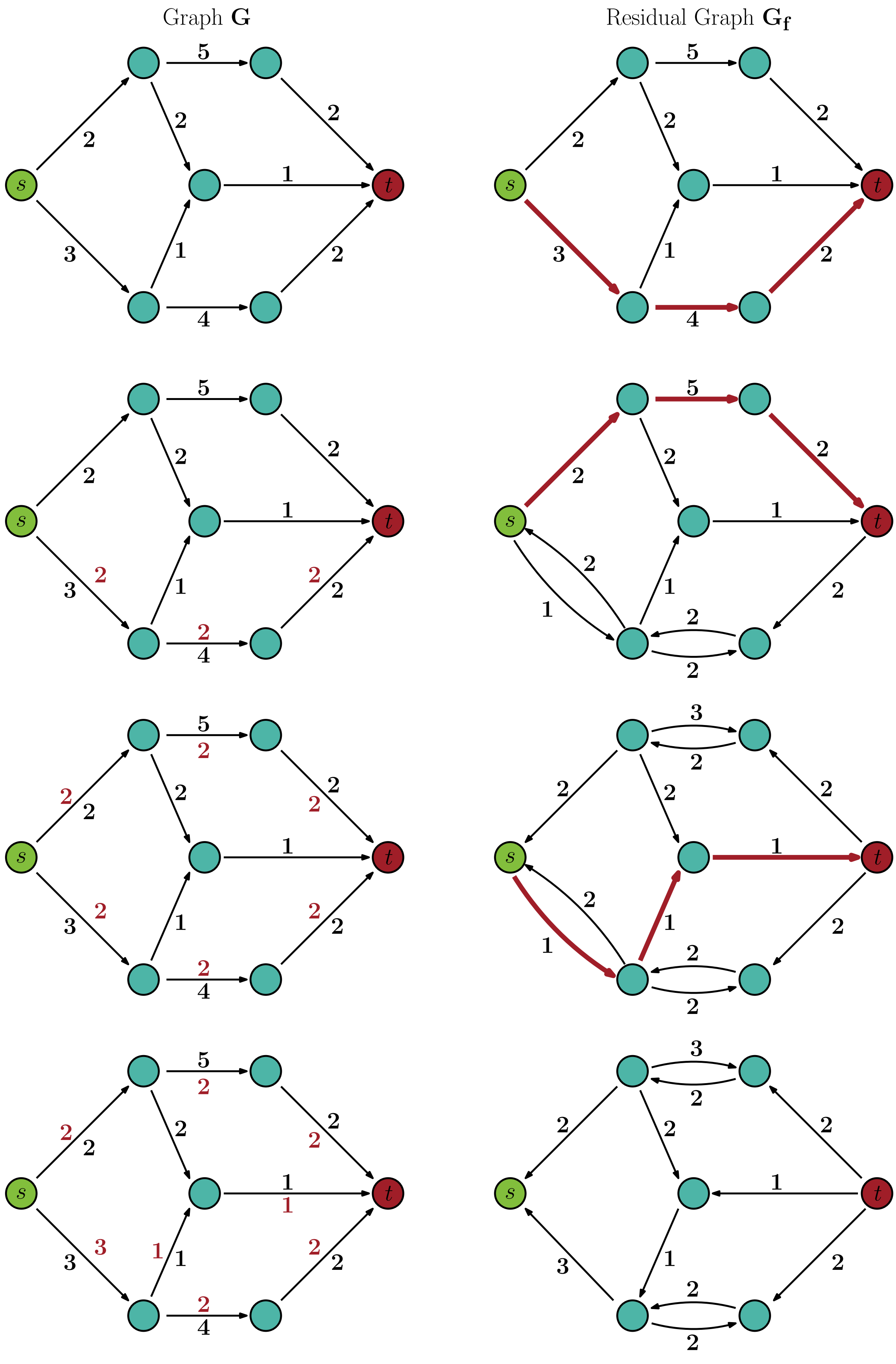

Figure 5: Sample execution of Edmond & Karps maximum flow algorithm [14]. The network G with its capacities c (black values) and flow f (red values) is illustrated on the left side. The residual graph G_f with its *residual capacities* r_f (black values) is presented on the right side. In each step the current *augmenting path* in G_f is highlighted by a red path.

The *push-relabel* algorithm is one of the fastest maximum flow algorithms in practice because there exist several speed-up heuristics. 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 [11]. This can be done with a backward *BFS* in linear time. This heuristic is performed periodically, e.g., after every n relabeling.
The second heuristic is the *gap heuristic* [10, 13]. 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. Modelling Flows on Hypergraphs

Consider the *bipartite graph* representation $G_*(H)$ of a hypergraph H (see Section 2.3). Hu and Moerder [23] introduced node capacities in $G_*(H)$. Each hyperedge node e has a capacity equal to $\omega(e)$ and each hypernode node has infinite capacity. Further, they showed 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 [28]:

Definition 3.1. *Let T_L be the transformation of a hypergraph $H = (V, E, c, \omega)$ into a flow network $T_L(H) = (V_L, E_L, c_L)$ proposed by Lawler [28]. $T_L(H)$ 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* $c_H(e', e'') = \omega(e)$

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

An example of this transformation is shown in Figure 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 $v \in V$. Because a hypernode cannot be in a minimum-capacity (s,t)-vertex separator, because each $v \in V$ has infinity capacity [23]. 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 [34]. Using *Edmond-Karps* maximum flow algorithm (see Section 3.1.1) on flow network $T_L(H)$ takes time $\mathcal{O}(|V|^2|E|^2)$ [28].
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 corresponding bipartition are all hypernodes $v \in V$ *reachable* from s in the *residual graph* of $T_L(H)$ and the counterpart are all hypernodes not *reachable* from s.
In 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 $e \in E$. E.g., $e'_1 \cap e'_2$ or $e''_1 \cap e'_2$ should be the same as $e_1 \cap e_2$. However, it should be clear out of the context which terminology is used.

3.3. Max-Flow-Min-Cut Based Local Search on Graphs

It seems naturals to utilize maximum flow computations to improve the cut metric of a given partition of a graph. Lang and Rao [27] use an approach, called *Max-Flow Quotient-cut Improvement* (MQI), to improve the cut of a graph when metrics such as *expansion* or *conductance*

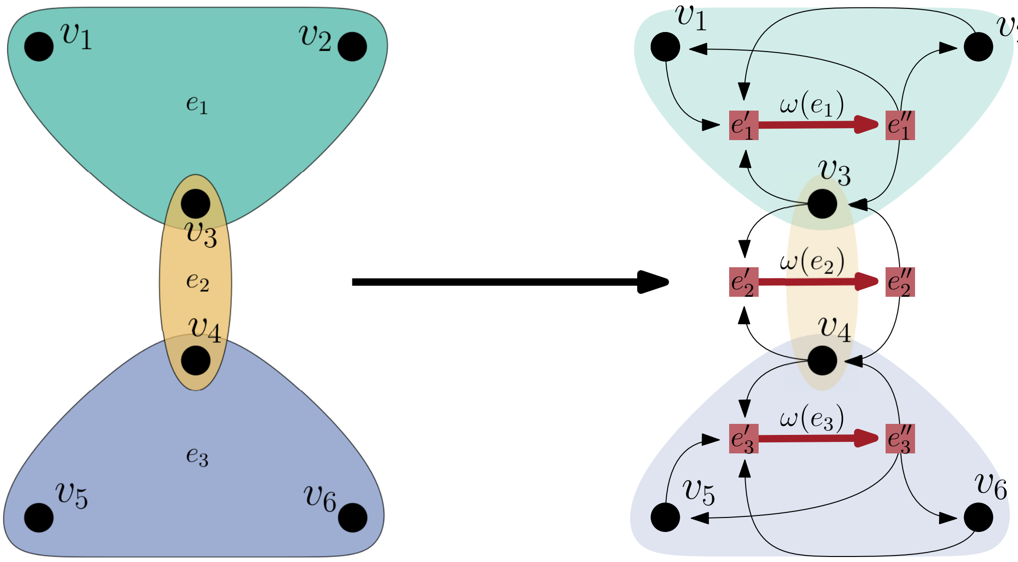

Figure 6: Transformation of a hypergraph into a equivalent flow network by Lawler [28]. Note, capacity of the black edges in the flow network is ∞.

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 constructing a flow problem. Andersen and Lang [4] proposed a flow-based improvement algorithm, called *Improve*, which works similar as MQI, but did 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 [36] integrate flow-based refinement algorithm in their *multilevel graph partitioning* framework *KaFFPa*. In general, they build a flow problem around a region B of 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 to a feasible cut in the original graph. 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 the cut of a graph.

3.3.1. Balanced Flow-Based 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 [36].
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* only touch 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 guarantees 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 to 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 Figure 7.

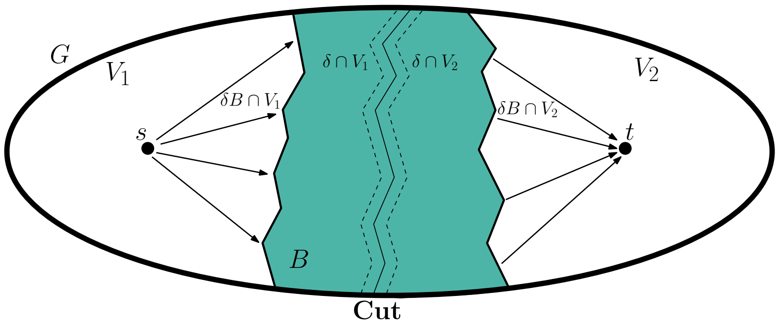

Figure 7: Illustration of setting up a flow problem around the cut of graph G [4].

3.3.2. Adaptive Flow Iterations

Sanders and Schulz [36] suggested several heuristics to improve their basic approach. If the *Max-Flow-Min-Cut* computation on G_B leads to an improvement in 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. For this propose 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 found 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* [36].

3.3.3. Most Balanced Minimum Cut

Picard and Queyranne [33] showed 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 Figure 8. 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 [39] and sweep to the reverse topological order of the contracted graph to enumerate all *closed node sets*. Note, if we contract all SCC of G the resulting graph is a *Directed Acyclic Graph* (DAC). Therefore, a topological order exists. With the Theorem of Picard and Queyranne [33] 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 induced 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 searching with a large α often leads to cuts in G which violates the balanced constraints. We are able with this technique to convert an infeasible solution into a feasible by finding the *Most Balanced Minimum Cut* (MBMC) with one maximum flow computation.

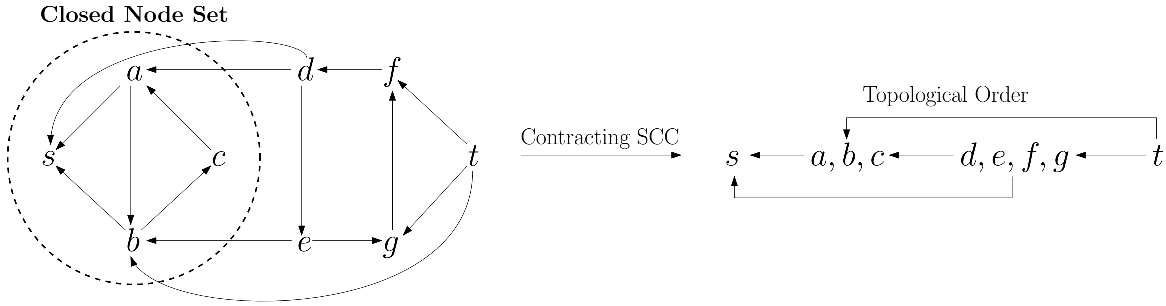

Figure 8: Nodes $C = \{s, a, b, c\}$ illustrates a *closed node set* in a graph G (left side). After contracting all *Strongly Connected Components*, we can enumerate all *closed node sets* of G by sweeping in reverse topological order to the contracted graph (right side).

3.3.4. Active Block Scheduling

Active Block Scheduling is a *quotient graph style refinement* technique for k-way partitions [22, 36]. 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*. Initial all blocks are *active*. A block becomes *inactive* if its boundary did not change in a round. The algorithm terminates, if all blocks are *inactive*.
Fiduccia and Mattheyses [15] introduces 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 its counterpart 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 [36].

3.4. Hypergraph Partitioning

In this Section, we review how most hypergraph partitioner solves the *hypergraph partitioning problem* (see Section 2.4). The most successful approach is the *multilevel paradigm* [3, 5, 32] 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 stages algorithm to solve the *hypergraph partitioning problem* (see Figure 9). In the first stage, called *coarsening phase*, pairs of hypernodes are chosen to be contracted. 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 *initial partition* H into k blocks. Afterwards, we can *uncontract* each *level* in reverse order of *contraction* and project the partition to the next *level*. After unpacking a *refinement* heuristic can be used to improve the quality of the current

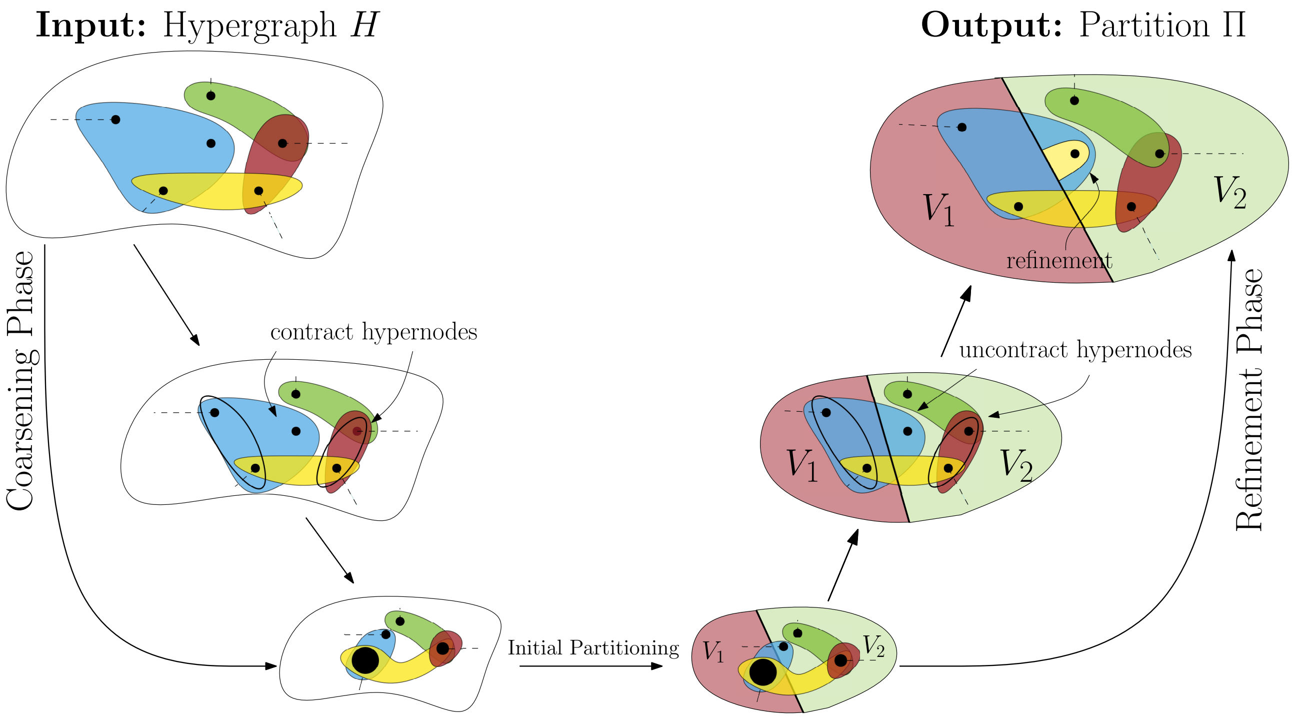

Figure 9: Multilevel Hypergraph Partitioning

partition according to an objective function. The most common used *refinement* algorithm is the *FM* algorithm [15] (see Section 3.3.4).

3.4.2. KaHyPar - n-Level Hypergraph Partitioning

KaHyPar is a multilevel hypergraph partitioner in its most extreme version by removing 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 [21]. 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 [37]. *KaHyPar* consists of four phases: *Preprocessing* and the three stages of the *multilevel paradigm*.
The *preprocessing* step detects community structures of a hypergraph. The hypergraph is transformed into a bipartite graph $G_*(H)$ (see Section 2.3) and a community detection algorithm is executed which optimize *modularity* [18, 21]. 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 $r(u, v) := \sum_{e \in I(u) \cap I(v)} \frac{\omega(e)}{|e|-1}$ [25]. The function prefers vertices which share a large number of heavy nets with small size. The rating function is evaluated *lazily* which means that after a contraction of two hypernodes u and v all ratings related to those vertices are *invalid* [37]. If the PQ returns an *invalid* rating, we immediately recalculate $r(u, v)$ and insert the new rating in the PQ. The *initial partitioning* uses the *recursive bisection* approach to calculate a k-way partition in combination with a portfolio of initial partitioning techniques [20]. In the *refinement phase*, a localized *FM* search is started [15], initialized with the current uncontracted vertices. The *local search* maintains k *priority queues* for each block V_i [1]. A hypernode v contained in PQ_i 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 [32]. The recalculation of all gain values at the beginning of a *FM* pass is one of the main bottlenecks of the algorithm [32]. Therefore, Schlag [1, 37] introduces a *Gain Cache* over the multilevel hierarchy, 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 [31] showed 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 and μ is the average gain and σ^2 the corresponding variance.

4. Optimized Approach on Modelling Flows in Hypergraphs

In Section 3.2 we have shown how a hypergraph H could be transformed into a flow network $T_L(H)$ such that every minimum-weight (S,T)-cutset in H is a minimum-capacity (S,T)-cutset in $T_L(H)$ [28]. However, the resulting flow network has significantly more nodes and edges than the original hypergraph. Finding a maximum (S,T)-flow is usually a very computation intensive task. Therefore, different modelling 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 $T_L(H)$. But 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 could be removed by inserting an undirected flow edge between the corresponding nodes $v_1, v_2 \in e$ (see Section 4.3). Finally, we combine the two suggested approaches in a *Hybrid*-Network (see Section 4.4).

4.1. Removing Hypernodes via Clique-Expansion

In this Section we show how all hypernodes of $T_L(H)$ could be removed. If a hypernode $v \in V$ occurs in an augmenting path P the previous node in the path must be a hyperedge node either e' or e''. Further, for all $e \in I(v)$ the capacity $c_L(v, e')$ is ∞. This leads to the conclusion, if we push flow over a hypernode v, comming from a hyperedge node, we can redirect the flow to any hyperedge node $e' \in I(v)$ during the whole maximum flow calculation, because $c_L(v, e') = \infty$. A hypernode v acts as a *bridge* between all incident hyperedges in the *Lawler*-Network. Therefore, the idea is to remove all hypernodes from $T_L(H)$ and instead inserting for all $v \in V$ a clique between all $e_1, e_2 \in I(v)$ with $e_1 \neq e_2$. In the following we will define our new network more general and show how to remove any subset $V' \subseteq V$.

Definition 4.1. *Let T_H be a transformation that converts a hypergraph $H = (V, E, c, \omega)$ into a flow network $T_H(H, V') = (V_H, E_H, c_H)$ 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) *$\forall v \in V'$ we add a directed edge (e_1'', e_2'), $\forall e_1, e_2 \in I(v)$ with $e_1 \neq e_2$ with capacity $c_H(e_1'', e_2') = \infty$ (clique expansion).*

(iii) *$\forall e \in E$ we add a directed edge (e', e'') with capacity $c_H(e', e'') = \omega(e)$ (same as in $T_L(H)$).*

(iv) *$\forall v \in V \setminus V'$ we add for each incident hyperedge $e \in I(v)$ two directed edges (v, e') and (e'', v) with capacity $c_H(v, e') = c_H(e'', v) := \infty$ (same as in $T_L(H)$).*

An example of the transformation is shown in Figure 10. To show the correctness of $T_H(H, V')$, we need to proof that a minimum-capacity (S,T)-cutset in $T_H(H, V')$ is equal with a minimum-weight (S,T)-cutset in H. However, in the correctness proof we need a preparing lemma.

Lemma 4.1. *Let $G = (V, E, c)$ be a graph with a capacity function $c : E \to \mathbb{R}_+$. Further, let S and T be a source and sink set with $S \cap T = \emptyset$ and $\forall s \in S : \forall (s, v) \in E : c(s, v) = \infty$ and $\forall t \in T : \forall (v, t) \in E : c(v, t) = \infty$.*
For any $V' \subseteq V$ a minimum-capacity (S,T)-cutset in G is equal with a minimum-capacity (S', T')-cutset in G, where $S' = S \setminus V' \ \cup \bigcup_{s' \in I(V' \cap S)} \{s'\}$ and $T' = T \setminus V' \ \cup \bigcup_{t' \in I(V' \cap T)} \{t'\}$ and $S' \cap T' = \emptyset$.

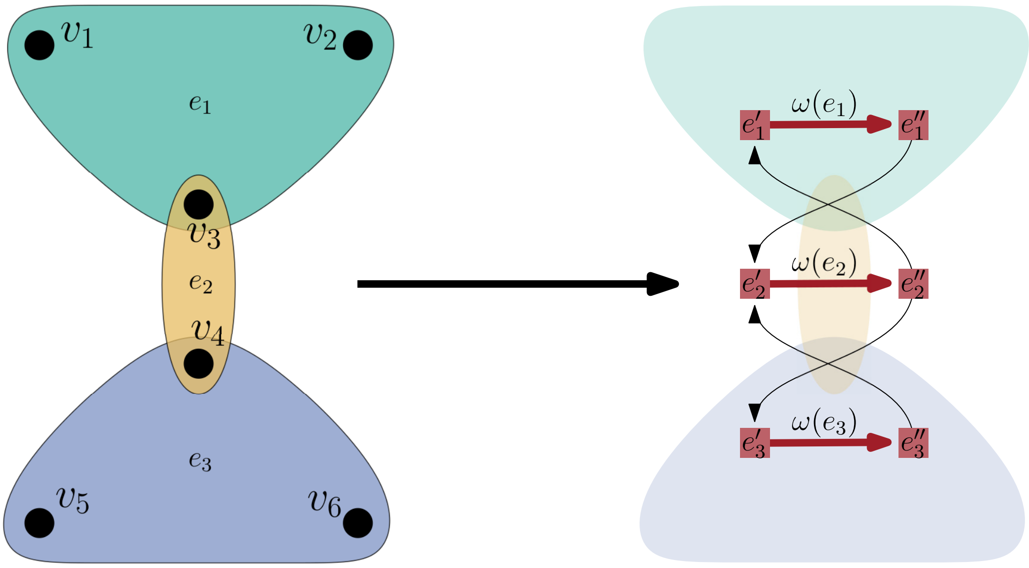

Figure 10: Transformation of a hypergraph into a equivalent flow network by removing all hypernodes. Note, capacity of the black edges in the flow network is ∞.

Proof. Let G' be the graph obtained by removing all $v \in V' \cap (S \cup T)$. If the minimum-capacity (S,T)-cutset in G is smaller than ∞, then no outgoing edge of a node $s \in S$ and no incomming edge of a node $t \in T$ can be cut, because for all those edges e the capacity $c(e) = \infty$. Therefore, if $S' \cap T' = \emptyset$ each minimum-capacity (S,T)-cutset in G must be equal with a minimum-capacity (S',T')-cutset in G'.
Each (S,T)-cutset in G is also a (S',T')-cutset in G' and vice versa. If the minimum-capacity (S,T)-cutset in G is ∞, every cutset separating (S,T) resp. (S',T') is a minimum (S,T)- resp. (S',T')-cutset. $\square$

As a consequence of this lemma, we could replace (or even remove) e.g. a source hypernode $v \in S$ of $T_L(H)$ and instead add all incomming hyperedge nodes $e' \in I(v)$ as source nodes to the flow problem. Because for all incomming 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 $T_H(H,V') = (V_H, E_H, c_H)$ ($V' \subseteq V$), where $S' = S \setminus V' \cup \bigcup_{e \in I(V' \cap S)} \{e'\}$ and $T' = T \setminus V' \cup \bigcup_{e \in I(V' \cap T)} \{e''\}$.*

Proof. Consider the bipartite graph representation $G_* = (V_*, E_*, c_*)$ of a hypergraph $H = (V, E, c, \omega)$ presented in Section 2.3 and 3.2, where for all $v \in V : c_*(v) = \infty$ and for all $e \in E : c_*(e) = \omega(e)$. A minimum-weight (S,T)-vertex separator in G_* is equal with a minimum-weight (S,T)-cutset in H. A minimum-weight (S,T)-vertex separator can be calculated by finding a minimum-capacity (S,T)-cutset in $T_L(H)$. Let G_H be the graph obtained by removing all $v \in V' \setminus (S \cup T)$ of G_* and adding a clique between all $e \in I(v)$. A minimum-weight (S,T)-vertex separator in G_H can be calculated by finding a minimum-capacity (S,T)-cutset in our new network $T_H(H, V' \setminus (S \cup T))$. We will show that each vertex separator in G_* is also a vertex separator in G_H and vice versa. With Lemma 4.1 we can show that each minimum (S,T)-cutset of $T_H(H, V' \setminus (S \cup T))$ is equal with a minimum (S',T')-cutset of $T_H(H,V')$ and conclude the proof. We will denote a vertex separator of a graph G with $V_S(G)$ and define $V'' := V' \setminus (S \cup T)$. We will show, that $V_S(G_*) = V_S(G_H)$ with the restriction $V_S(G_*) \subseteq E$ and $V_S(G_H) \subseteq E$.

Assume that $V_S(G_*) \subseteq E$ is not a vertex separator in G_H. After removing all $e \in V_S(G_*)$ of G_H, there exists still a path $P_H = \{v_1, \ldots, v_k\}$ with $v_1 \in S$ and $v_k \in T$ of G_H. We can extend P_H to a path P_* in G_*. We define $P_* := P_H$ and replaces every occurence of a sequence $v_i = e_1 \in E$ and $v_{i+1} = e_2 \in E$ with a triple (e_1, v, e_2) in P_*, where $v \in e_1 \cap e_2 \cap V''$ (not empty per construction). P_* does not contain a vertex of $V_S(G_*)$, because we removed all hyperedge nodes $e \in V_S(G_*)$ from G_H before construction of P_* and a hypernode is not part of the vertex separator $V_S(G_*) \subseteq E$ per defintion. P_* connects S and T in G_*, which is a contradiction that $V_S(G_*)$ is a vertex separator in G_*.

Assume that $V_S(G_H) \subseteq E$ is not a vertex separator in G_*. After removing all $e \in V_S(G_H)$ of G_*, there exists still a path $P_* = \{v_1, \ldots, v_k\}$ with $v_1 \in S$ and $v_k \in T$ of G_*. We can extend P_* to a path P_H in G_H. We define $P_H := P_*$ and remove all $v \in P_* \cap V''$ from P_H. G_* is a bipartite graph per definition. Therefore, each path P_* in G_* is an alternating path of hypernodes and hyperedges. The predecessor and successor of a hypernode $v \in P_* \cap V''$ must be hyperedges e_1 and e_2. If $v \in V''$, then v is not contained G_H. Instead, there is a clique between all $e \in I(v) \Rightarrow (e_1, e_2)$ is contained in G_H. P_H not contain any vertex of $V_S(G_H)$, because we removed all hyperedge nodes $e \in V_S(G_H)$ from G_*. P_H connects S and T in G_H, which is a contradiction that $V_S(G_H)$ is a vertex separator in G_H.

A minimum-weight (S,T)-vertex separator in G_* and G_H contains only hyperedges, because the weight of all hypernodes in G_* and G_H is ∞. Therefore, each minimum-weight (S,T)-vertex separator in G_* is also a minimum-weight (S,T)-vertex separator in G_H, because $c(V_S(G_*)) = c(V_S(G_H))$. With Lemma 4.1 follows that a minimum-weight (S,T)-vertex separator in G_* resp. G_H can also be calculated by finding a minimum-capacity (S',T')-cutset in $T_L(H)$ resp. $T_H(H,V')$. Therefore, there exists a equivalence between a minimum-weight (S,T)-cutset E_{min} of H and the following statements:

E_{min} is a minimum…

(i) …-weight (S,T)-cutset in H

(ii) …-weight (S,T)-vertex separator in G_*

(iii) …-capacity (S,T)-cutset in $T_L(H)$

(iv) …-capacity (S',T')-cutset in $T_L(H)$ (follows from (iii) with Lemma 4.1)

(v) …-weight (S,T)-vertex separator in G_H

(vi) …-capacity (S,T)-cutset in $T_H(H,V'')$

(vii) …-capacity (S',T')-cutset in $T_H(H,V')$ (follows from (vi) with Lemma 4.1)

□

As a consequence of this Theorem a minimum-weight (S,T)-cutset of H can also be calculated with $T_H(H,V')$. A open problem is how to obtain 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 with the following lemma.

Lemma 4.2. *Let f be a maximum (S,T)-flow of $T_L(H)$ and A be the set of all nodes reachable from a node $s \in S$ in the residual graph.*

$$\text{If } v \in A \Leftrightarrow \exists e \in I(v) : e'' \in A$$

Proof. If $e'' \in A$, then $v \in A$, because $c_L(e'', v) = \infty$ and $r_f(e'', v) = \infty$. Assume, if $v \in A$, then $\forall e \in I(v) : e'' \notin A \Rightarrow f(e'', v) = 0$ (Note, $c(e'', v) = \infty$). Otherwise $r_f(v, e'')$ would be greater than zero and this would imply that $e'' \in A$, because $v \in A$. Each path P in the *residual graph* of $T_L(H)$ from $s \in S$ to v must be of the form $P = (\ldots, e', v)$. For at least one $e \in I(v)$ there must be a positive flow $f(v, e') > 0$, otherwise edge (e', v) would be not contained in the *residual graph* of $T_L(H)$ (Note, $c_L(e', v) = 0$). There is a positive flow leaving node v, but there is no flow entering node v, because $\forall e \in I(v) : f(e'', v) = 0$. This violates the conservation of flow constraint for node v and therefore f is not a valid flow function. There must exist at least one $e \in I(v)$ with $f(e'', v) > 0 \Rightarrow r_f(v, e'') > 0 \Rightarrow e'' \in A$. □

Lemma 4.2 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. Removing Low-Degree Hypernodes

The resulting flow network $T_H(H, V)$ proposed in Section 4.1 has significantly less nodes than the network $T_L(H)$ suggested by Lawler. On the other hand, the number of edges could be much larger.

Let's consider a hypernode $v \in V$. We replace v in $T_L(H)$ with a clique between all hyperedges of $I(v)$. The number of edges added to $T_H(H, V)$ depends on the degree of v. Every hypernode $v \in V$ 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 with the drawback of an 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 reduce 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 Hyperedges via Undirected Flow-Edges

If we want to find a minimum-weight (S, T)-cutset in a graph $G = (V, E, \omega)$, we do not have to transform G into a equivalent flow network. We can directly operate on the graph with capacities $c(e) = \omega(e)$ for all $e \in E$ [16]. Hypergraphs are generalizations of graph, 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 modelling those edges as described by Lawler [28] (see hyperedge e_2 in Figure 6), we can remove all e', e'' for all $e \in E$ with $|e| = 2$ and add an undirected flow edge between $v_1, v_2 \in e$ (with $v_1 \neq v_2$) with capacity $c(\{v_1, v_2\}) = \omega(e)$.

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

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

(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 $c(v_1, v_2) = \omega(e)$ and $c(v_2, v_1) = \omega(e)$

(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 Figure 11. A hyperedge e of size 2 consists exactly of 4 nodes and 5 edges in $T_L(H)$ (see Figure 6). The same hyperedge induce 2 nodes and 2 edges in $T_G(H)$.

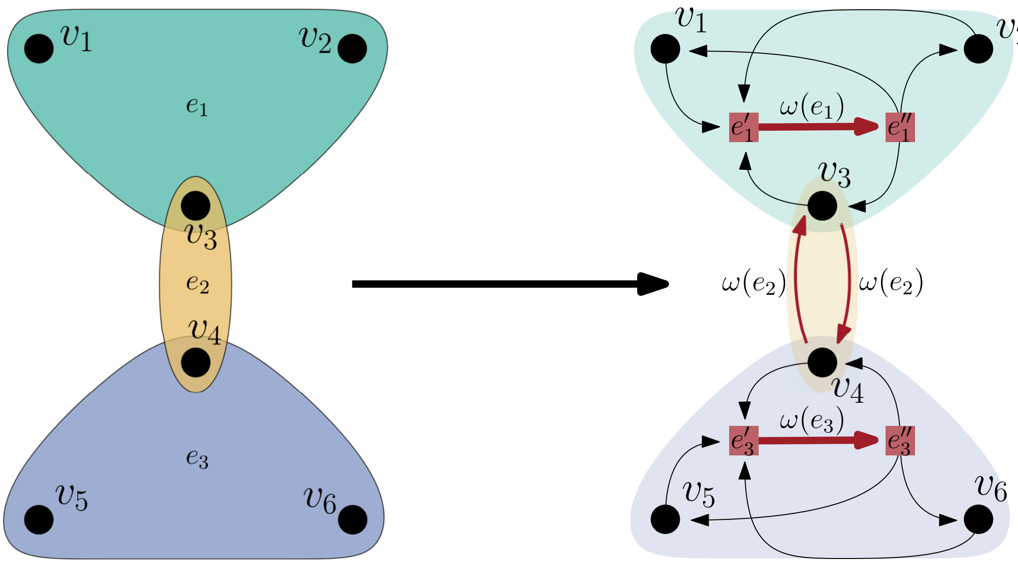

Figure 11: Transformation of a hypergraph into a equivalent flow network by inserting an undirected edge with capacity $\omega(e)$ for each hyperedge of size 2. Note, capacity of the black edges in the flow network is ∞.

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 $T_G(H) = (V_G, E_G, c_G)$.*

Proof. We define the bijective function $\Phi : E_L \to E_G$ as follows

$$\Phi(e', e'') = \begin{cases} (e', e''), \text{if } |e| \neq 2, \\ \{v_1, v_2\}, \text{otherwise (with } v_1, v_2 \in e \text{ and } v_1 \neq v_2) \end{cases}$$

We will show that each (S, T)-cutset A_L of $T_L(H)$ is a (S, T)-cutset $\Phi(A_L)$ of $T_G(H)$ and vice versa. Per defintion $c_L(A_L) = c_G(\Phi(A_L))$ and for each (S, T)-cutset A_G of $T_G(H)$ $c_G(A_G) = c_L(\Phi^{-1}(A_G))$. Therefore, each minimum-capacity (S, T)-cutset of $T_L(H)$ must be a minimum-capacity (S, T)-cutset of $T_G(H)$ and vice versa. In the following let $E^* = \bigcup_{e \in E}\{(e', e'')\}$.
Let $A_L \subseteq E^*$ be a (S, T)-cutset of $T_L(H)$. Assume $\Phi(A_L)$ is not a (S, T)-cutset of $T_G(H)$. Then there exists a path $P_G = \{v_1, \ldots, v_k\}$ connecting S and T in $T_G(H)$ not containing any edge $e \in \Phi(A_L)$. Let P_L be the path in $T_L(H)$ obtained by adding edge $\Phi^{-1}(v_i, v_{i+1}) = (e', e'')$ between all $v_i \in V$ and $v_{i+1} \in V$ in P_G. $\Phi^{-1}(v_i, v_{i+1}) \notin A_L$, because P_G does not contain any edge of $\Phi(A_L)$. P_L connects S and T in $T_L(H)$, which is a contradiction to the assumption that A_L is a (S, T)-cutset.
Let $A_G \subseteq \Phi(E^*)$ be a (S, T)-cutset in $T_G(H)$. Let's assume $\Phi^{-1}(A_G)$ is not a (S, T)-cutset of $T_L(H)$. Then there exists a path $P_L = \{v_1, \ldots, v_k\}$ connecting S and T in $T_L(H)$ not containing any edge $e \in \Phi^{-1}(A_G)$. Let P_G be the path in $T_G(H)$ obtained by removing each edge (v_i, v_{i+1}) with $v_i = e'$ and $v_{i+1} = e''$ and $|e| = 2$ from P_L. Based on the construction of $T_L(H)$ the predecessor of v_i and successor of v_{i+1} must be hypernodes $v_1, v_2 \in e$. Therefore, P_G is a valid path in $T_G(H)$ connecting S and T and not contain any edge of A_G. This is a contradiction to the assumption that A_G is a (S, T)-cutset. □

A minimum-weight (S, T)-cutset of H could 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 with $\Phi^{-1}(v_1, v_2)$. Since their exists a one-one correspondence between the hypernodes of $T_L(H)$ and $T_G(H)$ the corresponding bipartion are all hypernodes *reachable* from a all nodes in S and all not *reachable* from S in the *residual graph* of $T_G(H)$.

4.4. Combining Techniques in a Hybrid Flow Network

On many real world instances the average hyperedge size and hypernode degree are inversely proportional to each other. E.g., if the number of hyperedges is significantly larger than the number of hypernodes the average hypernode degree is usually much larger than 3. Whereas the average hyperedge size is often equal to 2. If the number of hyperedges is nearly equal to the number of hypernodes the average hypernode degree is usually smaller or equal than 3. Whereas the average hyperedge size is often much larger than 2. Of course, we can construct instances where this inversely proportional relationship can not be observed, but on many real world instances we often find the described behaviour.
Currently, we have two differenct modelling approaches which either perform better on low degree hypernode instances or on small hyperedge size instances. Taking our observation from real world instances into account means that either $T_G(H)$ or $T_H(H, V_d(3))$ performs significantly better on a specific instance. It would be preferable to combine the two approaches into one network which performs on most instances best.

Definition 4.3. *Let T_{Hybrid} be a transformation that converts a hypergraph $H = (V, E, c, \omega)$ into a flow network $T_{Hybrid}(H, V') = (V_{Hybrid}, E_{Hybrid}, c_{Hybrid})$, 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* $c_{Hybrid}(e''_1, e'_2) = \infty$ *(clique expansion).*

(iii) $\forall e \in E$ *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) *with capacity* $c_{Hybrid}(v_1, v_2) = \omega(e)$ *and* $c_{Hybrid}(v_2, v_1) = \omega(e)$

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

Figure 12 summarizes all explained transformations of this section. The proof of Theorem 4.2 can be used one-to-one to show that a minimum-capacity (S', T')-cutset of $T_H(H, V')$ is equal with a minimum-capacity (S', T')-cutset of $T_{\text{Hybrid}}(H, V')$ (for definition of S' and T' see Theorem 4.1). It follows with Lemma 4.1 that this is equal with a minimum-weight (S, T)-cutset of H.
Per definition of $T_{\text{Hybrid}}(H, V')$ we prefer a hyperedge removal over a hypernode removal. E.g., if a hypernode has a degree smaller or equal than 3, we only remove it, if there is no hyperedge $e \in I(v)$ with $|e| = 2$. The reason is that a hyperedge removal always decrease the number of nodes and edges more than a hypernode removal.
The minimum-weight (S, T)-cutset of H can be calculated with the same technique described in Section 4.3. Let's define with $(A, V \setminus A)$ 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 4.2).

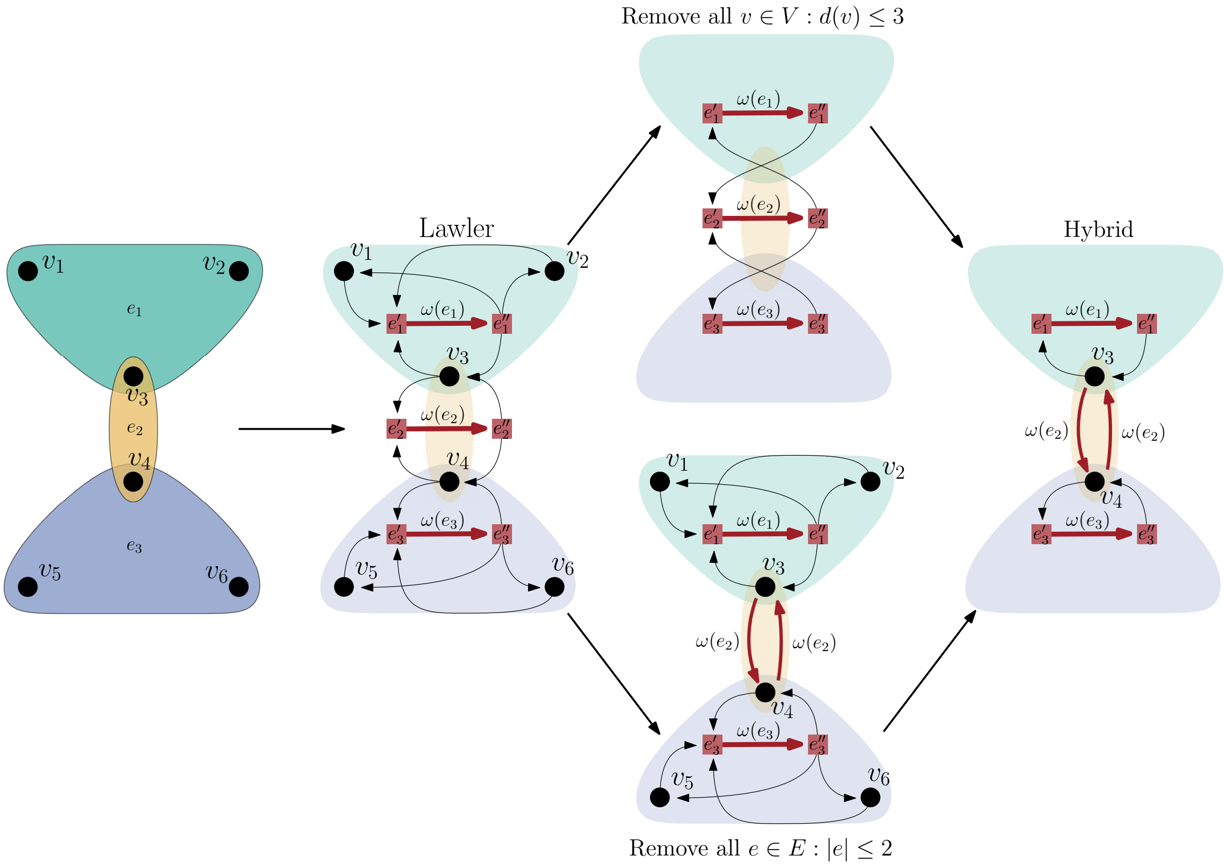

Figure 12: Illustration of all presented transformations of a hypergraph into a flow network.

5. Using Max-Flow-Min-Cut Computations as Refinement Strategy

We will give now a detailed overview of our flow-based refinement framework. The main idea is to extract a subhypergraph $H_{V'}$ out of a hypergraph H, which is already partitioned into k blocks. V' is choosen in such a way that it is a subset of two adjacent blocks V_i and V_j. We will show how to configure the sources S and sinks T of the corresponding flow network such that a minimum (S,T)-bipartition of $H_{V'}$ improves the connectivity metric of H (see Section 5.1). Further, we describe how the ideas of Sanders and Schulz [36] (see Section 3.3) could be adapted to work in a n-level hypergraph partitioner, called *KaHyPar* (see Section 5.2 and 5.3).

5.1. Modelling Sources and Sinks

Let $H = (V, E, c, \omega)$ be a hypergraph and $B_1 := (V_1, V_2)$ be a bipartition. $H_{V'}$ is the subhypergraph induced by $V' \subseteq V$. Further, let $E_\emptyset = \{e \cap V' \mid e \in E : e \cap V' = \emptyset\}$ be the set of all hyperedges contained in H, but not in $H_{V'}$. $T_L(H_{V'})$ (see Section 3.2) is the flow network induced by $H_{V'}$ with a source set S and a sink set T. Let (V_1', V_2') be the minimum (S,T)-bipartition obtained by a maximum (S,T)-flow calculation on $T_L(H_{V'})$ with f as maximum flow function. We can extend the bipartition (V_1', V_2') of $H_{V'}$ to a bipartition $B_2 := (V_1 \setminus V' \cup V_1',\ V_2 \setminus V' \cup V_2')$ of H. Finally, we define the cut on subhypergraph $H_{V'}$ related to a bipartion (V_1, V_2):

$$\omega_{H_{V'}}(V_1, V_2) := \sum_{e \in E(V_1, V_2) \setminus E_\emptyset} \omega(e)$$

Some will wonder about the defnition of the cut $\omega_{H_{V'}}$ over the cut edges of H. A cut hyperedge e of H must not necessarily be a cut hyperedge of $H_{V'}$. E.g., if $e = \{v_1, v_2\}$ with $v_1 \in V_1$ and $v_2 \in V_2$, but $v_1 \in V'$ and $v_2 \notin V'$. Then e is cut in H, but not in $H_{V'}$, because v_2 is removed from e per definition of $E_{V'}$. However, the reason that we still define e as cut hyperedge of $H_{V'}$ has to do with our problem statement, which we will define as follows:

Problem 5.1. *How do we have to define the source set S and sink set T for a subhypergraph $H_{V'}$ (with $V' \subseteq V$) and a bipartition B_1, such that after a maximum (S,T)-flow calculation (with f as maximum flow function) the resulting minimum (S,T)-bipartition B_2 of H satisfy the following conditions:*

(i) $\omega_H(B_2) \leq \omega_H(B_1)$

(ii) $\Delta_H := \omega_H(B_1) - \omega_H(B_2) = \omega_{H_{V'}}(B_1) - |f| =: \Delta_{H_{V'}}$

The first condition ensures that a maximum (S,T)-flow calculation on $T_L(H_{V'})$ never decrease the cut of H. The existence of the second condition has practical reasons. First, we can simply update the cut metric via $\omega_H(B_2) = w_H(B_1) - \Delta_{H_{V'}}$, instead of summing up the weight of all cut hyperedges. Since, we have to setup the subhypergraph $H_{V'}$ before each maximum flow computation we can implicitly calculate $\omega_{H_{V'}}(B_1)$. Therefore, the cut metric can be updated after a *Max-Flow-Min-Cut* computation in constant time instead of $\mathcal{O}(|E|)$. On the other hand, we can assert the correctness of our own maximum flow algorithm. If $\Delta_H \neq \Delta_{H_{V'}}$, then with high probability our flow algorithm is incorrect. Also, the reason why we define $\omega_{H_{V'}}(V_1, V_2)$ over the cut hyperedges of H is due to the fact that the equality

$$\Delta_H := \omega_H(B_1) - \omega_H(B_2) = \omega_{H_{V'}}(B_1) - \omega_{H_{V'}}(B_2)$$

holds. If we are able to show that $|f| = \omega_{H_{V'}}(B_2)$, we simultanously show that our source and sink set modelling approach satisfies condition (ii) $\Delta_H = \Delta_{H_{V'}}$.

We will now present a solution for our problem statement. First, we show how S and T can be choosen to satisfy condition (i). Afterwards, we extend S and T with additional nodes to fullfil condition (ii). Finally, we show how S and T can be modified, such that we can obtain smaller cuts on H and simultanously satisfy condition (i) and (ii) of our problem statement.

Let $V' \subseteq V$ and $\delta B = \{e \in E \mid \exists u, v \in e : u \in V' \wedge v \notin V'\}$ be the set of all *Border Hyperedges*. For a bipartition (V_1, V_2) of H, we say $v \in V_1$ is a source node of the flow network $T_L(H_{V'})$, if there exists a hyperedge $e \in \delta B$ containing v and at least one other node $u \in V_1$ with $u \notin V'$. More formal:

$$S_1 = \{s \in V' \cap V_1 \mid \exists v \notin V' : \exists e \in \delta B : v \in V_1 \ \wedge \ s, v \in e\} \tag{5.1}$$

$$T_1 = \{t \in V' \cap V_2 \mid \exists v \notin V' : \exists e \in \delta B : v \in V_2 \ \wedge \ v, t \in e\} \tag{5.2}$$

An example of a *Max-Flow-Min-Cut* computation on $H_{V'}$ with S and T as source and sink set is illustrated in Figure 13.

Lemma 5.1. *Let B_1 be a bipartition of H and $T_L(H_{V'})$ the flow network of subhypergraph $H_{V'}$ with S and T as defined in Equation 5.1 and 5.2 (with $V' \subseteq V$). Let B_2 be the bipartition obtained by a maximum (S,T)-flow computation on $T_L(H_{V'})$. Then, $\omega_H(B_2) \leq \omega_H(B_1)$.*

Proof. A maximum (S,T)-flow computation on $T_L(H_{V'})$ yields to a minimum (S,T)-cutset on $H_{V'}$ [16]. Thus, for all hyperedges $e \notin \delta B \cup E_\emptyset$, which are cut in B_2, the sum of their weight must be less or equal than the sum of all cut hyperedges $e \notin \delta B \cup E_\emptyset$ of bipartition B_1. We have to show that a non-cut hyperedge $e \in \delta B$ of $B_1 = (V_1, V_2)$ cannot become a cut hyperedge of $B_2 = (V_1', V_2')$. Let $e \in \delta B$ be such a hyperedge. e must be either a subset of V_1 or V_2,

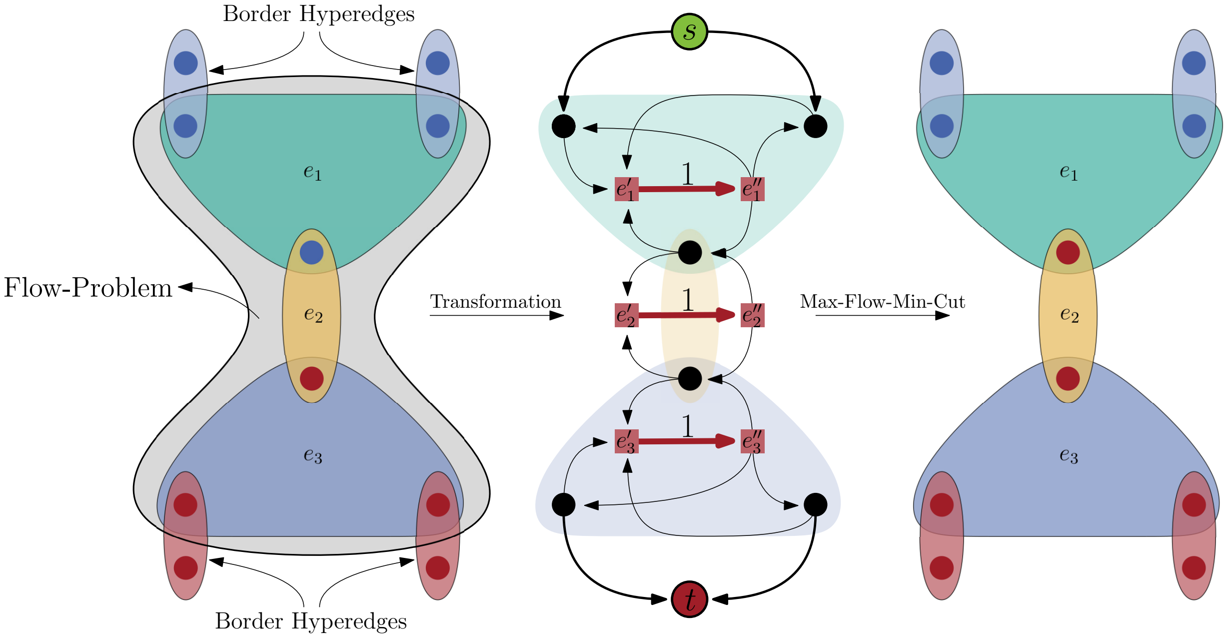

Figure 13: Example how *Border Hyperedges* are modelled as sources and sinks.

otherwise e is a cut hyperedge. Let $e \subseteq V_1$, then $e \cap V' \subseteq S$ (see Equation 5.1). Defining a node $s \in S$ as source node means that it cannot change its block after a *Max-Flow-Min-Cut* computation. Therefore, $e \subseteq V_1$ and $e \subseteq V_1' \Rightarrow e$ is a non-cut hyperedge in B_2. The proof for $e \subseteq V_2$ is equivalent $\Rightarrow \omega_H(B_2) \leq \omega_H(B_1)$. □

In the next step we will show how S and T can be extended to satisfy condition (ii) of Problem 5.1. Currently, $|f| \leq \omega_{H_{V'}}(B_2)$ (without a prove). Obviously, some nodes are missing in S and T. Consider Figure 14 for a understanding which nodes are missing. Transformation 1 illustrates our current modelling approach defined in Equation 5.1 and 5.2. The maximum flow on this network is $|f| = 1$, but the resulting minimum (S,T)-bipartition B_2 induced a cut of $\omega_{H_{V'}}(B_2) = 2$. This due to the fact that e_1 and e_3 are cut hyperedges in H, but non-cut hyperedges in $H_{V'}$. The actual cut of $H_{V'}$ is therefore 1 (instead of 2) and this is also a minimum (S,T)-cut. Transformation 2 illustrates the adapted modelling approach for cut hyperedges of H which are non-cut hyperedges in $H_{V'}$. For each hyperedge $e \in \delta B$ with $e \cap V' \subseteq V_2$ and $e \setminus V' \cap V_1 \neq \emptyset$, we add the *incomming hyperedge node* e' to S. More formal:

$$S = S_1 \cup \{e' \mid e \cap V' \subseteq V_2 \ \wedge \ e \setminus V' \cap V_1 \neq \emptyset\} \tag{5.3}$$
$$T = T_1 \cup \{e'' \mid e \cap V' \subseteq V_1 \ \wedge \ e \setminus V' \cap V_2 \neq \emptyset\} \tag{5.4}$$

Lemma 5.2. *Let B_1 be a bipartition of H and $T_L(H_{V'})$ the flow network of subhypergraph $H_{V'}$ with S and T as defined in Equation 5.3 and 5.4 (with $V' \subseteq V$). Let B_2 be the bipartition obtained by a maximum (S,T)-flow computation on $T_L(H_{V'})$ with f as maximum flow function. Then, $\omega_{H_{V'}}(B_2) = |f|$ ($\Rightarrow \Delta_H = \Delta_{H_{V'}}$).*

Proof. Let $V'' = \bigcup_{e \in \delta B} e \backslash V'$ be the set of all hypernodes contained in a *border hyperedge*, but not in V'. Let $H_{V' \cup V''}$ be the subhypergraph obtained by extending $H_{V'}$ with all missing hypernodes of *border hyperedges*. Let $T_L(H_{V' \cup V''})$ be the resulting flow network with $S' = S_1 \ \cup \ (V'' \cap V_1)$ and $T' = T_1 \ \cup \ (V'' \cap V_2)$ as sources and sinks. Further, let f' be a maximum (S',T')-flow on $T_L(H_{V' \cup V''})$ and B_2 be the corresponding minimum (S',T')-bipartition. Because all hypernodes which are part of a hyperedge in H and also in $H_{V'}$ are fully contained in $H_{V' \cup V''}$ the equality

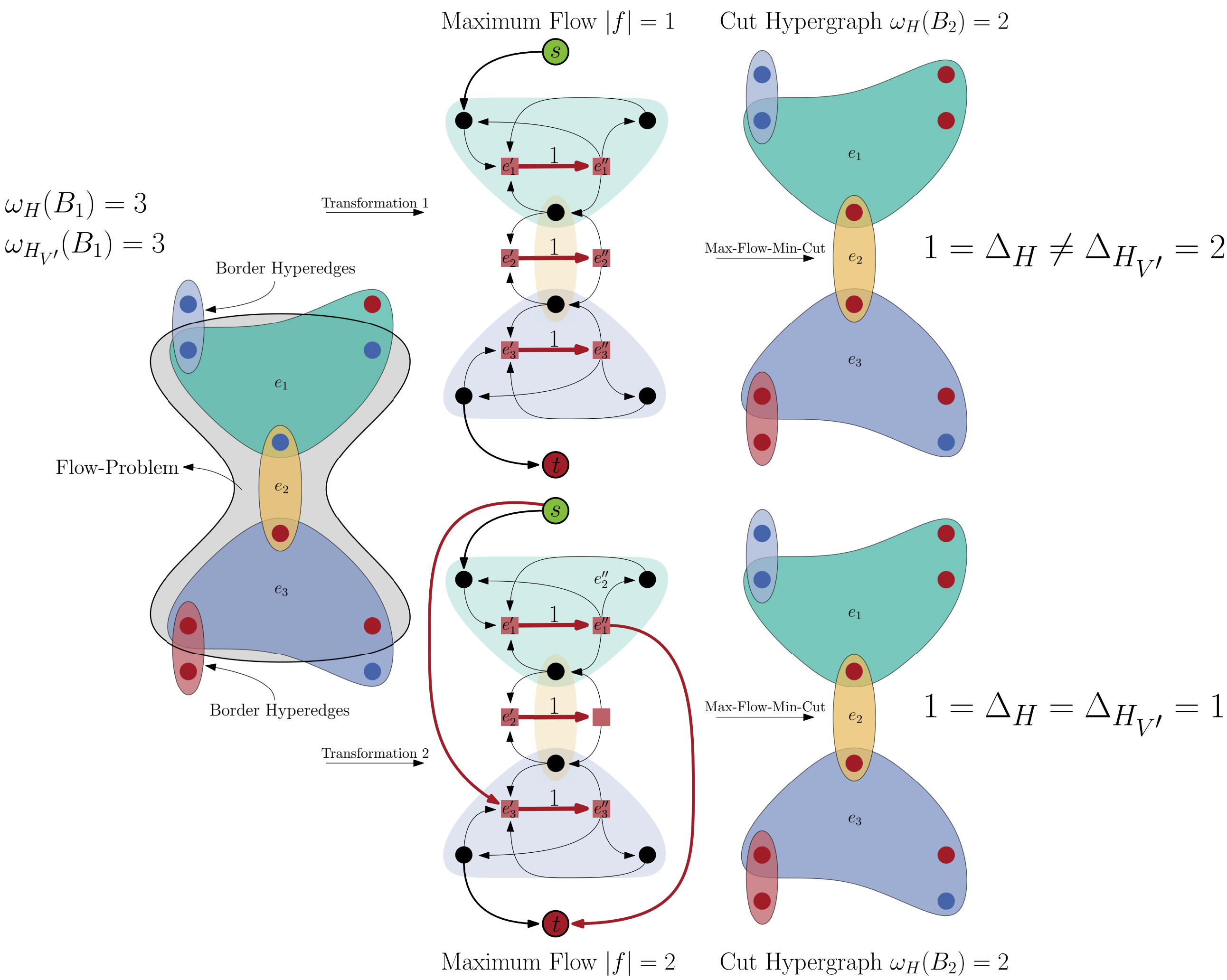

Figure 14: In this example e_1 and e_3 are cut hyperedges of the hypergraph, but not of the sub-hypergraph induced by the flow problem. Modelling the *outgoing* resp. *incomming* hyperedge node of e_1 resp. e_2 as sink resp. source ensures that $\Delta_H = \Delta_{H_{V'}}$.

$|f'| = \omega_{H_{V'}}(B_2)$ holds. In the following we present a technique with which we can obtain a new flow network $T_L(H_{V' \cup V'' \setminus \{v\}})$ with $v \in V''$ and S'' and T'' as sources and sinks. Simultanously we map the maximum (S', T')-flow f' of $T_L(H_{V' \cup V''})$ to a maximum (S'', T'')-flow of $T_L(H_{V' \cup V'' \setminus \{v\}})$ with $|f'| = |f''|$. Applying this technique successively on all nodes $v \in V''$ will result in the flow network $T_L(H_{V'})$ with S and T as sources and sinks from Equation 5.3 and 5.4.
A hypernode $v \in V''$ is either a source or a sink. We will show how to remove a source hypernode $v \in V'' \cap S'$. We define $S'' := S'$, $T'' := T'$ and $f'' := f'$. In order to remove $v \in V''$ we have to distingush two cases based on a incident hyperedge $e \in I(v)$:

$\mathbf{e \cap S \setminus \{v\} \neq \emptyset}$: Then there exists a hypernode $u \in e \cap S$ with $u \neq v$. We define $f''(u, e') = f''(u, e') + f'(v, e')$ and $f''(s, u) = f''(s, u) + f'(v, e')$.

$\mathbf{e \cap S \setminus \{v\} = \emptyset}$: In this case e must be a cut hyperedge in H, but not in $H_{V'}$, otherwise there would exist a hypernode $u \in e \cap S$ (see Equation 5.1). We define $S'' = S'' \cup \{e'\}$. Simultanously, we set $f''(s, e') = f'(v, e')$.

The two cases are illustrated in Figure 15. We can remove v from $T_L(H_{V' \cup V''})$ after applying this technique for all $e \in I(v)$. The cases for a vertex $v \in V'' \cap T'$ are equivalent. f'' is a valid flow function per construction and $|f'| = |f''|$. Also f'' is maximum (S'', T'')-flow on $T_L(H_{V' \cup V'' \setminus \{v\}})$, otherwise there would exist a augmenting path in the residual graph $T_L(H_{V' \cup V'' \setminus \{v\}})$ which we

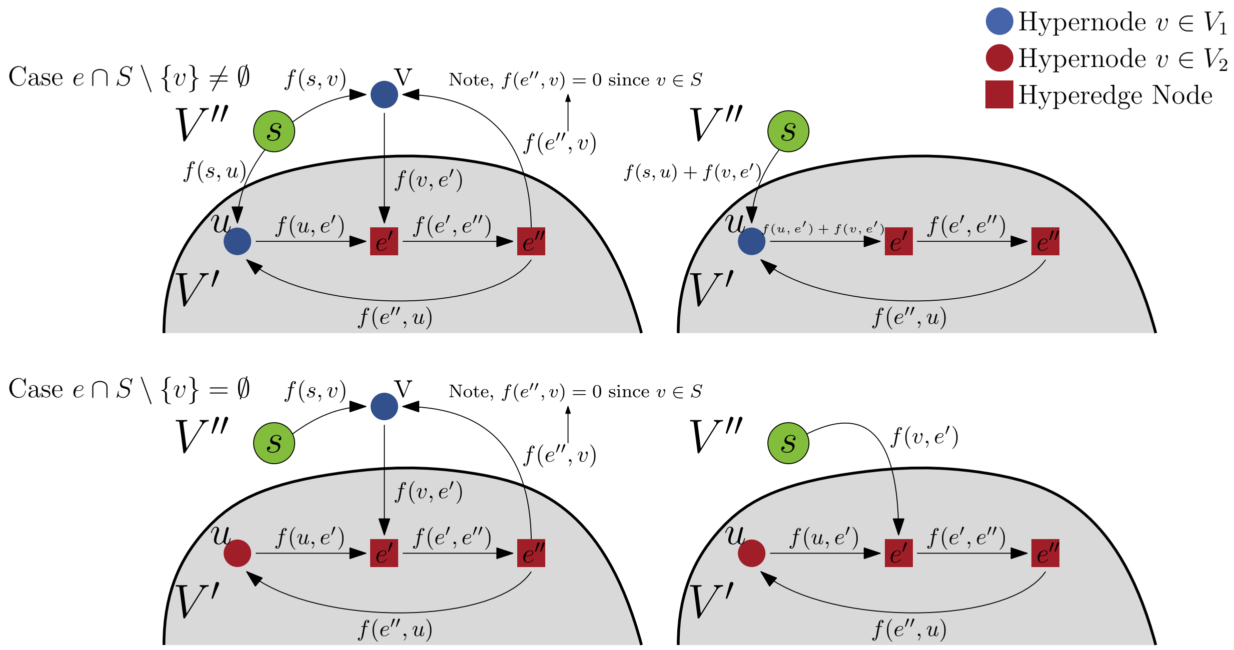

Figure 15: Illustration of the two cases presented in proof of Lemma 5.2 in order to remove a hypernode $v \in V'' \cap S$ from $T_L(H_{V' \cup V''})$.

can map to a augmenting path in $T_L(H_{V' \cup V''})$ (without a proof). We can successively remove all $v \in V''$ from $T_L(H_{V' \cup V''})$ with this method.
The resulting flow network is $T_L(H_{V'})$. For each $e \in E$ which is cut in H, but not in $H_{V'}$, we have added the corresponding *incomming hyperedge node* e' resp. *outgoing hyperedge node* e'' to S'' resp. T''. Therefore, S'' and T'' are equal with S and T as defined in Equation 5.3 and 5.4. Finally, the flow function f'' is a maximum (S, T)-flow on $T_L(H_{V'})$ and $|f''| = |f'| = \omega_{H_{V'}}(B_2)$ per construction. □

With our current modelling approach we are able to satisfy all conditions of our problem statement. However, in some cases we define hypernodes as source resp. sink which are unnecessary. Consider Figure 16 for an illustration. Hyperedge e_1 is cut in $H_{V'}$ and contains hypernodes from both blocks, which are not in the flow problem. Regardless of the maximum (S, T)-flow computation on $H_{V'}$ we can not remove e_1 from cut in H. Using our suggested source and sink modelling has as consequence that e_1 and e_2 are still cut after a *Max-Flow-Min-Cut* computation (see *Transformation 1* in Figure 16), because we define all vertices of e_1 as source resp. sinks. Another approach is to define hyperedges which are cut of $H_{V'}$ and are also of H the *incomming* resp. *outgoing hyperedge node* as source resp. sink (see *Transformation 2* in Figure 16). In our example all hypernodes of e_1 are still able to move and a *Max-Flow-Min-Cut* computation removes e_2 from cut.
To define our final source and sink set, we split the set of all *border hyperedges* into three different disjoint subsets as follows:

(i) $\delta B_1 = \{e \in \delta B \mid e \subseteq V_1 \;\vee\; e \subseteq V_2\}$
(ii) $\delta B_2 = \{e \in \delta B \mid e \cap V' \not\subseteq V_1 \;\wedge\; e \cap V' \not\subseteq V_2\}$
(iii) $\delta B_3 = \{e \in \delta B \setminus \delta B_1 \mid (e \cap V' \subseteq V_1 \;\vee\; e \cap V' \subseteq V_2\}$

δB_1 contains all non-cut *border hyperedges* of H. δB_2 contains all *cut border hyperedges* of H, which are also cut in $H_{V'}$ and δB_3 contains all *cut border hyperedges* of H, which are non-cut in $H_{V'}$.

$$S = \bigcup_{\substack{e \in \delta B_1 \\ e \subseteq V_1}} e \cap V' \;\cup \bigcup_{\substack{e \in \delta B_2 \cup \delta B_3 \\ e \setminus V' \cap V_1 \neq \emptyset}} \{e'\} \tag{5.5}$$

$$T = \bigcup_{\substack{e \in \delta B_1 \\ e \subseteq V_2}} e \cap V' \;\cup \bigcup_{\substack{e \in \delta B_2 \cup \delta B_3 \\ e \setminus V' \cap V_2 \neq \emptyset}} \{e''\} \tag{5.6}$$

Equation 5.5 and 5.6 are illustrated in Figure 17. A *Max-Flow-Min-Cut* computation on $T_L(H_{V'})$ with S and T as defined in Equation 5.5 and 5.6 satisfy condition (i) and (ii) of Problem 5.1. This can be prooven with similiar techniques used in the proof of Lemma 5.1 and 5.2. A maximum (S,T)-flow calculation yields to a minimum (S,T)-cut on $H_{V'}$. A non-cut hyperedge $e \in \delta B_1$ can not become a cut hyperedge after a *Max-Flow-Min-Cut* computation, because we still define all vertices of non-cut hyperedges of H and $H_{V'}$ as sources resp. sinks. Therefore, $\omega_H(B_2) \leq \omega_H(B_1)$. We can proof Lemma 5.2 for our new source and sink sets if we adapt the conditions of the cases for a hyperedge $e \in I(v)$ based on the set δB_1, δB_2 and δB_3 where e is contained. If $e \in \delta B_1$, then there must exists a hypernode $u \in e \cap S \setminus \{v\}$ on which we apply the first case (Case 1: $e \cap S \setminus \{v\} \neq \emptyset$). For all $e \in \delta B_2 \cup \delta B_3$, we simply apply the second case (Case 2: $e \cap S \setminus \{v\} = \emptyset$). After removing all hypernodes $v \in V''$ the resulting network is $T_L(H_{V'})$ with S and T as defined in Equation 5.5 and 5.6. Further, the flow function f'' is a maximum (S,T)-flow on $T_L(H_{V'})$ with $|f''| = |f'| = \omega_{H_{V'}}(B_2) \Rightarrow \Delta_H = \Delta_{H_{V'}}$.
Finally, we want to show that for a minimum (S',T')-bipartition B_2 with S' and T' as defined in Equation 5.5 and 5.6 and a minimum (S,T)-bipartition B_3 with S and T as defined in Equation 5.3 and 5.4 calculated with flow network $T_L(H_{V'})$ the inequality $\omega_H(B_2) \leq \omega_H(B_3)$ holds. For this propose we need a preparing lemma.

Lemma 5.3. *Let $G = (V, E, c)$ be a flow network with sources S and sinks T. Further, let $S' \subseteq S$ and $T' \subseteq T$. The value of a maximum (S',T')-flow f' is less or equal than the value of a maximum (S,T)-flow f. More formal, $|f'| \leq |f|$.*

Proof. Assume $|f'| > |f|$. Then, we can simply set $f = f'$, because $S' \subseteq S$ and $T' \subseteq T$. But this is a contradiction to assumption that f is a maximum (S,T)-flow on G. Therefore, $|f'| \leq |f|$. □

In the following theorem, we denote with S and T the source and sink sets as defined in Equation 5.3 and 5.4 and with S' and T' the source and sink sets as defined in Equation 5.5 and 5.6.

Theorem 5.1. *Let H be a hypergraph and $H_{V'}$ be the subhypergraph induced by the subset $V' \subseteq V$. Further, let B_1 be the current bipartition of H. For a minimum (S',T')-bipartition B_2 and a minimum (S,T)-bipartition B_3 obtained by a maximum (S',T')- resp. (S,T)-flow calculation on $T_L(H_{V'})$ the inequality $\omega_H(B_2) \leq \omega_H(B_3) \leq \omega_H(B_1)$ holds.*

Proof. Let $(\bar{S}', \bar{T}')$ resp. $(\bar{S}, \bar{T})$ be the sets obtained by removing all *incomming* and *outgoing hyperedge nodes* e' and e'' from (S', T') resp. (S,T). It holds that $\bar{S}' \subseteq \bar{S}$ and $\bar{T}' \subseteq \bar{T}$. Afterwards, we extend the subhypergraph $H_{V'}$ with all hypernodes $V'' = \bigcup_{e \in \delta B} e \setminus V'$ and obtain subhypergraph $H_{V' \cup V''}$ with flow network $T_L(H_{V' \cup V''})$. Also we extend $(\bar{S}', \bar{T}')$ and $(\bar{S}, \bar{T})$ exactly in the same way as in the proof of Theorem 5.2. With the *Max-Flow-Min-Cut*-Theorem [16] we can conclude that the cut value $\omega_{H_{V'}}(B_2)$ of a minimum $(\bar{S}', \bar{T}')$-bipartition B_2 on $H_{V'}$ is equal with the value of a maximum $(\bar{S}', \bar{T}')$-flow f' on $T_L(H_{V' \cup V''})$. The same holds for a minimum $(\bar{S}, \bar{T})$-bipartition B_3 and a maximum $(\bar{S}, \bar{T})$-flow f. After extending $(\bar{S}', \bar{T}')$

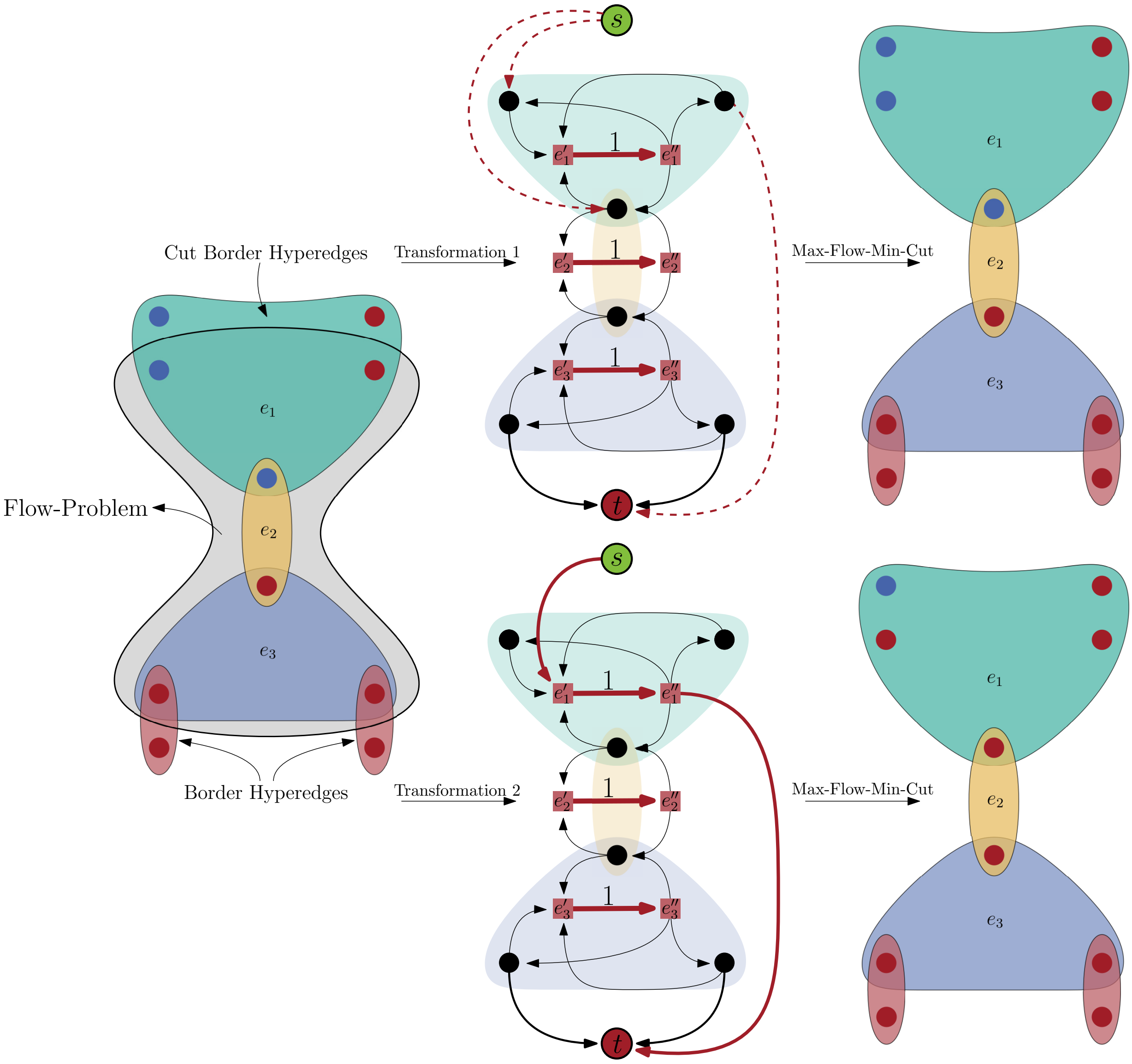

Figure 16: Example how *Cut Border Hyperedges* are modelled as sources and sinks. In this example e_1 contains node from block V_1 and V_2 not contained in the flow problem. Therefore, we can not remove e_1 from cut. Treating e_1 as a *Border Hyperedge* would result in Transformation 1. This has the consequence that we are not able to remove e_2 from cut with a *Max-Flow-Min-Cut* computation. Defining the *incomming* resp. *outgoing* hyperedge of e_1 as source resp. sinks allows the corresponding hypernodes of e_1 still to move. The consequence is that we can remove e_2 from cut with a *Max-Flow-Min-Cut* computation in Transformation 2.

resp. $(\bar{S}, \bar{T})$ with all hypernodes of V'' $\bar{S}' \subseteq \bar{S}$ and $\bar{T}' \subseteq \bar{T}$ still holds. With Lemma 5.3 and the *Max-Flow-Min-Cut*-Theorem follows $\omega_{H_{V'}}(B_2) = |f'| \leq |f| = \omega_{H_{V'}}(B_3)$.
We can transform $(\bar{S}', \bar{T}')$ resp. $(\bar{S}, \bar{T})$ and flow network $T_L(H_{V' \cup V''})$ back to $T_L(H_{V'})$ with (S', T') resp. (S, T) as source and sink sets with the technique described in the proof of Theorem 5.2 and in the sketch of the proof for our new source and sink sets (see Equation 5.5 and 5.6). Therefore, the inequality still holds for bipartitions B_2 and B_3 obtained by a maximum (S', T')- and (S, T)-flow calculation on $T_L(H_{V'})$. Finally, it follows

$$\begin{aligned}\omega_H(B_2) &\overset{\text{Problem 5.1(ii)}}{=} \omega_H(B_1) - \omega_{H_{V'}}(B_1) + |f'| \\ &\overset{\text{Lemma 5.3}}{\leq} \omega_H(B_1) - \omega_{H_{V'}}(B_1) + |f| \\ &\overset{\text{Problem 5.1(ii)}}{=} \omega_H(B_3) \overset{\text{Problem 5.1(i)}}{\leq} \omega_H(B_1)\end{aligned}$$

□

We are now able to extract a subhypergraph $H_{V'}$ out of a already bipartitioned hypergraph H and calculate a minimum (S, T)-bipartition of $H_{V'}$ with S and T as defined in Equation 5.5 and 5.6. The resulting bipartition induced a new cut on H smaller or equal than the old cut. Further, we show with our modelling technique of S and T that Δ_H can be calculated with the help of the value of a maximum (S, T)-flow computation on $T_L(H_{V'})$. Additionaly, we demonstrate that a different modelling approach of S and T which satisfy both conditions of Problem 5.1 can lead to an improved cut quality of the minimum (S, T)-bipartition of the original hypergraph H.

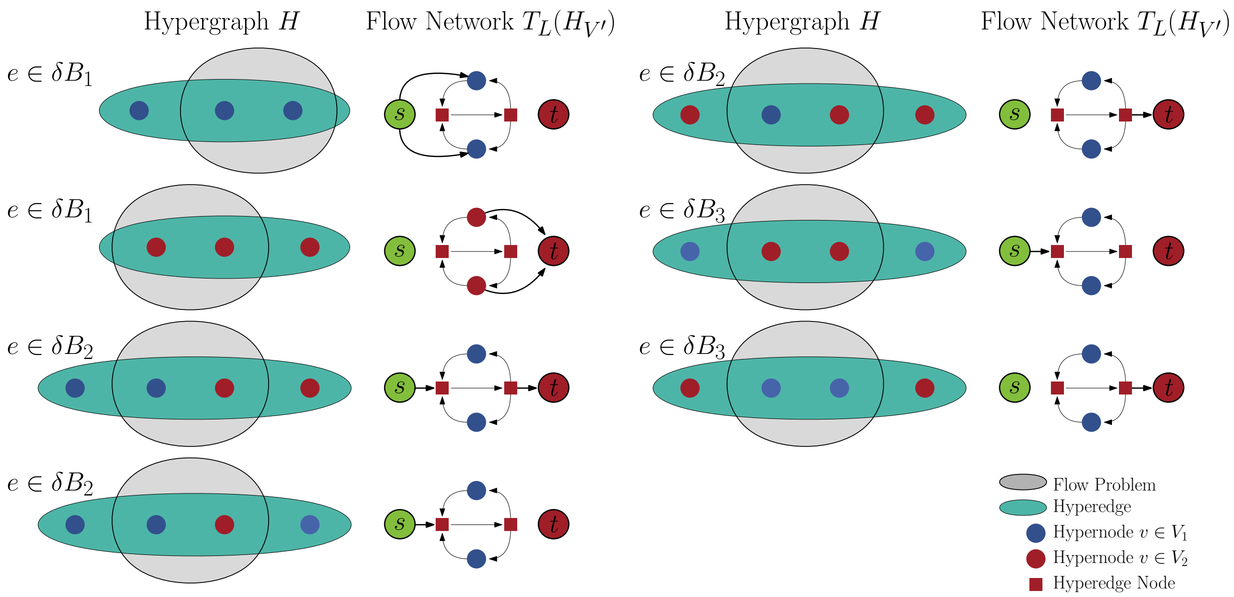

Figure 17: Illustration of source and sink set modelling defined in Equation 5.5 and 5.6.

In Section 4.3 we described how to remove hyperedges of size $|e| = 2$ by adding an undirected flow edge between the corresponding vertices $u, v \in e$. However, if the incomming or outgoing hyperedge node is a source or a sink node, we can not directly remove the hyperedge nodes. There are two special cases which are illustrated in Figure 18. This situation occurs if one of the two vertices is part of the flow problem and one not. In case, if the incomming hyperedge node e' is a source node, we only remove the outgoing hyperedge node e'' and add a directed flow edge from e' to v with capacity $\omega(e)$. In the second case, if the outgoing hyperedge node

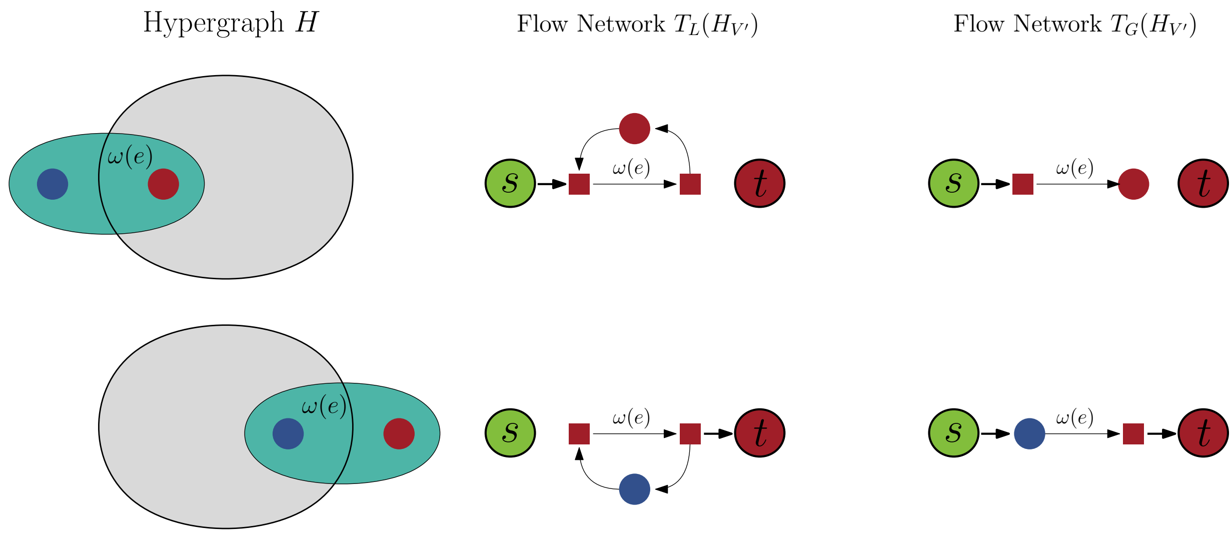

Figure 18: Illustration of modelling hyperedges of size two if the incomming or outgoing hyperedge node is a source or a sink node of the flow problem.

e'' is a sink node, we only remove the incomming hyperedge node e' and add a directed flow edge from v to e'' with capacity $\omega(e)$.

With the given approach we are able to optimize the cut metric of a given bipartition of a hypergraph H. We can transfer those results in order to improve a k-way partition $\Pi = (V_1, \ldots, V_k)$, if the objective is the connectivity metric. 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 simultanously optimize the connectivity metric of H. The reduction of the cut of $H_{V'}$ is then equal with the reduction of the connectivity metric of H.

5.2. Most Balanced Minimum Cuts on Hypergraphs

Picard and Queyranne [33] showed 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. The corresponding algorithm is presented in Section 3.3.3.
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 are able to find all minimum-capacities (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. This a problem, if we want to enumerate all minimum-weight (s,t)-bipartitions. The solution for this problem is quite simple. After a maximum (s,t)-flow calculation on one of the two mentioned networks we insert all removed hypernodes with their corresponding edges again into the residual graph of our flow network. The maximum (s,t)-flow is still maximal, otherwise we would had found an *augmenting path* on the flow network before. We are now able to compute all minimum-weight (s,t)-bipartitions the same way as with $T_L(H)$.

5.3. A Direct K-Way Flow-Based Refinement Framework

We have described how a hypergraph H could be transformed into a flow network $T_L(H)$ such that every minimum-capacity (s,t)-cutset on $T_L(H)$ is a minimum-weight (s,t)-cutset on H (see Section 3.2). Additionaly, we present techniques to sparsify the flow network $T_L(H)$ [28] in order to reduce the complexity of the flow problem (see Section 4). Further, we show how to configure the source and sink sets on the flow network of a subhypergraph $H_{V'}$ (with $V' \subseteq V$) such that a *Max-Flow-Min-Cut* computation improves a given bipartition of H (see Section 5.1). Finally, we are able to enumerate all minmum-weight (s,t)-cutsets of a subhypergraph $H_{V'}$ with one maximum (s,t)-flow calculation [33].
We will now present our direct k-way flow-based refinement framework which we integrated into the n-level hypergraph partitioner *KaHyPar* [21] (see Section 3.4.2). Our flow-based refinement approach optimizes the *connectivity* metric. We have used a similiar architecture as proposed by Sanders and Schulz [36] (see Section 3.3). The basic concepts of the framework are illustrated in Figure 19.
Our maximum flow calculations are embedded into an *Active Block Scheduling* refinement [22] (see Section 3.3.4). Each time we use flows to improve the connectivity metric of a given k-way partition Π we construct 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. In order to do that we use two *BFS*, one only touches hypernodes of V_i and the second only touches hypernodes of V_j. The *BFS* are initialized with all hypernodes contained in a cut hyperedge of the bipartition (V_i, V_j). A pairwise flow-based refinement is embedded into the *adaptive flow iterations* strategy [36] (see Section 3.3.2) which also determines the size of the flow problem.
After we define the subhypergraph $H_{V'}$, which we use to improve the bipartition (V_i, V_j) on H, we construct one of the flow networks proposed in Section 4 with sources S and sinks T defined in Section 5.1. We implemented two maximum flow algorithms. One is a slightly modified *augmenting path* algorithm of Emdond & Karp [14] (see Section 3.1.1) and the second is the *Push-Relabel* algorithm of Goldberg & Tarjan [11, 19] (see Section 3.1.2). 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* heurisitc [11]. We do not use an external implementation of a maximum flow algorithm. Since the $I \backslash O$ of writing a flow problem to memory and reading the solution would significant slowdown the performance of our algorithm, because we have to solve an enormous number of flow problems during the *Active Block Scheduling* refinement. After determining a maximum (S,T)-flow on our flow network we iterate over all minimum (S,T)-bipartitions of $H_{V'}$ [33] and choose the *Most Balanced Minimum Cut* (see Section 3.3.3 and 5.2) according to our *balanced contraint*.
KaHyPar is a n-level hypergraph partitioner ($|V| = n$) taking the multilevel paradigm to its extreme by removing only a single vertex in every level of the hierarchy [1] (see Section 3.4.2). During the refinement step n local searches are instantiated. Therefore, using our flow-based refinement as local search algorithm on each level is not applicable, because the performance slowdown would be tremendous. Therefore, we introduce *Flow Execution Policies*. One 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 tunning 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 executes the *Active Block Scheduling* refinement on the last level of the hierarchy. In all remaining levels where no flow is executed, we can use a *FM*-based local search algorithm [1, 15, 35] (see Section 3.3.4).

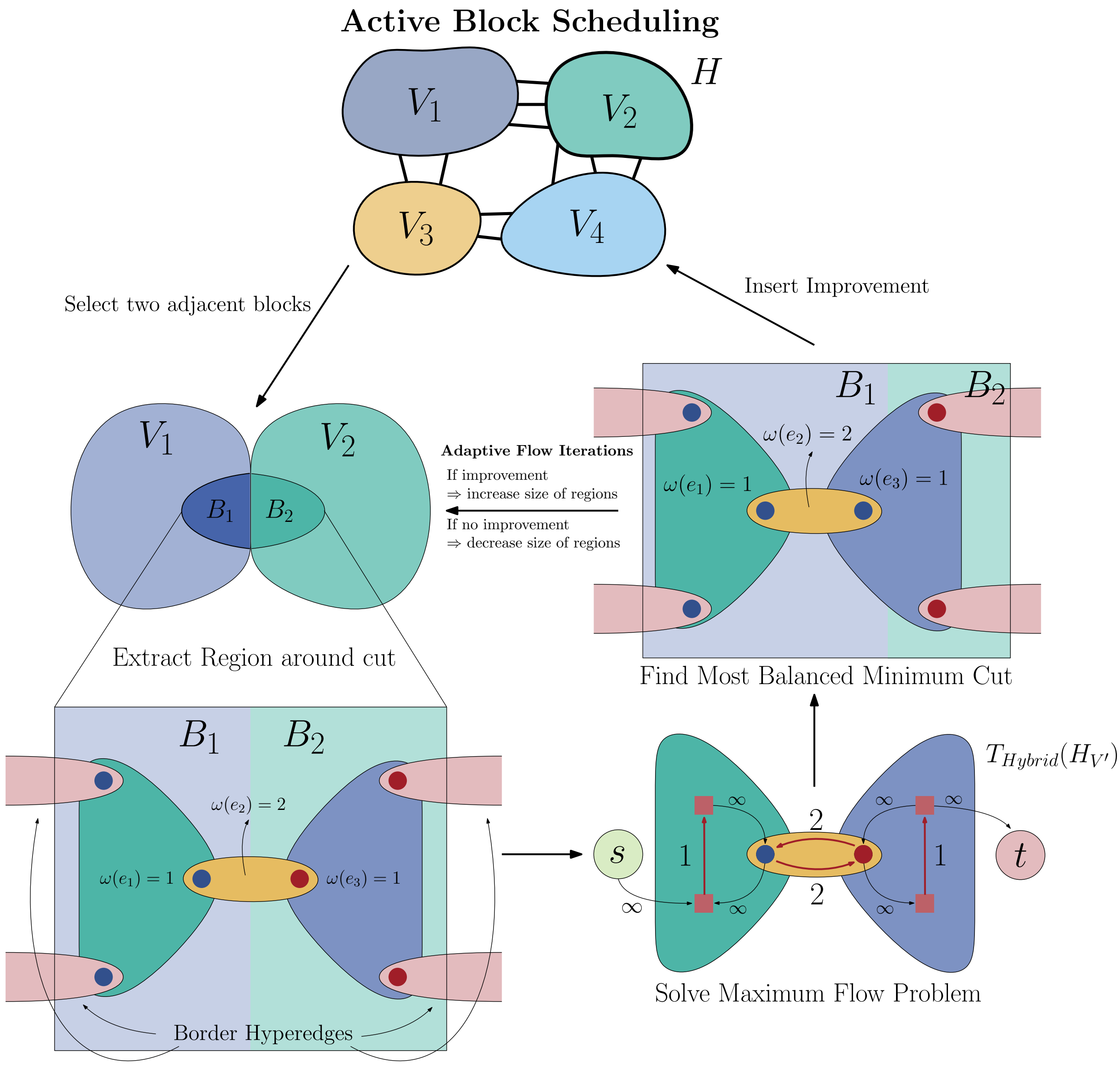

Figure 19: Illustration of our flow-based refinement framework on hypergraphs.

An observation during the implementation of this framework was that only a minority of the pairwise refinements based on flows yields to an improvement of the connectivity metric on hypergraph H. Therefore, we introduce several rules which might prevent unnecessary flow executions to improve the effectiveness ratio by simultanously speed up the runtime.

(R1) If a flow-based refinement did not lead to an improvement on two blocks in all previous executions, we use flows only in the first iteration of *Active Block Scheduling* on these blocks.

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

(R3) If the value of the cut of a minimum (S,T)-bipartition on $H_{V'}$ is the same as the cut before, we stop the pairwise refinement on these blocks.

6. Experimental Results

In this Secton we evaluate the performance of our flow-based refinement framework proposed in Section 4 and 5. We examine the impact of our sparsifying techniques of the *Lawler*-Network [28] on the performance of a maximum flow algorithm (see Section 6.3). Further, several configurations with different heuristics enabled or disabled are compared against the baseline configuration of *KaHyPar* in order to optimally configure our flow-based refinement algorithm (see Section 6.4 and 6.5). Finally, we compare our final configuration against other state-of-the-art hypergraph partitioner (see Section 6.6).

6.1. Instances

Our full benchmark set consists of 488 hypergraphs. We choose our benchmarks from three different research 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) [40]. Further, we interprete the Sparse Matrix instances of the *Florida Sparse Matrix collection* (Spm) [12] as hypergraphs using the row-net model [9]. 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 interprete the literals as vertices and each clause as a net (Literal) [32]. Mann and Papp [30] 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 induced nets containing all clauses where the corresponding variable occurs. A statiscal summary of the different instance types is presented in Table 7.

We divide our full benchmark set in two smaller subsets. Our *parameter tunning* 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 for each k 10 different *seeds* with $\epsilon = 3\%$.

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 against the state-of-the-art hypergraph partitioner *hMetis* [25, 26] and *PaToH* [9]. *hMetis* provides a direct k-way (hMetis-K) and recursive bisection (hMetis-R) implementation. Further, we also use the default configuration (PaToH-D) and quality preset (PaToH-Q) of *PaToH*. We configure *hMetis* to optimize the *sum-of-externel-degree*-metric (SOED) and calculate $(\lambda - 1)(\Pi) = \mathrm{SOED}(\Pi) - \mathrm{cut}(\Pi)$. This is also suggested by the authors of *hMetis* [26]. Further, 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 $0.55 \cdot c(V)$. In order 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 calculate the average (or minimum) of the different *seeds* of a hypergraph instance and than the *geometric mean* between all instances in order to give every instance comparable influence on the final result. In order to compare the performance of different hypergraph partitioner more detailed we use performance plots introduced in [37]. For each partitioner P and instance H we calculate the values $q_{H,P} := 1 - \text{best}_H/\text{algorithm}_{H,P}$ where best_H is the best quality achieved by a partitioner for instance H and $\text{algorithm}_{H,P}$ refers to the quality achieved by partitioner P for instance H. Afterwards, we sort all values $q_{H,P}$ of a partitioner P in decreasing order. For each partitioner P we plot the points $(H, q_{H,P})$. The faster the $q_{H,P}$ values intersect the zero line the better the performance of a partitioner in comparison to the others. If a partition of a partitioner P is not ϵ-balanced we set $q_{H,P} = 1 + \beta$ (with $\beta > 0$).

6.3. Flow Algorithms and Networks

In the first experiment we want examine the impact of our sparsifying techniques (see Section 4) on the performance of our maximum flow algorithms GoldbergTarjan and EdmondKarp. Therefore, we first take a look at the reduction of the number of nodes and edges on different benchmark types when using T_L (see Section 3.2), T_H (see Section 4.2), T_G (see Section 4.3) and T_{Hybrid} (see Section 4.4) . Further, we want to evaluate the performance of the two implemented maximum flow algorithms on these networks.
We evaluate the performance of the different flow networks on flow problems with size $|V'| \in \{500, 1000, 5000, 10000, 25000\}$ hypernodes. The instances are generated by executing *KaHyPar* on our benchmark subset (see Table 6) for $k = 2$ and five different seeds. After an instance is bipartitioned, we generate flow problem instances with the above mentioned sizes and execute each possible combination of flow algorithm and network on it.
The benchmark instances can be splitted into 6 different benchmark types. The properties of these instances in terms of the average hypernode degree and average hyperedge size is shown in Table 6. Remember, T_G should perform best on instances with a small average hyperedge size and T_H should perform best on instances with a small average hypernode degree. Based on Table 6, T_G should significantly reduce the number of nodes and edges on Primal and Literal instances and T_H on Dual instances in comparison to a our baseline T_L. Also both should sparsify the resulting flow network of Ispd98 and Dac instances. Further, we expect that T_{Hybrid} combines the advantages of both networks and performs best on all bechmark instances.
Figure 20 shows the predicted behaviour for flow problems of size 25000 hypernodes. T_{Hybrid} reduces the number of nodes of nearly every benchmark type by at least a factor of 2, except on Spm instances. Another observation is that instances with a large average hypernode degree, like Primal or Literal, yield to big flow problem instances and vice versa (see Dual instances).
In Figure 21 we compare the performance of our flow algorithms on different flow networks. The bars in the plot indicates speed ups relative to the flow algorithm EdmondKarp on flow network T_L. The main observation is that EdmondKarp performs better on small flow network instances and GoldbergTarjan on large flow network instances. For $|V'| \leq 1000$ EdmondKarp is faster than GoldbergTarjan in most of different bechmark types. For

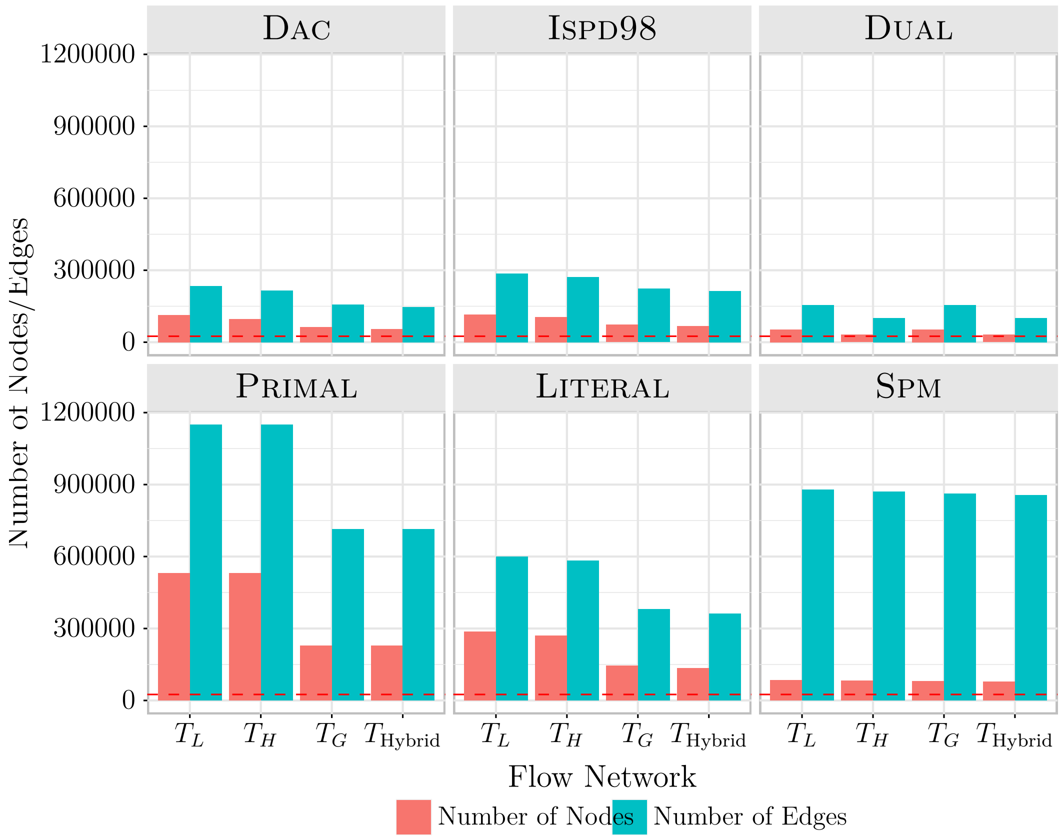

Figure 20: Comparison of the number of nodes and edges on our flow networks for flow problems of size $|V'| = 25000$ hypernodes on different benchmark types. The red dashed lines indicates 25000 nodes.

$|V'| > 1000$ we can observe the opposite behaviour except for Dac and Dual instances. But the resulting flow problems of these instances are still the smallest among all benchmark types (see Figure 20). On the largest flow network instances Primal and Literal for $|V'| = 25000$ GoldbergTarjan is up to a factor of 4-7 faster than EdmondKarp. Further, both algorithms perform best on T_{Hybrid}. Table 1 shows the summary of our flow algorithm and network experiment on all benchmark instances. This proofs our assumption that EdmondKarp works best on small instances and GoldbergTarjan on large instances. However, our *Max-Flow-Min-Cut* computations are embedded in a *Adapative Flow Iteration* strategy (see Section 3.3.2). Therefore, the running time of flow instances generated with a large α will dominate the ones with small α. Therefore, we choose GoldbergTarjan in combination with our flow network T_{Hybrid} in the following experiments.

6.4. Configuration of the k-way Flow-based Refiner

In this Section we examine the quality of our k-way flow-based refinement algorithm with different configurations on our parameter tuning benchmark subset (see Table 5). There are several configurations and tunning parameters which we have to evaluate:

- *Max-(F)low-Min-Cut* computations as refinement algorithm (see Section 5.3)
- *Adaptive Flow Iteration* parameter α' (see Section 3.3.2)
- *(C)ut Border Hyperedges* as sources and sinks (see Section 5.1)
- *(M)ost Balanced Minimum Cut* heuristic (see Section 5.2)

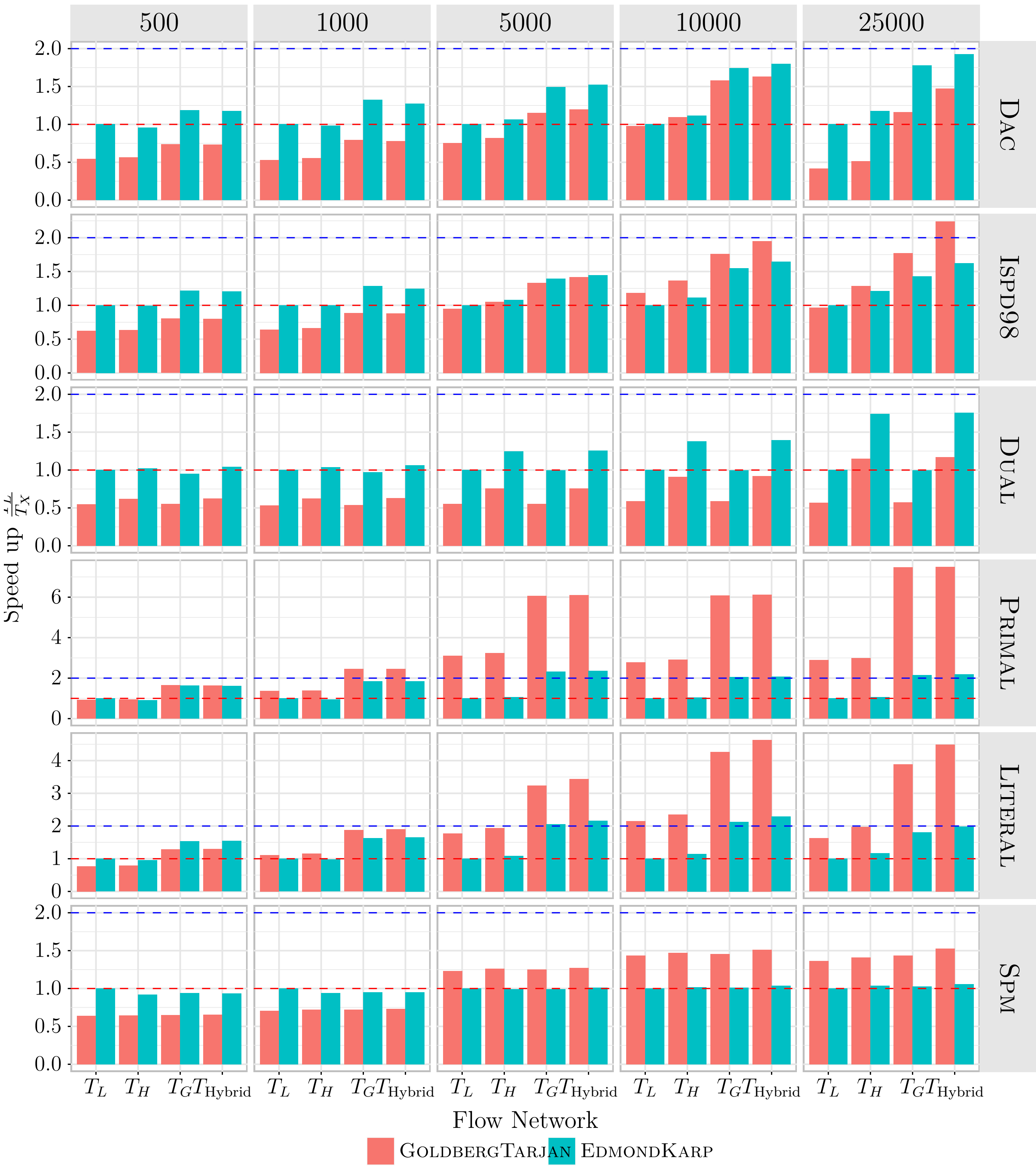

Figure 21: Speed up of our flow algorithms and networks relative to EdmondKarp on T_L for different instance sizes and types. The red dashed line indicates the (EdmondKarp, T_L) implementation and the blue dashed line indicates a speed up by a factor of 2.

Instance	$\lvert V'\rvert$	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	0.91	+2.24	+24.93	+29.35	**−25.39**	−24.3	−6.68	−11.53
	1000	1.95	+3.65	+26.19	+32.95	**−13.99**	−12.36	+10.81	+7.51
	5000	**13.71**	+8.63	+29.39	+43.11	+27.03	+35.33	+73.97	+86.31
	10000	**30.54**	+12.57	+36.15	+54.62	+47.93	+61.72	+100.41	+123.31
	25000	**67.96**	+23.36	+52.12	+87.8	+53.25	+77.85	+100.95	+138.8

Table 1: Running time comparison of maximum flow algorithms on different flow networks. Note, all values in the table are in percentage relative to GoldbergTarjan on flow network T_{Hybrid}. In each line the fastest variant is marked bold.

- Combining *Max-(F)low-Min-Cut* computations with *(FM)* refinement

In the following we will denote a configuration e.g. with (+F,-C,-M,-FM) which indicates which heuristic resp. technique is enabled (+) or disabled (−). The meaning of the abbreviations are explained in the enumeration above (see letters inside parenthesis). We evaluate a configuration for $k \in \{2, 4, 8, 16, 32, 64, 128\}$, $\alpha' \in \{1, 2, 4, 8, 16\}$ and 10 different seeds on our parameter tuning benchmark subset ($\epsilon = 3\%$). Our pairwise flow-based refinement is embedded in a k-way *Active Block Scheduling* refinement which is executed on each level i with $i = 2^j$ ($j \in \mathbb{N}_+$) (see Section 5.3). As reference we use the latest quality configuration of *KaHyPar* (KaHyPar-CA) [21].

The results are summarized in Table 2. The values in the column *Avg* are improvements of the connectivity metric relative to our baseline configuration (-F,-C,-M,+FM). The running time are absolute values in seconds. The first observation is that flows on its own as refinement strategy are not strong enough to outperform the *FM* heuristic. Our strongest configuration with $\alpha' = 16$ is 2.5% worse than our *FM* baseline. But the result is still remarkable, because we only execute flows on $\log n$ levels instead on n as the *FM* algorithm does. The running time scales nearly linear with parameter α'. Using our improved source and sink modelling approach with *Cut Border Hyperedges* (see Equation 5.5 and 5.6) significantly improves the solution quality especially for small α'. For small α most of the hypernodes are either a source or a sink. Introducing *Cut Border Hyperedges* reduces the number of hypernode sources and sinks by adding hyperedge sources and sinks. The quality improvement with this technique is therefore more effective for small α', because it significantly increase the possibilities of moving hypernodes between the blocks compared to the source and sink set modelling approach with Equation 5.3 and 5.4. The opposite effect can be observed, if we use the *Most Balanced Minimum Cut* heuristic without *Cut Border Hyperedges*. The quality improvement is more significant for large α'. The larger the flow problem, the larger is the number of different minimum (S, T)-cutsets and this increases the possibility to find a feasible solution according to our balanced constraint. If we combine both techniques, we obtain a configuration which significantly improves the solution quality for all α' compared to our baseline flow configuration. Also it outperforms our baseline *FM* configuration for $\alpha' = 16$ by 0.51%. If we enable *FM* refinement in all levels where no flow is executed, we improve the solution quality by nearly 2% (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. This has as consequence that a block becomes faster *inactive* during *Active Block Scheduling* and this decreases the number of rounds of complete pairwise flow-based refinements on the quotient graph.

Config.	(+F,-C,-M,-FM)		(+F,+C,-M,-FM)		(+F,-C,+M,-FM)	
α'	Avg.[%]	$t[s]$	Avg.[%]	$t[s]$	Avg.[%]	$t[s]$
1	−20.02	12.44	−15.48	12.94	−19.69	12.63
2	−14.61	15.16	−10.5	16.07	−14.17	15.77
4	−8.99	19.92	−5.98	21.22	−8.22	21.2
8	−4.96	28.71	−3.22	30.73	−3.37	31.25
16	−2.58	47.35	−1.52	50.89	−0.34	52.19
Ref.	(-F,-C,-M,+FM)		6373.88	13.73		

Config.	(+F,+C,+M,-FM)		(+F,+C,+M,+FM)	
α'	Avg.[%]	$t[s]$	Avg.[%]	$t[s]$
1	−15.26	13.29	0.14	14.99
2	−10.12	16.93	0.36	16.93
4	−5.08	23.01	0.67	20.76
8	−1.64	33.72	1.25	28.65
16	0.51	56.39	1.87	46.17
Ref.	(-F,-C,-M,+FM)		6373.88	13.73

Table 2: Table contains results for different configurations of our flow algorithm with increasing α'.

TODO 1: *evaluate effectiveness of flows*

6.5. Speed-Up Heuristics

At the end of Section 5.3 we present several heuristics to prevent unnecessary 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 on H. To prove that we execute KaHyPar-MF on our benchmark subset (see Table 6) and enable one heuristic after another. Table 3 summarizes the results of the experiment. KaHyPar-CA is the currently best configuration of *KaHyPar* and KaHyPar-MF is our baseline flow configuration of Section 6.4. The index of the remaining variants of KaHyPar-MF describes which speed-up heuristics are enabled (see Section 5.3). On average, enabling all speed up heuristics worsen the quality of KaHyPar-MF only by 0.09%. On the other hand the *Max-Flow-Min-Cut* computations are significantly faster by a factor of ≈ 2. In its final configuration KaHyPar-MF$_{(\mathrm{R1,R2,R3})}$ computes partitions with 2% better quality ($(\lambda-1)$-metric) than KaHyPar-CA by a slowdown only of a factor of ≤ 2. In the following we will denote our final configuration KaHyPar-MF$_{(\mathrm{R1,R2,R3})}$ with 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 9.

Variant	Avg.[%]	Min.[%]	$t_{\text{flow}}[s]$	$t[s]$
KaHyPar-CA	7077.2	6820.17	-	29.26
KaHyPar-MF	−2.13	−1.8	52.28	81.54
KaHyPar-MF$_{(R1)}$	−2.05	−1.74	41.48	70.74
KaHyPar-MF$_{(R1,R2)}$	−2.05	−1.73	35.27	64.54
KaHyPar-MF$_{(R1,R2,R3)}$	−2.04	−1.75	27.62	56.88

Table 3: Table shows results for our flow algorithm with different speed up heuristics.

Figure 22 summarizes the results of the experiment. KaHyPar-MF produced on ≈ 70% of all benchmark instances the best partition. It is followed by hMetis-R (14%), hMetis-K (11%), KaHyPar-CA (2.4%), PaToH-Q (1.9%) and PaToH-D (1.4%). Since KaHyPar-MF build on top of KaHyPar-CA it outperforms KaHyPar-CA on most of the instances. Comparing KaHyPar-MF individually with each partitioner, KaHyPar-MF produced better partitions than ...

- ...KaHyPar-CA in 96% of the instances.
- ...hMetis-R in 80% of the instances.
- ...hMetis-K in 82% of the instances.
- ...PaToH-Q in 95% of the instances.
- ...PaToH-D in 95% of the instances.

Especially on *VLSI* instances KaHyPar-MF calculates significantly better partitions than all other hypergraph partitioner (see Dac and Ispd98 in Figure 22).

Table 11 shows the running time of all partitioner on the 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	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 4: Comparing the average running time of KaHyPar-MF with KaHyPar-CA and other tools.

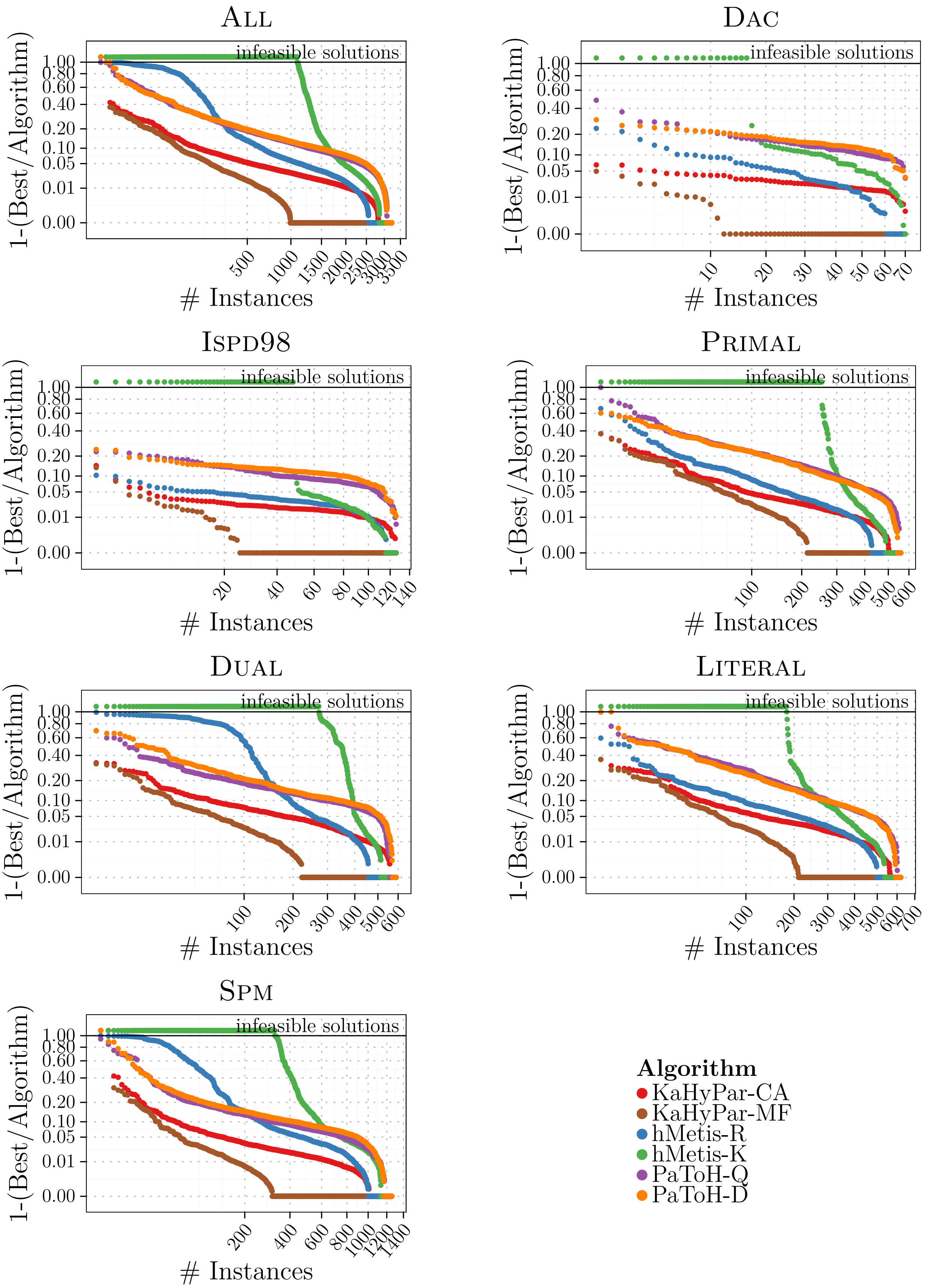

Figure 22: Min-Cut performance plots comparing KaHyPar-MF with KaHyPar-CA and other systems. The plots are explained in Section 6.2.

7. Conclusion

In this thesis, we developed a novel *local search* technique based on *Max-Flow-Min-Cut* computations for multilevel hypergraph partitioning. We integrated our *flow*-based refinement framework into the n-level hypergraph partitioner *KaHyPar* and show that in combination with the *FM* heurisitc our new approach produces the best-known partitions for a wide range of applications.

On the road to a practical implementation, we developed several concepts to speed up flow computations on a flow network of a hypergraph (see Section 3.2). One is to remove low-degree hypernodes from the network and instead insert a clique between all incident hyperedge nodes. We show that the number of nodes and edges could be reduced if the degree of a hypernode is smaller or equal than 3. Further, we model a hyperedge of size 2 as an 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 [28] on real-world benchmarks.

Our *flow*-based refinement framework is based on the ideas of Sanders and Schulz [36] (developed for multilevel graph partitioning). Given an already bipartitioned hypergraph, we show how to configure the source and sink sets of the flow network of a subhypergraph such that a *Max-Flow-Min-Cut* computation yields to a cut smaller or equal than the cut before on the original hypergraph. A main contribution is that we proof that applying the source and sink set modeling approach of Sanders and Schulz one-to-one on hypergraphs results in cuts greater or equal than with our optimized definition. This is because *border hyperdeges*, which induced sources and sinks, can be split into three different disjoint subsets on hypergraphs. This distinction enables a more efficient configuration of the sources and sinks. 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*. A *flow*-based refinement is executed in $\log n$ levels of the multilevel hierarchy between each adjacent block in the quotient graph. The pairwise block scheduling refinement is implemented in rounds and terminates if none of the blocks changes anymore. The sizes of the flow problems are chosen adaptively. If a *flow* computation on two blocks yields to an improvement the flow problem size is increased, otherwise it is decreased. Additionally, we try to automatically balance the partition after *Max-Flow-Min-Cut* computation by iterating over each minimum (S, T)-cutset. 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 of the original partition. Therefore, we implement several speed-up heuristics which prevents the execution of additional pairwise *flow* refinements.

Our new quality configuration KaHyPar-MF produced on 95% of our benchmark instances better partitions than our old baseline configuration KaHyPar-CA. On average the solution quality is 2% better and only within a factor of 2 slower. In comparison with other state-of-the-art hypergraph partitioners, KaHyPar-MF produced on 70% of the benchmark instances the best-known partitions with a running time comparable to the direct k-way implementation of *hMetis*.

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 Hypergraph Partitioner*

which runs in reasonable time. The quality mainly depends on the number of flow executions through the multilevel hierarchy. The number of flow executions 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 [28] 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 [34] 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. Adding a new sparsifying technique would require with great certainty a reimplementation of the flow network.

In Section 5.3 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 $v_2 \in e$ and there exists one hyperedge e' where $v_2 \in e'$ and $v_1 \notin e'$. All hypernodes with $d(v) \leq 3$ are removed 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 induced $2d(v_2) = 8$ edges. Alternatively, we could 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 induced 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 their 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] T. N. Bui and C. Jones. Finding Good Approximate Vertex and Edge Partitions is NP-Hard. *Information Processing Letters*, 42(3):153–159, 1992.

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

[9] 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.

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

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

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

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

[14] 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.

[15] 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.

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

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

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

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

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

[27] 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.

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

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

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

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

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

[33] 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.

[34] 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.

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

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

[37] 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.

[38] 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.

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

[40] 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.

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

[42] 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 5: 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 6: 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 7: 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 9: Instances excluded from the full benchmark set evaluation.

B. Detailed Flow Network and Algorithm Evaluation

Instance	$\lvert V' \rvert$	GoldbergTarjan T_{Hybrid} $t[ms]$	T_G $t[\%]$	T_H $t[\%]$	T_L $t[\%]$	EdmondKarp T_{Hybrid} $t[\%]$	T_G $t[\%]$	T_H $t[\%]$	T_L $t[\%]$
All	500	0.91	+2.24	+24.93	+29.35	**−25.39**	−24.3	−6.68	−11.53
	1000	1.95	+3.65	+26.19	+32.95	**−13.99**	−12.36	+10.81	+7.51
	5000	**13.71**	+8.63	+29.39	+43.11	+27.03	+35.33	+73.97	+86.31
	10000	**30.54**	+12.57	+36.15	+54.62	+47.93	+61.72	+100.41	+123.31
	25000	**67.96**	+23.36	+52.12	+87.8	+53.25	+77.85	+100.95	+138.8
Dac	500	0.34	−0.36	+30.14	+34.98	−37.61	**−38.08**	−23.12	−26.56
	1000	0.8	−1.7	+41.18	+47.43	−38.94	**−41.19**	−20.88	−22.17
	5000	5.2	+4.11	+46.02	+58.5	**−21.35**	−19.79	+12.55	+19.6
	10000	10.67	+3.2	+48.92	+66.83	**−9.41**	−6.44	+46.23	+63
	25000	31.43	+26.81	+186.2	+255.32	**−23.53**	−17.16	+25.16	+47.29
Ispd98	500	0.48	−0.58	+26.23	+28.54	−33.85	**−34.5**	−19.55	−20.14
	1000	1.11	−0.8	+32.35	+37.47	−29.32	**−31.59**	−11.91	−11.88
	5000	7.06	+6.65	+35.1	+49.35	**−1.67**	+1.64	+31.03	+41.91
	10000	**16.33**	+10.97	+42.54	+64.68	+18.38	+25.84	+75.19	+95.09
	25000	**75.01**	+26.26	+73.85	+132.06	+37.85	+56.79	+85.28	+124.01
Dual	500	0.3	+12.37	+0.99	+13.6	**−40.36**	−34.35	−39.13	−37.67
	1000	0.6	+16.87	+0.83	+18.38	**−40.93**	−35.35	−39.47	−37.18
	5000	3.2	+37.54	+0.21	+37.78	**−39.66**	−23.77	−39.17	−24.01
	10000	5.78	+55.72	+1.21	+55.86	**−34.01**	−7.81	−33.3	−8
	25000	14.71	+105.19	+2.15	+105.88	**−33.35**	+17.43	−32.59	+17.28
Primal	500	1.85	**−0.73**	+73.92	+76.03	+0.86	+0.17	+79.92	+63.57
	1000	**3.9**	+0.15	+77.48	+81.23	+33.02	+33.57	+160.43	+145.98
	5000	**29.8**	+0.84	+88.23	+96.71	+160	+162.28	+481.91	+510.71
	10000	**45.94**	+0.69	+109.75	+120.04	+195.68	+197.69	+487.6	+511.93
	25000	**174.32**	+0.21	+151.07	+159.04	+243.77	+248.81	+609.44	+648.46
Literal	500	0.86	+0.72	+63.65	+67.45	**−16.1**	−15.41	+35.63	+29.41
	1000	**1.92**	+1.64	+64.51	+71.46	+15.13	+17.07	+95.07	+90.72
	5000	**12.31**	+6.15	+76.65	+94.2	+59.04	+66.99	+216.7	+243.13
	10000	**29.75**	+8.55	+97.28	+115.37	+102.47	+117.45	+302.93	+363.17
	25000	**64.4**	+15.75	+128.34	+175.78	+126.59	+148.78	+286.31	+349.43
Spm	500	1.46	+0.35	+1.22	+2.47	−29.92	−30.42	−28.84	**−34.57**
	1000	3.09	+1.45	+1.14	+3.28	−23.32	−22.94	−22.17	**−26.89**
	5000	**25.81**	+1.79	+1.09	+3.26	+26.02	+28.55	+28.61	+27.43
	10000	**74.81**	+3.78	+2.48	+5.38	+45.86	+49.36	+48.77	+51.06
	25000	**107.6**	+6.67	+8.56	+12.07	+44.39	+48.88	+47.68	+52.96

Table 10: 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.

C. Detailed Speed Up Heuristic Evaluation

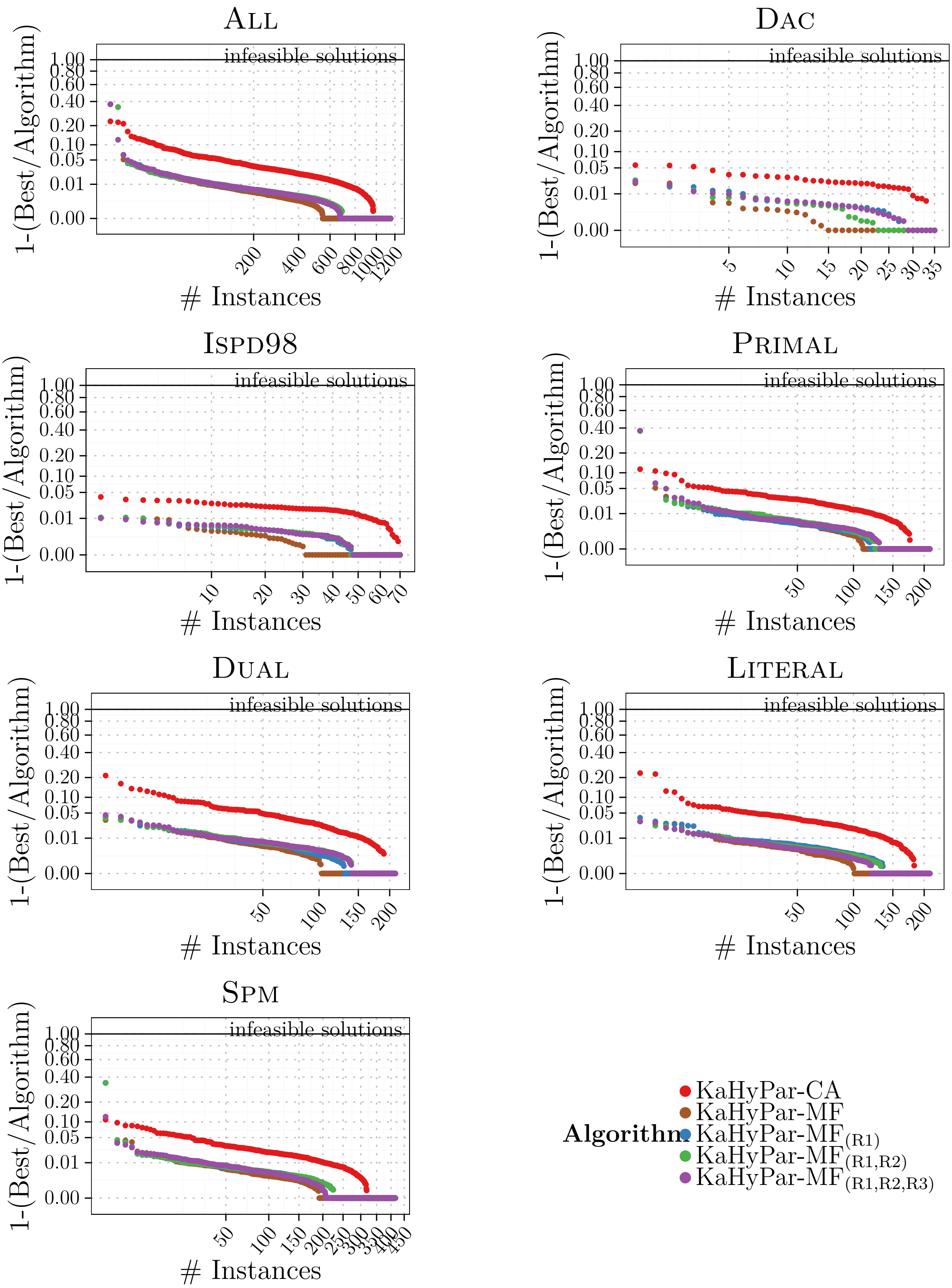

Figure 23: Min-Cut performance plots comparing KaHyPar-MF with KaHyPar-CA. The plots are explained in Section 6.2.

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	81.54	699.18	75.97	114.67	185.56	143.93	28.74
KaHyPar-MF$_{(R1)}$	70.74	600.87	59.69	94.9	150.56	128.67	26.47
KaHyPar-MF$_{(R1,R2)}$	64.54	573.41	50.28	88.11	134.84	113.59	24.8
KaHyPar-MF$_{(R1,R2,R3)}$	56.88	526.86	43.32	74.76	116.79	101.76	22.31

Table 11: Comparing the average running time of KaHyPar-MF with KaHyPar-CA.

D. Detailed Comparison with other Systems

Partitioner	Average $\lambda - 1$						
	All	Dac	Ispd98	Primal	Literal	Dual	Spm
KaHyPar-MF	7819.11	17590.1	5671.37	15923.74	15844.61	3061.94	6165.74
KaHyPar-CA	2.03	2.47	1.72	1.69	2.25	2.71	1.75
hMetis-R	15.21	2.99	1.14	1.69	2.31	42.33	19.22
hMetis-K	14.71	7.78	0.9	3.66	8.77	27.66	19.09
PaToH-Q	8.98	12.86	7.41	11.72	12.81	7.96	6.37
PaToH-D	16.21	22.98	14.54	17.83	20.97	17.4	12.5

Table 12: 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	1064.06	3147.96	6062.8	9406	14756.03	21978.89	31820.94
KaHyPar-CA	1.73	2.06	2.36	2.28	2.11	1.9	1.73
hMetis-R	26.46	18.26	16.34	15.25	12.33	10.23	8.08
hMetis-K	26.86	17.19	15.18	15.06	11.29	9.83	8.1
PaToH-Q	11.1	8.5	8.57	9.49	8.87	8.6	7.7
PaToH-D	14.62	15.94	18.55	19.34	15.62	15.31	14.09

Table 13: 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.

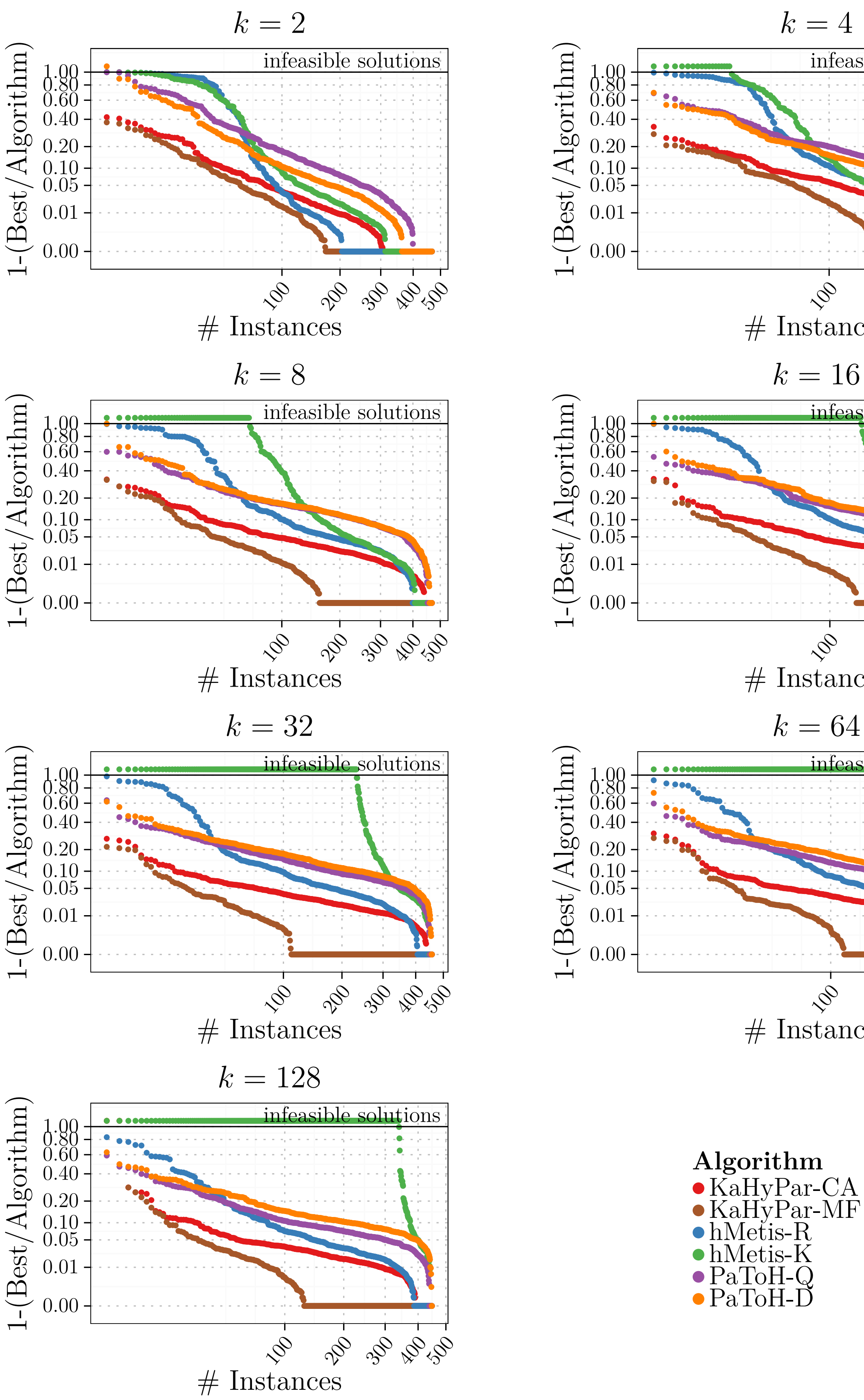

Figure 24: Min-Cut performance plots comparing KaHyPar-MF with KaHyPar-CA and other systems for different values of k.

Partitioner	Running Time $t[s]$						
	$k = 2$	$k = 4$	$k = 8$	$k = 16$	$k = 32$	$k = 64$	$k = 128$
KaHyPar-MF	22.13	38.51	55.04	67.83	85.75	108.97	128.04
KaHyPar-CA	12.68	17.16	23.88	31.01	41.69	57.35	76.61
hMetis-R	27.87	51.59	74.74	91.09	109.13	128.66	149.34
hMetis-K	25.47	32.27	42.5	53.41	74	109.12	152.92
PaToH-Q	1.93	3.61	5.44	7.01	8.4	10.06	11.44
PaToH-D	0.43	0.77	1.12	1.42	1.71	2.02	2.29

Table 14: Comparing the average running time of KaHyPar-MF with KaHyPar-CA and other systems for different values of k.