Engineering Algorithms for Hypergraph Minimum Cut

Loris Wilwert

October 15, 2025

4165279

Master Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:
Univ.-Prof. PD. Dr. rer. nat. Christian Schulz

Co-Supervisor:
Prof. Dr. Bora Uçar

Co-Supervisor:
M. Sc. Adil Chhabra

Acknowledgments

Firstly, I would like to thank Prof. Dr. Christian Schulz for introducing me to a variety of fascinating topics within the field of algorithm engineering, and for giving me the great opportunity to contribute to his area of research. It was a pleasure not only learning from him but also working with him over the past couple of years. The same goes for Prof. Dr. Bora Uçar, whose passionate and inventive nature never fails to offer interesting new perspectives. I would also like to thank Adil Chhabra, who was a great help and support in every situation and with whom I had many insightful discussions.

I would like to express my deepest gratitude to my parents, who have dedicated parts of their lives to helping me become the person that I am today. In return, I would like to dedicate this work to them, as it would not have been possible without their constant care and love. Likewise, I cannot thank my sister enough, who has always been a source of joy throughout my entire life and who, together with her partner, supports me in all my decisions. Finally, I am grateful for the help of all my friends in Luxembourg, Heidelberg and elsewhere who have been on my side in the various stages of my life up to this point.

Hiermit versichere ich, dass ich die Arbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und wörtlich oder inhaltlich aus fremden Werken Übernommenes als fremd kenntlich gemacht habe. Ferner versichere ich, dass die übermittelte elektronische Version in Inhalt und Wortlaut mit der gedruckten Version meiner Arbeit vollständig übereinstimmt. Ich bin einverstanden, dass diese elektronische Fassung universitätsintern anhand einer Plagiatssoftware auf Plagiate überprüft wird.

Heidelberg, October 15, 2025

Loris Wilwert

Abstract

A central task of many applications such as network reliability, community detection or VLSI design is to solve the hypergraph minimum cut problem, which aims to separate the vertices of a hypergraph into two non-empty partitions while minimizing the total weight of hyperedges crossing the cut. Although many advances were made for graphs in recent years, extending minimum cut solvers to hypergraphs remains challenging. For this reason, we introduce several novel algorithms, starting with a Binary Integer Program (BIP) and a Mixed-Integer Linear Program (MILP) as two simple baselines. We then present HEICUT, a scalable and fast algorithm for finding a minimum cut in hypergraphs with up to hundreds of millions of hyperedges. HEICUT applies seven provably exact reduction rules that reduce the size of the hypergraph while still allowing the retrieval of the optimal solution. These reduction rules consist of novel hypergraph-specific rules as well as generalizations of already proven rules from graph algorithms. Optionally, HEICUT can apply a heuristic reduction based on label propagation to shrink complex structures. The reduced instance is then solved by an exact ordering-based algorithm, for which we propose novel recurrent rules that search for multiple contractions per iteration. We also outline how the different parts of HEICUT can be parallelized, including the first parallel implementation of the ordering-based solver, to the best of our knowledge. Our extensive evaluation on more than 500 real-world hypergraphs shows that combining the exact reduction rules with early stopping already identifies a minimum cut in over 95% of the instances. HEICUT solves twice as many instances as the next best state-of-the-art algorithm, while being three to four orders of magnitude faster. Based on these results, we use some of the techniques of HEICUT to extend and improve the hypercactus algorithm of Chekuri and Xu, which aims to find all minimum cuts in a hypergraph. We present the first ever implementation of the hypercactus algorithm, to the best of our knowledge. In the experiments, we manage to find all minimum cuts in more hypergraphs than a single minimum cut is found by the current state-of-the-art algorithms.

Contents

CHAPTER 1

Introduction

1.1 Motivation

In today's increasingly complex world, interactions between more than two entities lie at the heart of many phenomena that arise in areas such as communication networks, circuit design, database systems and computational biology. In order to facilitate the analysis, these systems are typically modeled as so-called hypergraphs, which use hyperedges to capture the higher-order relationships between vertices. Unlike the edges of traditional graphs, which model only pairwise dependencies, hyperedges can connect any number of vertices and represent multi-way interactions. This property makes hypergraphs ideal for representing group-level dependencies [9].

In this thesis, we tackle the hypergraph minimum cut problem, which aims to separate the vertices of a hypergraph into two non-empty partitions such that the sum of the weights of the hyperedges running between the partitions is minimized. This problem, much like its restricted variant for graphs, is of fundamental algorithmic importance and has been studied for multiple decades. In particular, solving this problem efficiently can boost the performance of several other hypergraph algorithms that rely on minimum cuts, covering fields such as cybersecurity [78], hypergraph expansion [65] and quantum computing [55]. In addition, a well-optimized solver can be of great benefit for applications that currently rely on graph-based alternatives, including those in clustering [36], network reliability [45], community detection [11] and VLSI design [3]. In network reliability for example, a minimum cut is most likely to disconnect the network when all hyperedges fail with equal probability. In community detection, the absence of a minimum cut within a region may indicate the presence of a community. In VLSI design, reducing the number of interconnections between logical blocks is key to reducing the cost of the circuit. Furthermore, a natural extension of the hypergraph minimum cut problem is to search for *all* minimum cuts rather than just one. This is achieved by constructing a so-called hypercactus, i.e., a compact representation that preserves all minimum cuts of the hypergraph [14, 15]. Im-

plementing a fast algorithm to find such a hypercactus representation would pave the way for a multitude of hypergraph algorithms in related fields, such as incremental minimum cuts [34], enumerating all minimum cuts [7] and connectivity augmentation [17, 25].

Although the minimum cut problem can be solved efficiently on graphs, leading to sophisticated algorithms [46, 59, 60, 62, 72] as well as scalable solvers such as VIECUT [41], the hypergraph setting remains more challenging. More specifically, most state-of-the-art hypergraph algorithms are rigorous theoretical extensions of graph algorithms but they fail to achieve satisfactory execution times in practice. To address this issue, we present a series of algorithms that efficiently solve the hypergraph minimum cut problem. In addition, we propose a number of improvements and extensions that will greatly help to advance the research in this area.

1.2 Our Contribution

We present the key contributions of our research, highlighting novel insights and concepts. Our work addresses existing gaps in the literature and improves upon previous techniques. In particular, we make five key contributions to the hypergraph minimum cut problem:

- We propose several efficient algorithms for finding a minimum cut in a hypergraph. First, we formulate a Binary Integer Program (BIP) as well as a Mixed-Integer Linear Program (MILP), which serve as simple baselines and already match the performance of current state-of-the-art algorithms on most instances. As our main contribution, we present HEICUT, a lightweight and scalable algorithm that quickly finds a minimum cut in unweighted or weighted hypergraphs with up to hundreds of millions of hyperedges. HEICUT applies a series of provably exact reduction rules to reduce the input hypergraph to a smaller kernel without losing information about the optimal solution. These exact reduction rules consist of novel hypergraph-specific rules as well as generalizations of already proven rules from graph algorithms. HEICUT also supports the option of performing a heuristic reduction by identifying and contracting a clustering via label propagation. After shrinking the hypergraph, HEICUT applies an existing ordering-based algorithm to find an exact minimum cut in the reduced hypergraph. In our evaluation, HEICUT solves twice as many instances as the next best state-of-the-art competitor, while being three to four orders of magnitude faster. This is because 85% of all instances can be maximally reduced by the exact reduction rules, eliminating the need to run the underlying ordering-based solver. Combined with further stopping criteria, HEICUT can already solve 95% of all instances with the exact reduction rules. Overall, HEICUT achieves near-linear running time on most instances in practice, even if its asymptotic time complexity is non-linear.

- We extend the well-known ordering-based algorithm for identifying a minimum cut in a hypergraph, which was proposed in three main variants [52, 56, 66]. For each variant, we define novel recurrent contraction rules that allow to search for multiple contractions per iteration, improving upon the standard single-contraction approach. When using the ordering-based solver as a standalone algorithm with the ordering of Mak and Wong [56], searching for multiple contractions per iteration leads to faster results on 70% of the instances. In HEICUT, the multiple contractions technique speeds up every single instance that is passed to the ordering-based solver. Besides, our recurrent rules can be applied to any other algorithm that uses the ordering-based solver in a subroutine, directly improving its performance in practice.

- We provide a detailed description of how HEICUT can be parallelized. For this, we show that the exact reduction rules and the label propagation heuristic are embarrassingly parallel. In addition, we present the first ever parallel implementation of the ordering-based solver, to the best of our knowledge. In particular, we use persistent threads with barriers to perform thread-specific contractions in each iteration. This new parallel approach benefits any algorithm that uses the ordering-based solver in a subroutine. The greatest speedup is obtained when combining the parallel execution with our novel multiple contractions technique, meaning that all threads search independently for multiple contractions in each iteration.

- To the best of our knowledge, we perform the first extensive comparison between the state-of-the-art algorithms for the hypergraph minimum cut problem. For this, we run a series of experiments on more than 500 real-world and synthetic hypergraphs [1]. To ensure maximum transparency, the source code of all implemented algorithms is publicly available in a repository [2]. We also introduce a new dataset of $(k, 2)$-core instances, where the minimum cut value is guaranteed to be different from the smallest weighted vertex degree. These instances are harder to solve as they do not admit a trivial minimum cut and provide a benchmark for future research.

- We describe how a weaker version of our exact reduction rules can be applied to the hypercactus algorithm of Chekuri and Xu [14, 15], which aims to find all minimum cuts in a hypergraph. Together with several practical improvements to their split oracle, we manage to propose the first ever implementation of their hypercactus algorithm, to the best of our knowledge. The experiments show that our implementation highly benefits from the proposed kernelization. In particular, the hypercactus representation can be found within less than a second for 67.41% of the unweighted and all but one weighted medium-sized instances. This means that our proposed hypercactus algorithm finds all minimum cuts in more medium-sized hypergraphs than a single minimum cut is found by the current state-of-the-art algorithms.

[1] https://doi.org/10.5281/zenodo.17142170
[2] https://github.com/HeiCut/HeiCut

1.3 Structure

The remainder of this thesis is organized as follows. We start by introducing the general definitions and notation used throughout this thesis in Chapter 2. This includes the problem definition as well as descriptions of the concepts of kernelization and linear programming. We then outline previous work on the minimum cut problem in Chapter 3. In particular, we describe how the research landscape on graphs has evolved over the years, before explaining how this work has been extended and optimized for the hypergraph scenario. At the end of the chapter, we mention recent advances in parallelization and the representation of all minimum cuts. Afterwards, we propose several algorithms to solve the hypergraph minimum cut problem efficiently in Chapter 4, starting with the definition of BIP and MILP formulations that serve as a simple baseline. We then present HEICUT, a lightweight and scalable algorithm that finds a minimum cut in unweighted or weighted hypergraphs of arbitrary size. First, we explain the kernelization phase of HEICUT, consisting of provably exact reduction rules and an optional heuristic reduction that uses clustering contraction via label propagation. Next, we describe how HEICUT implements an underlying ordering-based algorithm to solve the kernel and how this can be improved by searching for multiple contractions. Finally, we outline how the different parts of HEICUT can be parallelized. Chapter 5 describes how the techniques from HEICUT can be used to extend and improve the hypercactus algorithm of Chekuri and Xu [14, 15], which aims to find all minimum cuts in a hypergraph. In addition, we provide a series of improvements that allow us to propose the first ever implementation of the hypercactus algorithm of Chekuri and Xu, to the best of our knowledge. In Chapter 6, we perform extensive experiments on more than 500 real-world and synthetic hypergraphs to compare our proposed algorithms to the current state-of-the-art competitors. Chapter 7 contains the conclusion from our experiments as well as an outlook to possible approaches and improvements in future work.

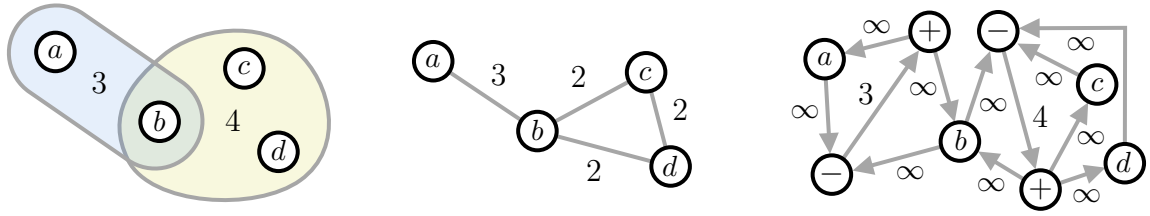
CHAPTER 2

Fundamentals

This chapter first introduces the general definitions and notation used throughout this thesis. Afterwards, we define the hypergraph minimum cut problem together with its submodular properties, followed by an overview of the hypercactus representation and related problems. Finally, we introduce the notion of kernelization and linear programming.

2.1 General Definitions

Let $H = (V, E)$ be an *undirected* hypergraph with V being the set of *vertices* (or *nodes*) and E being the multiset of *hyperedges* (or *nets*). Note that E is a multiset over $\mathcal{P}(V) \setminus \emptyset$ where $\mathcal{P}(V) := \{V' : V' \subseteq V\}$ represents the *power set* of V, i.e., each hyperedge consists of a (non-empty) subset of vertices. Each vertex can only be contained in a hyperedge once, but multiple hyperedges can contain the same set of vertices. The vertices that compose a hyperedge $e \in E$ are called the *pins* of e. The *size* of a hyperedge $e \in E$ is defined as the number of pins in e and denoted by $|e|$. We define $n := |V|$ as the number of vertices, $m := |E|$ as the number of hyperedges and $p := \sum_{e \in E} |e|$ as the total number of pins in the hypergraph H. Let $c : V \to \mathbb{R}_{\geq 0}$ denote a vertex-weight function and let $\omega : E \to \mathbb{R}_{\geq 0}$ denote a hyperedge-weight function. We generalize the functions c and ω to sets, such that $c(V') := \sum_{v \in V'} c(v)$ for $V' \subseteq V$ and $w(E') := \sum_{e \in E'} \omega(e)$ for $E' \subseteq E$. For *unweighted* hypergraphs, we assume unit hyperedge weights, i.e., for all $e \in E$ we have $\omega(e) = 1$. A hyperedge $e \in E$ is *incident* to a vertex $v \in V$ if $v \in e$. Two vertices u, v are *adjacent* if at least one hyperedge is incident to both of them. For a vertex $v \in V$, we define $I(v) := \{e \in E : v \in e\}$ as the set of incident hyperedges to the vertex v and generalize it to sets, such that $I(V') := \bigcup_{v \in V'} I(v)$. The *neighborhood* of the vertex v is defined as $N(v) := \{u \in V \setminus \{v\} : I(u) \cap I(v) \neq \emptyset\}$. Let $d(v) := |I(v)|$ be the *degree* of v and $d_\omega(v) := \omega(I(v))$ be the *weighted degree* of v. We denote $\delta := \min_{v \in V}\{d(v)\}$ as the minimum (unweighted) degree and $\Delta := \max_{v \in V}\{d(v)\}$ as the maximum (unweighted) degree of H. The weighted counterparts are δ_ω and Δ_ω. A *connected path* P between two vertices u and v in H is a sequence of s pairwise distinct hyperedges $(e_1, \ldots, e_s)$



(i): Hypergraph H **(ii):** Clique-expanded graph $\mathring{G}_H$ **(iii):** Digraph $\vec{G}_H$

Figure 2.1: The clique-expanded graph $\mathring{G}_H$ and digraph $\vec{G}_H$ of an example hypergraph H.

with $u \in e_1 \wedge v \in e_s$ such that each consecutive pair of hyperedges shares at least one vertex, i.e., $e_i \cap e_{i+1} \neq \emptyset$ for all $1 \leq i < s$. The *vertex-induced sub-hypergraph* of H with respect to $V' \subseteq V$ is defined as $H[V'] := (V', \{e : e \subseteq V' \wedge e \in E\})$.

A *contraction* of two vertices $u, v \in V$ in a hypergraph H consists of creating a new vertex w such that $c(w) = c(u) + c(v)$ and $I(w) = I(u) \cup I(v)$, after which u and v are removed from the vertex set. We denote $H \,/\, \{u, v\}$ as the hypergraph obtained from H after contracting the vertices u and v. Note that hyperedges incident to both vertices u and v end up with one pin less after the contraction, i.e., their size is reduced by one. In the case where merging the two vertices leads to single-pin hyperedges, they are removed. Similarly, if the contraction produces *parallel* hyperedges, i.e., hyperedges with the same set of pins, they are unified to a new hyperedge whose weight is the sum of the weights of these parallel hyperedges. The concept of contraction is generalized to sets $V' \subseteq V$, such that $c(w) = \sum_{v \in V'} c(v)$ and $I(w) = \bigcup_{v \in V'} I(v)$. The contracted hypergraph is $H \,/\, V'$.

We call a hypergraph d*-uniform* if each hyperedge contains exactly d vertices. A *graph* is defined as a 2-uniform hypergraph, i.e., it is a special hypergraph where the size of all hyperedges is exactly two. Note that for a graph, we speak of *edges* rather than hyperedges and we denote the graph by G rather than H. A graph is *directed* if its edges are ordered pairs instead of 2-element sets, i.e., for all $u, v \in V$ the directed edges (u, v) and (v, u) are not the same. Since some problems are easier to solve on graphs than on hypergraphs, it can be useful to transform a hypergraph into an appropriate graph representation. We will briefly outline two of such graph transformations, which are visualized in Figure 2.1.

Clique-expanded graph. We obtain the *clique-expanded graph* $\mathring{G}_H = (\mathring{V}, \mathring{E})$ from a hypergraph $H = (V, E)$ by setting $\mathring{V} := V$ and replacing each hyperedge $e \in E$ from H by a *graph clique* between the pins of e. In a graph clique, each pair of vertices is connected by an edge, i.e., $\mathring{E} := \bigcup_{e \in E} \{\{u, v\} : u, v \in e \wedge u \neq v\}$. Each edge in the graph clique that replaces e is undirected and obtains the weight $\frac{\omega(e)}{|e|-1}$.

Digraph. The *digraph* $\vec{G}_H = (\vec{V}, \vec{E})$ of a hypergraph $H = (V, E)$ is a directed graph where $\vec{V} := V \cup E^+ \cup E^-$ with $E^+ := \{e^+ : e \in E\}$ and $E^- := \{e^- : e \in E\}$. The edge set $\vec{E}$ is constructed by adding for every hyperedge $e \in E$ the directed edge (e^-, e^+) with weight $c(e)$. Besides, for every $e \in E$ and $v \in e$, we add the two directed edges (v, e^-) and (e^+, v) with infinite weights. The original name of the digraph is *Lawler network* [54].

2.2 Problem Definition

We define a *cut* (V_1, V_2) in a hypergraph H as a *bipartition* of the vertex set V, i.e., it divides V into two disjoint non-empty *partitions* $V_1 \cup V_2 = V$ with $V_1 \cap V_2 = \emptyset$. Since $V_2 = V \setminus V_1$, the cut can be fully represented by the set V_1, which simplifies the notation. The *value* (or *capacity*) of the cut V_1 is denoted by $\lambda[V_1]$ and it is defined as the sum of the weights of all hyperedges running between the partitions V_1 and $V \setminus V_1$:

$$\lambda[V_1] := \sum_{e \in E \,\wedge\, e \cap V_1 \neq \emptyset \,\wedge\, e \cap V \setminus V_1 \neq \emptyset} \omega(e) \tag{2.1}$$

The hypergraph minimum cut problem consists of finding a *minimum cut*, i.e., a valid cut in the hypergraph H whose value is minimal. Note that the solution to this problem is not necessarily unique, as there may exist multiple minimum cuts in H. The problem is often simplified to finding only the value of the minimum cut(s), since the partitions of a specific minimum cut can often be extracted with minor adjustments. We denote $\lambda(H)$ (or just λ if the context is clear) as the *minimum cut value* of the hypergraph H and define it as:

$$\lambda(H) := \min_{\emptyset \neq A \subsetneq V} \{\lambda[A]\} \tag{2.2}$$

For some algorithms, $\hat{\lambda}(H)$ (or just $\hat{\lambda}$) is the smallest upper bound of the minimum cut value $\lambda(H)$ found so far. A cut is *trivial* if one of its partitions contains exactly one vertex. The *minimum trivial cut* is the trivial cut that isolates the vertex $v \in V$ with the smallest weighted vertex degree, i.e., $d_\omega(v) = \delta_\omega$. Thus, the minimum weighted vertex degree δ_ω can be used to initialize the upper bound $\hat{\lambda}(H)$. A *split* is defined as a non-trivial minimum cut. For two vertices $s, t \in V$, we define the *minimum s-t cut value* as the lowest value of all cuts separating the vertices s and t and refer to it as $\lambda(H, s, t)$. The notion of trivial cuts and splits is extended naturally to s-t cuts. The (global) minimum cut value and the minimum s-t cut value for any $s, t \in V$ can be found in polynomial time [54].

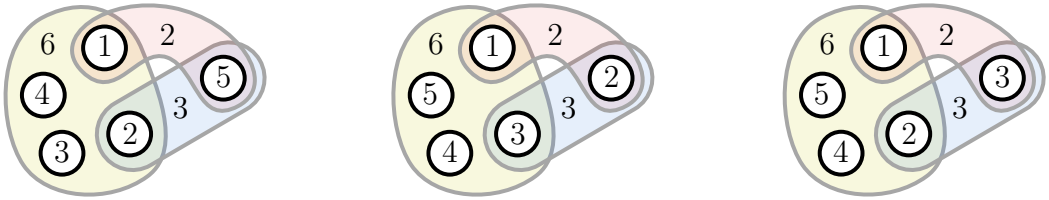
2.2.1 Submodularity

The function in Equation 2.1 defines a symmetric submodular function over the ground set V [66]. More specifically, the following equations hold for all $A, B \subseteq V$:

$$\lambda[A] = \lambda[V \setminus A] \tag{2.3}$$

$$\lambda[A \cup B] + \lambda[A \cap B] \leq \lambda[A] + \lambda[B] \tag{2.4}$$

Since any symmetric submodular function admits a *cut-equivalent tree* [31, 29], it can be shown that there always exists a so-called *pendent pair* for the hypergraph H [66], i.e., an ordered pair of vertices (s, t) such that the trivial cut $\{t\}$ is a minimum s-t cut. Finding a pendent pair is straightforward, as one can apply a so-called α*-ordering* on the



(i): MA ordering ($\alpha = 1$) **(ii):** Tight ordering ($\alpha = 0$) **(iii):** Queyranne ordering ($\alpha = \frac{1}{2}$)

Figure 2.2: The three special cases of the α-ordering on the same hypergraph H. Although the starting vertex is the same for all of them, each one leads to a different vertex ordering.

vertices $(v_1, \ldots, v_n)$, so that the last two vertices form a pendent pair (v_{n-1}, v_n) [15, 66]. More formally, an α-ordering of the vertices must satisfy the following inequality for all $1 < i < j \leq n$ where $V_{i-1} := (v_1, \ldots, v_{i-1})$ is the respective partial α-ordering:

$$\alpha d_\omega(\{v_i\}, V_{i-1}) + (1-\alpha) d'_\omega(\{v_i\}, V_{i-1}) \geq \alpha d_\omega(\{v_j\}, V_{i-1}) + (1-\alpha) d'_\omega(\{v_j\}, V_{i-1}) \quad (2.5)$$

For each $v \in V$, the scalar $\alpha \in [0, 1]$ is used to interpolate between the *adjacency* contribution $d_\omega(\{v\}, V_{i-1})$ and the *containment* contribution $d'_\omega(\{v\}, V_{i-1})$. More generally, for two disjoint sets $A, B \subseteq V$, the adjacency contribution $d_\omega(A, B)$ is defined as the total weight of the hyperedges incident to at least one vertex of both A and B. The containment contribution $d'_\omega(A, B)$ has the additional constraint that the hyperedges must be fully contained within $A \cup B$:

$$d_\omega(A, B) := \sum_{e \in I(A) \cap I(B)} \omega(e) \quad (2.6)$$

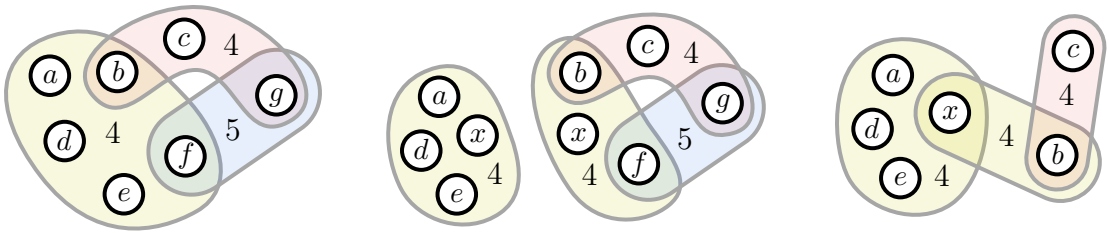
$$d'_\omega(A, B) := \sum_{\substack{e \in I(A) \cap I(B) \\ \wedge e \subseteq A \cup B}} \omega(e) \quad (2.7)$$

An α-ordering can be computed in $O(p + n \log(n))$ time for weighted and in $O(p)$ time for unweighted hypergraphs when using a Fibonacci heap. There exist three special cases of an α-ordering that are most commonly used. Figure 2.2 provides an example for each of these three special cases. One advantage of these three cases is that, when the hyperedge weights are restricted to integer values, the interpolation also yields an integer value for each vertex, either directly or with minor modifications:

Maximum adjacency (MA) ordering ($\alpha = 1$) [52]. Selects the vertices based only on adjacency, i.e., it prioritizes vertices with high connectivity to the already-ordered set.

Tight ordering ($\alpha = 0$) [56]. Selects the vertices based only on containment, i.e., it considers only hyperedges that are contained within the growing set.

Queyranne ordering ($\alpha = \frac{1}{2}$) [66]. Balances adjacency and containment equally. Note that one can multiply both sides of Equation 2.5 by two to eliminate the fractional scalar.



(i): Hypergraph H **(ii):** Canonical decomposition D_H^c **(iii):** Hypercactus H^*

Figure 2.3: The canonical decomposition D_H^c and the associated hypercactus H^* of a hypergraph H. For H^*, the elements of D_H^c are first transformed and then connected via the marker vertex x. The vertex mapping is the identity, except for $\phi_{H^*}(f) = \phi_{H^*}(g) = b$.

2.2.2 Hypercactus

For every hypergraph $H = (V, E)$, there exists a so-called *hypercactus* $H^* = (V^*, E^*)$, i.e., a compact representation of size $O(n)$ that preserves all minimum cuts of H [17]. A hypercactus H^* implicitly defines an (injective) vertex mapping $\phi_{H^*} : V \to V^*$, so that each minimum cut in H corresponds to a minimum cut in H^* and vice versa. In structural terms, a hypercactus is defined as a hypergraph in which each *block* is either a single hyperedge or a graph cycle. A block is defined as an inclusion-wise maximal vertex-induced sub-hypergraph that does not contain a so-called *articulation vertex*, i.e., a vertex whose removal would disconnect the hypergraph into at least two separate components [15]. Note that the hypercactus representation is not necessarily unique for a given hypergraph.

A *decomposition* D_H of a hypergraph H is a set of (contracted) hypergraphs that is obtained by starting at $\{H\}$ and repeatedly replacing one of the hypergraphs in the set by its *simple refinement*. A simple refinement $\{H_1, H_2\}$ of a hypergraph H is obtained by contracting the partitions of a split (V_1, V_2), i.e., $H_1 := H \,/\, V_1$ and $H_2 := H \,/\, V_2$. Note that both contractions create the same contracted *marker vertex* x, so that the refinement can be easily undone by identifying and merging the respective marker vertex x. An element in the decomposition is called *prime* if it does not contain any split. A *solid polygon* is a member of the decomposition that only consists of a hyperedge containing all vertices and a graph cycle where each edge has the same weight. We call a decomposition *prime* if it consists only of prime elements and *standard* if it every element is either prime or a solid polygon. The *canonical* decomposition D_H^c is the minimal standard decomposition, i.e., it is standard and was obtained by refining a non-standard decomposition. The submodularity of the cut function implies that every hypergraph H admits a unique canonical decomposition [19], which can be used to construct a hypercactus representation [17], as shown in Figure 2.3. For this, the elements of D_H^c are first transformed and then connected via the marker vertices.

(i): Graph G with a maximum flow f (in blue) **(ii):** Residual graph G_f

Figure 2.4: The residual graph G_f of an example graph G with a maximum flow f.

2.2.3 Related Problems

The *clustering* problem is closely related to the minimum cut problem, but there are some key differences between them. We briefly outline the clustering problem, as we will later describe how computing a clustering can be used as a heuristic for solving the minimum cut problem. Finding a *clustering* in a hypergraph H consists of separating the vertices into k disjoint *clusters* $V_1 \cup \cdots \cup V_k = V$, such that an objective is optimized that ensures intra-cluster density and inter-cluster sparsity. The parameter k is not given in advance and the clusters may be of any size. *Label propagation* is a common clustering algorithm that was originally proposed for graphs [67] but can also be extended to hypergraphs [39].

A *flow network* $F = (G, f, s, t)$ consists of a directed, weighted graph $G = (V, E)$ and a function $f : E \to \mathbb{R}_{\geq 0}$, which assigns to each edge $e \in E$ a non-negative *flow* $f(e)$ that is upper-bounded by the weight of the edge, i.e., $\forall e \in E : 0 \leq f(e) \leq \omega(e)$. The goal of the *maximum s-t flow* problem is to find a feasible flow f that maximizes the value of the flow $val(f)$, i.e., the total flow starting at the *source* vertex $s \in V$ and ending in the *sink* vertex $t \in V$. Note that for a flow to be feasible, it must ensure the *flow conservation*, i.e., each vertex other than the source and the sink must receive as much flow as it gives. By construction, the flow value is equal to the flow originating from the source and received at the sink. The *max-flow min-cut theorem* states that for a flow network F, the maximum flow value passing from s to t is identical to the minimum s-t cut value [26]. Thus, finding a minimum s-t cut in any graph is the same as introducing the flow function f and finding a maximum flow from s to t through this graph. Most algorithms that solve the maximum flow problem construct the so-called *residual graph* $G_f = (V_f, E_f)$, which uses the same vertex set as G, i.e., $V_f := V$, and whose edge set E_f is constructed by adding each edge $e \in E$ where $f(e) < \omega(e)$ with the new weight $\omega_f(e) = \omega(e) - f(e)$. Furthermore, for each edge $e \in E$ where $f(e) > 0$, we add the reversed edge e^{ref}, i.e., the flipped ordered pair of e, with the new weight $\omega_f(e^{\text{ref}}) = f(e)$. If the flow f has maximum value, the residual graph G_f contains enough information to retrieve all minimum s-t cuts of the original graph G. More specifically, every minimum s-t cut of G is represented by a so-called *closed vertex set* containing s but not t in G_f [63]. A closed vertex set $C \subsetneq V_f$ is defined as a set that has no edges leaving the set, i.e., there is no directed edge (u, v) such that u is included in C but v is excluded from C. Figure 2.4 visualizes a graph G with a maximum s-t flow f and its associated residual graph G_f.

2.3 Kernelization

For a given problem, *kernelization* is the concept of preprocessing the input to replace it with a smaller instance. The solution to this smaller instance is either identical to that of the original input, or it can easily be transformed into it. The reduced instance is referred to as the *kernel* of the input. Intuitively, kernelization is a technique that removes all parts of an input that are irrelevant to solving the given problem. Since the kernel can most often be computed in polynomial time, kernelization is commonly used for NP-hard problems in the context of *fixed-parameter tractability*. However, it can also improve the efficiency of polynomial-time algorithms, which is the focus of this thesis. In general, a series of so-called *reduction rules* are applied to transform an input instance into its corresponding kernel. These reduction rules identify specific structures or patterns that can be removed without losing important information about the solution. For the hypergraph minimum cut problem, we restrict the reduction rules to the removal of hyperedges and the contraction of a set of vertices. Note that, for simplicity, we also speak of kernelization and kernel when applying clustering contraction as a heuristic reduction, even though the exact minimum cut value of the original hypergraph may be lost.

2.4 Linear Programs

A *linear program* is a mathematical formulation of an optimization problem, in which a linear objective function should be maximized or minimized by choosing the appropriate values of the *variables*. During the optimization process, the variables must satisfy a series of linear (in)equalities, referred to as *constraints*. Linear programs are categorized by the domain over which the variables are defined. For an *Integer Linear Program* (ILP), all variables are restricted to integer values. If only some of the variables are integers, the problem is called *Mixed-Integer Linear Program* (MILP). In a *Binary Integer Program* (BIP), the variables are required to be binary. Note that the task of solving any problem defined as an ILP, MILP or BIP is NP-complete [47], meaning that finding the optimal values for the variables can take exponential time in the worst case. Nonetheless, the ability to use specialized and scalable solvers for any linear program has proven to be useful in practice.

CHAPTER 3

Related Work

The field of research for the hypergraph minimum cut problem has a long history, with researchers studying it for over fifty years. First, we outline the steady improvements that have been made in solving the minimum cut problem for the graph scenario. Subsequently, we describe in detail how this work has been extended and optimized for the hypergraph scenario, including advances in parallelization and the representation of all minimum cuts.

3.1 Graph Minimum Cut

The first graph minimum cut algorithm was proposed by Gomory and Hu [31], who showed that a (global) minimum cut can be computed from $n - 1$ minimum s-t cuts. The idea is that a minimum cut always separates a given vertex $s \in V$ from at least one other vertex, meaning that it can be determined by computing, for each other vertex $t \in V \setminus \{s\}$, a minimum s-t cut, and then selecting one of the cuts with the lowest value. As a result of the max-flow min-cut theorem of Ford and Fulkerson [26], the same can be achieved by computing $n - 1$ maximum s-t flows. Thus, for many decades, the most effective way to improve the minimum cut computation was to find a better maximum s-t flow algorithm, such as the push-relabel algorithm by Goldberg and Tarjan [30].

Nagamochi et al. [59, 60] introduce the first algorithm for finding a minimum cut in a graph without relying on flow computations. Their idea is to repeatedly identify edges that can safely be contracted without losing information about the minimum cut value. To achieve this, they define an upper bound $\hat{\lambda}(G)$ of the minimum cut value, which is initialized with the minimum weighted vertex degree δ_ω. Furthermore, they exploit the fact that, for any two vertices $s, t \in V$, a minimum cut either separates s from t or the two vertices can be safely contracted without altering the minimum cut value:

$$\lambda(G) = \min\{\lambda(G, s, t), \lambda(G \,/\, \{s, t\})\} \qquad \forall s, t \in V \tag{3.1}$$

Importantly, Nagamochi et al. only consider adjacent s and t and they estimate the minimum s-t cut value with a lower bound. More precisely, they determine, for each edge $e \in E$ with endpoints $u, v \in V$, a lower bound $q(e)$ of the minimum u-v cut value. Based on Equation 3.1, the edge e can be safely contracted if the lower bound $q(e)$ is larger or equal to the upper bound $\hat{\lambda}(G)$. Note that if $q(e) = \hat{\lambda}(G)$, the contraction may destroy a minimum cut, but this is accounted for by updating and maintaining $\hat{\lambda}(G)$. The algorithm uses at most $n - 1$ iterations, since at least one edge can be contracted in each iteration. In practice, fewer iterations are needed, as multiple edges can usually be contracted at once. The algorithm has an asymptotic time complexity of $O(mn + n^2 \log(n))$ and is currently the fastest known deterministic algorithm.

Stoer and Wagner [72] propose a variant of the algorithm of Nagamochi et al. that is conceptually simpler and has the same time complexity. In particular, they observe that the algorithm of Nagamochi et al. implicitly computes a maximum adjacency (MA) ordering of the vertices. This means that the last two vertices v_{n-1} and v_n in the ordering form a pendent pair (v_{n-1}, v_n) and thus the trivial cut $\{v_n\}$ is a minimum cut that separates v_{n-1} from v_n, i.e., $\lambda(G, v_{n-1}, v_n) = d_\omega(v_n)$. By initialization, we have $\hat{\lambda}(G) \leq d_\omega(v_n)$, meaning that it is sufficient to contract the last two vertices of the ordering in each iteration, even if they are not connected by an edge. After each iteration, the minimum weighted vertex degree is re-computed and $\hat{\lambda}(G)$ is updated if necessary. The algorithm of Stoer and Wagner performs worse in practice [43], since each iteration identifies exactly one contraction, while the algorithm of Nagamochi et al. usually contracts multiple edges per iteration.

Matula [58] modifies the algorithm of Nagamochi et al. by setting the initial value of the upper bound $\hat{\lambda}(G)$ to $(\frac{1}{2} - \varepsilon) \cdot \delta_\omega$, resulting in a $(2 + \varepsilon)$-approximation algorithm with linear time complexity. The tighter initial value causes more edges to be contracted, but at the cost of losing the guarantee of obtaining the exact minimum cut value.

Padberg and Rinaldi [62] define a series of exact reduction rules that can be used to repeatedly contract the input graph into a smaller kernel, so that the original minimum cut value can still be retrieved. Some of these reduction rules depend on $\hat{\lambda}(G)$ and can thus become more effective if the upper bound is further reduced. However, contrary to the algorithm of Nagamochi et al., there is no guarantee of finding a valid edge contraction. This means at some point, it may no longer be possible to apply any reduction rule and thus no further edges can be contracted. In this case, the algorithm computes a maximum s-t flow in order to identify at least one edge that can be contracted. Chekuri et al. [16] offer a linear-time implementation of the reduction rules.

Karger and Stein [46] introduce a randomized Monte Carlo algorithm that is based on random edge contractions. The algorithm runs in $O(n^2 \log(n))$ time and finds a minimum cut with a probability of $\Theta(\log^{-1}(n))$. Repeating the algorithm $log^2(n)$-times improves the success probability to $O(n^{-1})$. Karger [44] further optimizes the algorithm by using techniques such as cut sparsification and tree packing.

Henzinger et al. propose VIECUT [41], which is the current state-of-the-art algorithm for the graph minimum cut problem in both sequential and shared-memory settings. First, they quickly determine a good initial value for the upper bound $\hat{\lambda}(G)$ by using a heuristic

algorithm [40], which is based on label propagation and the reduction rules of Padberg and Rinaldi. This initialization is usually much better than simply taking the minimum weighted vertex degree δ_ω. Then, they return to the initial graph and re-apply the reduction rules of Padberg and Rinaldi with the improved upper bound $\hat{\lambda}(G)$ to obtain a small kernel. Finally, they use an extended and parallelized version of the algorithm of Nagamochi et al. to determine the exact minimum cut value of the kernel, which can easily be transferred to the original graph. Building on their previous work and the findings of Nagamochi et al. [61], Henzinger et al. [42] also propose a fast algorithm for transforming a graph into its associated cactus representation, which preserves all minimum cuts.

3.2 Hypergraph Minimum Cut

In the hypergraph scenario, finding a minimum cut is more challenging because hyperedges can represent multi-way relations of any size, making even simple tasks such as traversal and representation more complex. In 1973, Lawler [54] was the first to prove that the hypergraph minimum s-t cut problem for any $s, t \in V$ can be solved in polynomial time. This is achieved by transforming the hypergraph into its associated digraph representation and computing a maximum s-t flow on it. Consequently, a (global) minimum cut of the hypergraph can also be computed in polynomial time, since it can be derived from $n - 1$ minimum s-t cuts, which is analogous to the graph scenario.

Queyranne [66] shows that the hypergraph minimum cut function is both symmetric and submodular. Therefore, any submodular minimization algorithm can be used to compute a minimum cut in a hypergraph. However, specialized algorithms are preferred in practice, since direct methods without any additional restrictions always yield invalid solutions [23]. In particular, Queyranne extends the algorithm of Stoer and Wagner to hypergraphs, since vertex orderings, such as the MA ordering, can be applied to any symmetric submodular function. Note that the algorithm computes a so-called Queyranne ordering but the idea is the same as for the MA ordering, i.e., the last two vertices in the ordering form a pendent pair and can therefore be contracted in each iteration. In fact, a particularity of hypergraphs is that using a different vertex ordering results in a different algorithm, whereas all vertex orderings collapse to the same definition on graphs, leading to the same result. Klimmek and Wagner [52] propose a variant that uses the MA ordering, while Mak and Wong [56] introduce a variant that computes a so-called tight ordering. Chekuri and Xu [15] prove that all of these vertex orderings are special cases of a more general α-ordering with $\alpha \in [0, 1]$. This means that there is an infinite number of possible vertex orderings, each resulting in a different variant of the ordering-based algorithm. All variants require $O(np + n^2 \log(n))$ time for weighted and $O(np)$ time for unweighted hypergraphs.

Chekuri and Xu [14, 15] introduce the idea of first trimming the input hypergraph in a preprocessing phase, before passing it to one of the variants of the ordering-based algorithm. Trimming is a technique where individual pins are removed from hyperedges without deleting the hyperedges themselves. In particular, they construct a so-called k-trimmed

certificate H_k with $O(kn)$ hyperedges for a given k, where all local connectivities up to k are preserved, i.e., $\lambda(H_k, s, t) \geq \min\{k, \lambda(H, s, t)\}$ for all $s, t \in V$. The idea is that the minimum cut value is guaranteed to be preserved if $\lambda(H_k) < k$. Therefore, the algorithm starts at $k = 2$ and performs multiple iterations, each of which uses the ordering-based algorithm on the k-trimmed certificate H_k to compute $\lambda(H_k)$. If $\lambda(H_k) < k$, the minimum cut value is preserved, i.e., $\lambda(H) = \lambda(H_k)$, and the algorithm stops. Otherwise, k is doubled and the next iteration starts. To improve the efficiency of the algorithm, Chekuri and Xu first construct a data structure from which they can quickly retrieve the k-trimmed certificate H_k for any k. The algorithm only applies to unweighted hypergraphs and has a time complexity of $O(p + \lambda n^2)$. In addition, Chekuri and Xu provide a theoretical description of how the $(2 + \varepsilon)$-approximation algorithm of Matula can be extended to hypergraphs.

More recently, the hypergraph minimum cut problem has been approached from several novel perspectives. Walter and Witteveen [77] propose an algorithm that approximates the minimum cut value using the entropies of quantum states. They show that, in theory, the entropy function can be used to approximate the hypergraph minimum cut function with high probability to arbitrary precision. Veldt et al. [75] introduce a series of splitting functions that impose different penalties on cutting hyperedges based on the way they are cut. They also demonstrate that, for some of these splitting functions, finding a minimum cut in the hypergraph becomes NP-hard.

To the best of our knowledge, there exists currently no parallel algorithm that efficiently solves the hypergraph minimum cut problem. The only existing approach is to naively transform the hypergraph into its associated digraph representation and perform $n - 1$ parallel s-t flow computations. Gottesbüren et al. [32] have used this approach in the context of hypergraph partitioning, relying on a parallel version of the push-relabel algorithm [6]. However, they emphasize that this method is not optimal, as the parallelization of s-t flow computations still remains challenging.

In terms of finding all minimum cuts of a hypergraph, some work has been done for transforming a hypergraph into a hypercactus, i.e., a compact representation of size $O(n)$ that preserves all minimum cuts of the hypergraph. Cheng [17] shows that every hypergraph has a hypercactus representation, since it can be derived from the canonical decomposition of the hypergraph, whose existence and uniqueness has been proven by Cunningham [19]. Based on these results, Chekuri and Xu [14, 15] provide an extensive theoretical description of a hypercactus algorithm, whose time complexity is equivalent to that of computing a single minimum cut. He et al. [37] propose a randomized Monte Carlo algorithm that returns a hypercactus with high probability. Their algorithm is almost-linear and has even been de-randomized in subsequent work [38], although only for the graph scenario.

CHAPTER 4

Hypergraph Minimum Cut Algorithms

Although the minimum cut problem can be solved efficiently on graphs, the hypergraph setting remains more challenging. Most state-of-the-art hypergraph algorithms are rigorous theoretical extensions of graph algorithms but they fail to achieve satisfactory execution times in practice. To tackle this, we propose several algorithms to solve the hypergraph minimum cut problem efficiently. We start by formulating a Binary Integer Program (BIP) in Section 4.1, as this provides us with a simple and straightforward algorithm that can be used as a baseline for the other algorithms. In Section 4.2 we define a Mixed-Integer Linear Program (MILP) that, contrary to the BIP formulation, achieves a linear number of constraints at the cost of a slightly higher number of variables. Our main contribution is described in Section 4.3, where we present HEICUT, a lightweight and scalable algorithm that finds a minimum cut in unweighted or weighted hypergraphs with up to hundreds of millions of hyperedges. More specifically, HEICUT applies a series of exact reduction rules to aggressively reduce the input hypergraph to a smaller kernel without losing information about the original minimum cut value. Besides, HEICUT supports the option of performing a heuristic reduction by identifying and contracting a clustering via label propagation. Note that, for simplicity, we also speak of kernelization and kernel when applying clustering contraction as a heuristic reduction, even though the exact minimum cut value of the original hypergraph may be lost. After the kernelization, HEICUT uses one of the variants of an existing exact ordering-based algorithm to determine the minimum cut value of the kernel. For this, we extend the ordering-based algorithm, so that it searches for multiple contractions per iteration. Section 4.4 outlines the shared-memory parallelization of our algorithms. In particular, we show that the kernelization phase of HEICUT can be parallelized with very little effort. Besides, we present, to the best of our knowledge, the first ever parallel implementation of the ordering-based algorithm.

4.1 Binary Integer Program

We can obtain a simple algorithm for the hypergraph minimum cut problem by generalizing the well-known BIP formulation for graphs to the hypergraph scenario. More specifically, for each vertex $v \in V$, we define a binary variable $x_v \in \{0, 1\}$ that indicates to which of the two partitions the vertex belongs. In addition, we define for each hyperedge $e \in E$ a binary variable $y_e \in \{0, 1\}$ that indicates whether the hyperedge is cut, i.e., whether it runs between the two partitions or not. The formulation has a total of $n + m$ variables. The hypergraph minimum cut problem consists of minimizing the sum of the weights of all hyperedges running between the two partitions, which leads to Objective Function 4.1. In addition, to ensure that the resulting minimum cut is valid, several constraints must be enforced. Firstly, neither of the two partitions can be empty, i.e., each partition must contain at least one vertex, as defined in Constraint 4.2 and Constraint 4.3. Secondly, the vertex variables must be properly linked to the hyperedge variables. This means that if two adjacent vertices are not in the same partition, each hyperedge that is incident to both of these vertices must be cut and therefore have a non-zero variable. In other words, a hyperedge is cut if contains pins that belong to different partitions, as formalized by Constraint 4.4. Note that even if all of the pins of a hyperedge belong to the same partition, the constraint does not force the variable of the hyperedge to be zero. However, such solutions are always sub-optimal, i.e., it is possible to reduce the objective even further by setting the variable of the hyperedge down to zero without breaking any of the constraints. The total number of constraints is equal to $2 - m + \sum_{e \in E} |e|^2$ and therefore not linear in the size of the hypergraph, since each hyperedge introduces a constraint for every possible ordered pair of its pins.

$$\min \sum_{e \in E} \omega(e) \cdot y_e \tag{4.1}$$

$$\sum_{v \in V} x_v \geq 1 \tag{4.2}$$

$$\sum_{v \in V} x_v \leq n - 1 \tag{4.3}$$

$$\forall e \in E : \forall (u, v) \in e \times e, u \neq v : \quad y_e \geq x_u - x_v \tag{4.4}$$

In practice, it is common to relax the binary restriction on the variables during the computation. This means that the variables can take any floating-point value that lies within a predefined tolerance around the binary values. For the final solution, the variables are rounded back to the nearest binary value. Relaxing the BIP formulation generally improves scalability, as solving the exact BIP formulation is computationally expensive. For a small enough tolerance such as 10^{-7}, it is extremely unlikely that this rounding heuristic leads to sub-optimal solutions. Nonetheless, for theoretical clarity, we refer to the relaxed BIP as near-optimal.

4.2 Mixed-Integer Linear Program

A disadvantage of the straightforward BIP formulation in Section 4.1 is that it comes with a non-linear number of constraints, which may lead to inefficient solving times in practice. This is especially true for hypergraphs with a high average hyperedge size. Fortunately, we observe that Constraint 4.4 can be simplified further, as it contains information that is irrelevant for linking the vertex variables to the hyperedge variables. In particular, to correctly mark a hyperedge as cut, it is irrelevant to know which two incident pins belong to different partitions. This is because knowing about the existence of such a pair is all that is necessary. Therefore, one way to reduce the number of constraints is to introduce additional auxiliary variables. More precisely, we define for each hyperedge $e \in E$ the real variable $z_e^{\min} \in \mathbb{R}$, which represents the minimum vertex variable among all pins of e. Analogously, we define the real variable $z_e^{\max} \in \mathbb{R}$ to represent the maximum vertex variable among the pins of e. This way, the hyperedge e is cut if and only if its two auxiliary variables differ, i.e., $z_e^{\min} \neq z_e^{\max}$. Note that we do not restrict the auxiliary variables to be integer or binary, as this would make the formulation too rigid and the new constraints will indirectly enforce binary values on the auxiliary variables. This means that we formulate an MILP with $n + 3m$ variables, consisting of the real auxiliary variables $z_e^{\min}, z_e^{\max} \in \mathbb{R}$ as well as the binary variables $x_v \in \{0, 1\}$ and $y_e \in \{0, 1\}$ as defined in Section 4.1. The objective and the first two constraints are identical to those of the BIP. However, we replace Constraint 4.4 with the following new constraints:

$$\forall e \in E : \forall v \in e : \quad z_e^{\min} \leq x_v \tag{4.5}$$

$$\forall e \in E : \forall v \in e : \quad z_e^{\max} \geq x_v \tag{4.6}$$

$$\forall e \in E : \quad y_e \geq z_e^{\max} - z_e^{\min} \tag{4.7}$$

Constraint 4.5 and Constraint 4.6 ensure that the auxiliary variables represent the minimum and maximum vertex variables among the pins of each hyperedge. The identification of cutting hyperedges is formalized by Constraint 4.7. In total, the MILP formulation consists of $2 + 2p + m$ constraints. Therefore, compared to the BIP formulation, the MILP achieves a linear number of constraints at the cost of a slightly higher number of variables. Similar to the BIP, we make use of the rounding heuristic, i.e., we relax the restriction on the binary variables of the MILP formulation. When returning the solution, we round these values to the nearest binary value. The rounding heuristic is useful for solving the MILP formulation on larger hypergraphs and only leads to sub-optimal solutions in very rare cases. Thus, we consider the relaxed MILP to be near-optimal in theoretical terms.

4.3 HeiCut

Although computing a minimum cut in a hypergraph is a fundamental task, the multi-way relations of hyperedges introduce greater structural and computational complexity than in the graph scenario. Directly applying an exact algorithm to large hypergraphs often results in unsatisfactory memory or execution time performance. Therefore, we present HEICUT, a lightweight and scalable algorithm that finds a minimum cut in unweighted or weighted hypergraphs of arbitrary size. In particular, HEICUT tackles the complexity issue from two angles. Firstly, the algorithm applies kernelization in a preprocessing phase by aggressively reducing the input hypergraph to a smaller kernel without losing information about the original minimum cut value. This is achieved through a series of provably exact reduction rules. HEICUT also supports the option to further reduce the size of the kernel heuristically by identifying and contracting clusters based on label propagation. Note that, for simplicity, we also speak of kernelization and kernel when applying clustering contraction as a heuristic reduction, even though the exact minimum cut value of the original hypergraph may be lost. Secondly, HEICUT refines and extends the work on the exact ordering-based algorithm by searching for multiple contractions per iteration. Applying this extension on the kernel allows to quickly determine its associated minimum cut value. Overall HEICUT can be considered as the hypergraph counterpart of the graph algorithm VIECUT [41], with which it shares several conceptual similarities.

4.3.1 Reduction Rules

In a preprocessing phase, we iteratively reduce the hypergraph to a smaller kernel before passing it to the exact ordering-based algorithm. More precisely, we define a set of seven provably exact reduction rules. Each of these reduction rules identifies specific structures or patterns within the hypergraph that can be safely removed or contracted without losing information about the original minimum cut value. This kernelization idea was originally proposed by Padberg and Rinaldi [62], refined by Chekuri et al. [16] and efficiently implemented by Henzinger et al. [41] for graphs. However, to the best of our knowledge, it was never transferred to the hypergraph scenario.

In detail, HEICUT maintains an upper bound $\hat{\lambda}(H)$ of the minimum cut value throughout the entire kernelization phase. We initialize $\hat{\lambda}(H)$ with the minimum trivial cut value, i.e., the minimum weighted vertex degree δ_ω. At any moment, the upper bound guarantees that there exists a cut in the hypergraph H with a value of $\hat{\lambda}(H)$. The core idea of the algorithm is to assume that the hypergraph admits an even better cut with a value strictly lower than $\hat{\lambda}(H)$. Based on this assumption, we repeatedly identify and eliminate any structure in the hypergraph for which we can ascertain that it never crosses such a better cut. In other words, if such a better cut does exist, the identified structure will lie entirely within one of the two partitions and can therefore safely be removed or contracted without destroying the better cut. If the assumption is incorrect, i.e., there is no cut with a value

strictly lower than $\hat{\lambda}(H)$, then $\hat{\lambda}(H)$ is already the minimum cut value and applying the reduction rules helps to reach this conclusion more quickly. In this case, the kernelization does not necessarily preserve the minimum cut value, i.e., we may end up with a kernel whose minimum cut value is greater than that of the input hypergraph. However, since the upper bound $\hat{\lambda}(H)$ can only be decreased and is already equal to the minimum cut value, the algorithm remains correct.

The kernelization phase of HEICUT is composed of multiple rounds. In each round, the seven reduction rules are applied in a fixed predefined order. After each reduction rule, the size of the hypergraph is reduced by removing or contracting the identified structures before applying the next reduction rule. While doing so, we recompute the minimum vertex degree δ_ω and update the upper bound if $\delta_\omega < \hat{\lambda}(H)$. The idea is that updating $\hat{\lambda}(H)$ after every reduction rule may make subsequent reduction rules more effective, as some of them depend heavily on the quality of the upper bound. Note that improving $\hat{\lambda}(H)$ may also make some reduction rules applicable that could not be applied before. To further speed up the performance in practice, HEICUT checks after each reduction rule if it can terminate early. This is the case if the upper bound has reached its lowest possible value, i.e., $\hat{\lambda}(H) = 0$, or if the hypergraph has been fully reduced, i.e., $|E| = 0$ or $|V| = 1$. The algorithm starts a new round of reduction rules if at least one reduction rule could be applied in the previous round. If the hypergraph cannot be reduced any further, the kernel is passed to one of the variants of the exact ordering-based algorithm.

We define two types of reduction rules. On the one hand, the *estimate-based* reduction rules make use of the upper bound $\hat{\lambda}(H)$ to compute a smaller kernel. The effectiveness of these reduction rules depends heavily on the quality of the upper bound. On the other hand, the *structure-based* reduction rules identify and exploit specific structures within the hypergraph to reduce its size. This type of reduction rule is independent of the upper bound $\hat{\lambda}(H)$, meaning that it can be effectively applied at any moment. Additionally, we distinguish between reduction rules specifically designed for hypergraphs and those that are taken from the graph scenario without being generalized to hypergraphs. The seven exact reduction rules are applied in the order in which they are presented. The idea is to start with reduction rules that have a low computational overhead but a high expected effectiveness. The graph-based reduction rules are applied last, since contractions form earlier reduction rules may increase the number of hyperedges of size two, thereby improving their applicability. Each exact reduction rule is illustrated by an example in Figure 4.1 with the exception of the first one, which is self-evident and does not require an illustration.

Reduction Rule 1 (Singleton). *Remove any hyperedge $e \in E$ where $|e| = 1$ or $\omega(e) = 0$.*

Proof: *If $|e| = 1$, the hyperedge can never cross any cut and if $\omega(e) = 0$, the hyperedge does it does not contribute to the cut value.* □

Complexity: $O(m)$ *when iterating once over all hyperedges and removing the hyperedges for which the rule is applicable.*

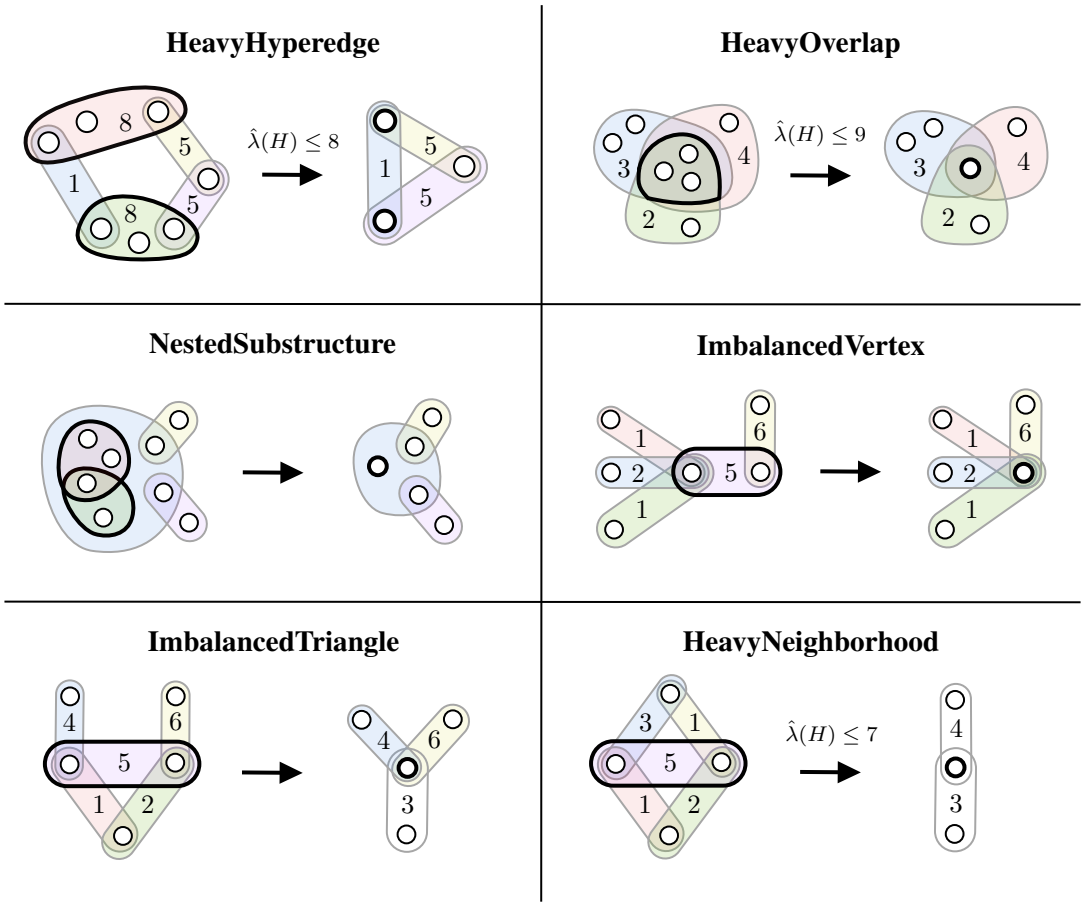


Figure 4.1: Example for each reduction rule, excluding Reduction Rule 1, which is self-evident.

Reduction Rule 2 (HeavyHyperedge). *Contract any hyperedge $e \in E$ for which $|e| \geq 2$ and $\omega(e) \geq \hat{\lambda}(H)$.*

Proof: *Assume that a hyperedge e with $|e| \geq 2$ and $\omega(e) \geq \hat{\lambda}(H)$ crosses a minimum cut with value $\lambda(H)$. Then we must have $\lambda(H) \geq \omega(e) \geq \hat{\lambda}(H)$. However, $\hat{\lambda}(H)$ is the upper bound of $\lambda(H)$, and so it follows that $\lambda(H) = \hat{\lambda}(H)$. Thus, any cut that separates the vertices of e cannot lead to a strictly smaller cut value, and e can be safely contracted without eliminating the possibility of recovering a minimum cut.* □

Complexity: *$O(n + m)$ when iterating once over all hyperedges and contracting the hyperedges for which the rule is applicable.*

Reduction Rule 3 (HeavyOverlap). *Contract any overlap $\bigcap_{i=1}^{s} e_i$ of $s \geq 2$ hyperedges where $|\bigcap_{i=1}^{s} e_i| \geq 2$ and $\sum_{i=1}^{s} \omega(e_i) \geq \hat{\lambda}(H)$.*

Proof: *Assume that an overlap $\bigcap_{i=1}^{s} e_i$ of $s \geq 2$ hyperedges is cut by a minimum cut with value $\lambda(H)$. Then each hyperedge e_i of the overlap contributes with a weight of $\omega(e_i)$ to*

the minimum cut value. It follows that $\lambda(H) \geq \sum_{i=1}^{s} \omega(e_i) \geq \hat{\lambda}(H)$*. Similar to the proof of Reduction Rule 2, it must hold that the minimum cut value is equal to the upper bound that we have already discovered, i.e.,* $\lambda(H) = \hat{\lambda}(H)$*. Therefore, cutting the overlap cannot lead to a strictly smaller cut value and we can safely contract the overlap.* □

Complexity: $O(n + \sum_{e \in E} |e|^2)$ *when checking the neighborhood of every vertex* $v \in V$ *with* $d_\omega(v) \geq \hat{\lambda}(H)$*.*

Definition 4.1 (Strictly Nested Substructure)
For a hyperedge $e \in E$*, we define a nested substructure of* e *as a set of* $s \geq 1$ *different hyperedges* $\{e_1, \ldots, e_s\} \subseteq E \setminus \{e\}$ *such that all hyperedges are contained within* e*, i.e.,* $e_i \subseteq e$ *for all* $1 \leq i \leq s$*. We call it a strictly nested substructure if the hyperedges and their union are strictly contained in* e*, i.e.,* $e_i \subsetneq e$ *for all* $1 \leq i \leq s$ *and* $\bigcup_{i=1}^{s} e_i \subsetneq e$*.*

Definition 4.2 (Escaping Path)
For a hyperedge $e \in E$*, we define an escaping path of* e *as a connected path* P *between two vertices* $u \in e$ *and* $v \notin e$*, such that none of the hyperedges in the path* P *fully contain the hyperedge* e*, i.e., every hyperedge* $e' \in P$ *satisfies* $e \not\subseteq e'$*.*

Reduction Rule 4 (NestedSubstructure). *Contract any inclusion-wise maximal strictly nested substructure of a hyperedge* $e \in E$ *for which there exists no escaping path of* e *that starts in the substructure.*

Proof: *Let* $U = \bigcup_{i=1}^{s} e_i$ *be the union of the hyperedges of an inclusion-wise maximal strictly nested substructure of* e*. Suppose there is no escaping path of* e *that starts in* U*. Then all paths from* U *to* $V \setminus U$ *must traverse* e *or a hyperedge that fully contains* e*. Now, consider an arbitrary minimum cut for which at least one hyperedge* $e_i \subseteq U$ *is cut. It follows that every hyperedge that connects* U *with* $V \setminus U$ *also crosses the cut, since it is either* e *or a hyperedge that fully contains* e*, i.e., it contains all vertices in* U*. This means that moving* U *entirely on one side of the cut does not lead to new cut hyperedges and decreases the cut value by the weight of the previously cut hyperedges* e_i*. Thus, we can safely contract* U*.* □

Complexity: $O(n + m + \Delta \cdot p + \sum_{e \in E} \sum_{e' \in E \wedge e' \subsetneq e} |e'|)$ *as each hyperedge* $e \in E$ *goes over all its pins and their incident hyperedges to detect strictly nested substructures. Traversals are performed to verify whether pins can escape through other incident hyperedges.*

Reduction Rule 5 (ImbalancedVertex). *Contract any hyperedge* $e_{uv} = \{u, v\} \in E$ *for which* $d_\omega(u) < 2\omega(e_{uv})$ *or* $d_\omega(v) < 2\omega(e_{uv})$*.*

Proof: *We only apply the reduction rule to hyperedges of size two, which have the same properties as regular graph edges. The intuition is that the hyperedge* e_{uv} *never crosses a minimum cut, because shifting either* u *or* v *to the other partition would decrease the cut value. The validity of a single application of the reduction rule has been proven by Padberg and Rinaldi [62]. When being applied multiple times, the inequalities of the reduction rule must be strict to avoid erroneous contractions. Otherwise, if two hyperedges* e_{uv} *and* e_{vw}

have equal weight and share a common pin v, both are independently selected for contraction. However, this assumes that v can be shifted to opposite sides at the same time, resulting in invalid contractions that may destroy minimum cuts, as shown in Figure 4.2. □

Complexity: *$O(n + m)$ when iterating once over all hyperedges and contracting the hyperedges for which the rule is applicable.*

Reduction Rule 6 (ImbalancedTriangle). *Contract any hyperedge $e_{uv} = \{u, v\} \in E$ where there exists a vertex $w \in V$ such that $e_{uw} = \{u, w\} \in E$ and $e_{vw} = \{v, w\} \in E$ with $d_\omega(u) \leq 2\left(\omega(e_{uv}) + \omega(e_{uw})\right)$ and $d_\omega(v) \leq 2\left(\omega(e_{uv}) + \omega(e_{vw})\right)$.*

Proof: *We only apply the reduction rule to hyperedges of size two, which have the same properties as regular graph edges. The intuition is that the hyperedge e_{uv} never crosses a minimum cut, because then e_{uw} or e_{vw} also crosses this minimum cut and shifting either u or v to the other partition would decrease the cut value. The validity of a single application of the reduction rule has been proven by Padberg and Rinaldi [62]. When being applied multiple times, we follow the approach of* VieCut *[41], i.e., we enforce the restriction that each vertex can participate in at most one contraction.* □

Complexity: *$O(n+m)$ when marking the vertices similar to* VieCut *and thus not looking at all possible triangles.*

Reduction Rule 7 (HeavyNeighborhood). *Contract any hyperedge $e_{uv} = \{u, v\} \in E$ for which $\omega(e_{uv}) + \sum_{w \in U} \min\{\omega(e_{uw}), \omega(e_{vw})\} \geq \hat{\lambda}(H)$ with the set U being defined as $U := \{w \in V : e_{uw} = \{u, w\} \in E \wedge e_{vw} = \{v, w\} \in E\}$.*

Proof: *We only apply the reduction rule to hyperedges of size two, which have the same properties as regular graph edges. The intuition is that if the hyperedge e_{uv} crosses a minimum cut, then e_{uw} or e_{vw} also crosses this minimum cut for every $w \in U$, which means that $\lambda(H) \geq \sum_{w \in U} \min\{\omega(e_{uw}), \omega(e_{vw})\} \geq \hat{\lambda}(H)$. Thus, similar to the proof of Reduction Rule 2, we have $\lambda(H) = \hat{\lambda}(H)$. The validity of a single application of the reduction rule has been proven by Padberg and Rinaldi [62]. When being applied multiple times, we follow the approach of* VieCut *[41], i.e., we enforce the restriction that each vertex can participate in at most one contraction.* □

Complexity: *$O(n+m)$ when marking the vertices similar to* VieCut *and thus not looking at all possible triangles.*

Reduction Rules 2, 3 and 7 are estimate-based, whereas Reduction Rules 1, 4, 5 and 6 are structure-based. Besides, the four graph-based reduction rules taken from VieCut are Reduction Rules 2, 5, 6 and 7, which were originally proposed by Padberg and Rinaldi. We only generalize Reduction Rule 2 to hyperedges of any size. Generalizing the other graph-based reduction rules to the hypergraph scenario could be future work. Note that, contrary to the formulation by Padberg and Rinaldi, the inequality in Reduction Rule 5 must be strict when it is applied to multiple hyperedges at the same time. Otherwise, two hyperedges e_{uv}

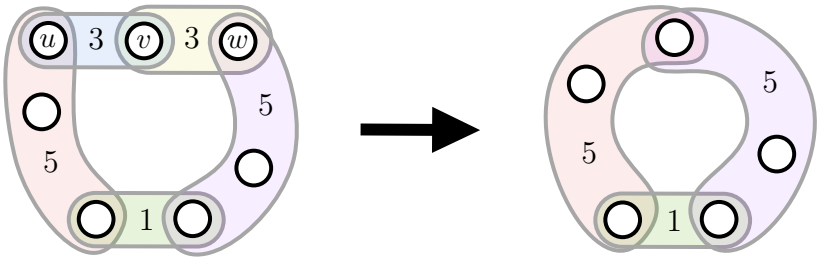


Figure 4.2: Example of a hypergraph H with $\lambda(H) = 4$. When using a non-strict inequality in Reduction Rule 5, we contract the hyperedges e_{uv} and e_{vw} at the same time. After the contraction, we have $\lambda(H) = 5 > 4$, i.e., the original minimum cuts got destroyed.

and e_{vw} with the same weight and incident to a common vertex v will both be contracted, since they assume that they can shift v so that the respective other hyperedge crosses the minimum cut. This means that v must be in both partitions at the same time, which is impossible and therefore minimum cuts may be destroyed when contracting both hyperedges. An example of this issue is visualized by Figure 4.2, where we have $\omega(e_{uv}) = \omega(e_{vw}) = 3$. This problem also occurs in the graph scenario and although the strict inequality was implemented correctly in the code of VieCut, the associated paper [40] defines the reduction rule with a non-strict equality. Thus, to the best of our knowledge, we are the first to make explicitly aware of this important detail. Note that the inequality in Reduction Rule 6 can remain non-strict, since we ensure that each vertex can only participate in at most one contraction. Additionally, the authors of VieCut mention that its four graph-based reduction rules only apply if e_{uv} is not the only edge incident to the endpoints u and v. This restriction should prevent contractions that may eliminate minimum cuts. However, we observe that this restriction is not necessary in practice, since it was shown by Padberg and Rinaldi that only trivial minimum cuts may not be preserved. As this is already handled by computing δ_ω and updating $\hat{\lambda}(H)$ after each reduction rule whenever $\delta_\omega < \hat{\lambda}(H)$, this restriction is not needed in our setting.

4.3.2 Label Propagation

As there is no guarantee that any of the reduction rules in Section 4.3.1 can be applied to the hypergraph, HeiCut supports the option of performing a heuristic reduction, inspired by VieCut. When enabled, the algorithm identifies a clustering of the hypergraph in each round and contracts the respective dense clusters before applying the reduction rules. The intuition is that vertices within the same dense cluster are unlikely to be separated by a minimum cut, as this would lead to a large cut value. Therefore, contracting the clusters allows to aggressively reduce the size of the hypergraph. As this reduction is only a heuristic, there is no guarantee of optimality, i.e., it is possible that minimum cuts are destroyed by the reduction. However, in the graph scenario, the authors of VieCut show that the optimal solution is preserved in most cases, even for large and complex instances.

Similar to VIECUT, we use label propagation to compute the clustering, which was originally proposed by Raghavan et al. [67] for graphs but can also be extended to the hypergraph scenario [39]. In particular, each vertex starts in its own singleton cluster with a unique label corresponding to its vertex ID. The label propagation algorithm uses multiple iterations, each of which processes the vertices of the hypergraph in the same random order. More specifically, each vertex is assigned the label of the cluster to which it has the strongest connection. Note that ties are broken uniformly at random. Let L be the current set of labels in the hypergraph and C_l be the set of vertices that currently have the label $l \in L$, then the vertex $v \in V$ is assigned to the following label:

$$\arg\max_{l \in L} \left\{ \sum_{e \in I(v)} |e \cap C_l| \cdot \frac{\omega(e)}{|e| - 1} \right\} \tag{4.8}$$

This approach is known as *absorption clustering using pins* [12, 39] and is equivalent to graph-based label propagation on the clique-expanded graph, where each hyperedge $e \in E$ is replaced by a graph clique between the pins of e and each edge of the graph clique is given the weight $\frac{\omega(e)}{|e|-1}$. There exist alternative ways of determining the label of each vertex $v \in V$, but Equation 4.8 is the default hypergraph clustering approach for most algorithms, such as HMETIS [51], PATOH [12] and MT-KAHYPAR [33]. The approach runs in $O(n+\sum_{e \in E} |e|^2)$, since each vertex considers the pins of all its incident hyperedges. Kothapalli et al. [53] show that with high probability the same-label sets C_l converge to dense clusters after a few iterations.

4.3.3 Ordering-Based Solver

After the kernelization phase, HEICUT determines the exact minimum cut value of the kernel. This is achieved by using the exact ordering-based algorithm that was originally proposed for graphs by Nagamochi et al. [59, 60] and simplified by Stoer and Wagner [72]. Queyranne [66] shows that the algorithm can be extended to the hypergraph scenario due to the submodularity of the cut function. The simple intuition of the algorithm is that, for any two vertices $s, t \in V$, a minimum cut either separates s from t or the two vertices can be safely contracted without altering the minimum cut value, as formalized by Equation 3.1. In other words, the minimum s-t cut value is either identical to the (global) minimum cut value or not. Therefore, finding a minimum cut of the hypergraph H is equivalent to repeatedly identifying a minimum s-t cut for any $s, t \in V$, storing the respective cut value $\lambda(H, s, t)$ if it is lower than the current upper bound $\hat{\lambda}(H)$, and always contracting the vertices s and t. This process results in a sequence of $(n-1)$ contracted hypergraphs, each with one fewer vertex, until only one vertex remains and the algorithm terminates.

One way to quickly find a minimum s-t cut for any $s, t \in V$ is to determine a so-called pendent pair (s, t), i.e., an ordered pair for which the trivial cut $\{t\}$ is a minimum s-t cut in H. Finding such a pendent pair is straightforward due to the fact that solving the hypergraph minimum cut problem is equivalent to minimizing a symmetric submodular

function. In particular, it is possible to construct an ordering of the vertices such that the last two vertices v_{n-1} and v_n in the ordering form a pendent pair (v_{n-1}, v_n). This means that $\{v_n\}$ is a trivial minimum v_{n-1}-v_n cut and both vertices can be contracted after storing the value of $\{v_n\}$ if it is lower than the current upper bound $\hat{\lambda}(H)$. Note that the first vertex in the ordering is chosen at random. Chekuri and Xu [15] prove that tuning the parameter $\alpha \in [0, 1]$ of a more general α-ordering leads to a different vertex ordering and thus to a different variant of the ordering-based algorithm. However, this property only applies to hypergraphs, since all α-orderings collapse to the same vertex ordering on graphs. For more details, we refer to Section 2.2.1. There exist three variants of the algorithm that are most commonly used. Klimmek and Wagner [52] propose a variant that uses the maximum adjacency ordering ($\alpha = 1$), while Mak and Wong [56] compute the tight ordering ($\alpha = 0$). Queyranne [66] strikes a balance between the two implementations by defining a variant that follows the Queyranne ordering ($\alpha = \frac{1}{2}$).

The most time-consuming task of the ordering-based algorithm is recomputing the vertex ordering in every iteration. In particular, finding an α-ordering requires $O(p + n \log(n))$ time for weighted and $O(p)$ time for unweighted hypergraphs when using a Fibonacci heap. Therefore, it is natural to ask whether it is possible to improve the performance of the ordering-based algorithm by reusing the α-ordering from the previous iteration in the next one. More formally, let $(v_1, \ldots, v_n)$ be an α-ordering of the vertices of a hypergraph H and let $V_i := (v_1, \ldots, v_i)$ be the respective partial α-ordering. The hypergraph resulting from the contraction of the vertices v_{n-1} and v_n into a new vertex v^* is denoted as $H \,/\, L$ with $L := \{v_{n-1}, v_n\}$. We want to know whether $(v_1, \ldots, v_{n-2}, v^*)$ is a valid α-ordering of the hypergraph $H \,/\, L$. If so, we automatically know that (v_{n-2}, v^*) forms a pendent pair and thus $\{v^*\}$ is a minimum v_{n-2}-v^* cut in $H \,/\, L$. This means that we can perform the next iteration in linear time by storing the value of $\{v^*\}$ if it is lower than the current upper bound $\hat{\lambda}(H)$ and contracting the vertices v_{n-2} and v^* directly, without having to recompute the α-ordering from the ground up.

To further improve the performance of the ordering-based algorithm, we observe that we can reuse the α-ordering of H even if we cannot construct an α-ordering of $H \,/\, L$ in which v^* is the last vertex. In particular, although the pendent pair formed by the last two vertices of an α-ordering is order-sensitive, the contraction of these two vertices is not. Therefore, if we know that $(v_1, \ldots, v_{n-3}, v^*, v_{n-2})$ is a valid α-ordering of the contracted hypergraph $H \,/\, L$, we can still perform the next iteration in linear time by contracting the vertices v_{n-2} and v^* directly. The only difference is that, due to the order-sensitivity of the pendent pair, we must now use $\{v_{n-2}\}$ and not $\{v^*\}$ as the trivial minimum v_{n-2}-v^* cut. This leads us to Theorem 4.1.

Theorem 4.1
After performing an iteration of the ordering-based algorithm on H, the next iteration simplifies to contracting v_{n-2} and v^* directly within $H \,/\, L$ in linear time if the following property holds in the original hypergraph H for all $1 \leq i < n - 3$:

$$\alpha d_\omega(L, V_i) + (1 - \alpha) d'_\omega(L, V_i) = \alpha d_\omega(\{v_{n-1}\}, V_i) + (1 - \alpha) d'_\omega(\{v_{n-1}\}, V_i)$$

Proof of Theorem 4.1 *First, we will show that the given property ensures that V_{n-3} remains a valid partial α-ordering in $H \,/\, L$ after performing the first iteration of the ordering-based algorithm. For this, we need to construct $H \,/\, L$. Instead of contracting the vertices v_{n-1} and v_n directly, we can construct $H \,/\, L$ in two separate steps, first deleting v_{n-1} and v_n and then adding the new vertex v^*. We observe that, for all $1 \leq i < n-3$, the partial α-ordering V_i of the vertices of H still remains a valid partial α-ordering after removing the vertices v_{n-1} and v_n from H. This is because, by definition, the index of a vertex in the α-ordering is only affected by the vertices that have already been ordered, i.e., those with lower indices. Thus, removing the vertices v_{n-1} and v_n does not affect the partial α-ordering V_i, since they are not present in V_i. It remains to be shown that V_i is a valid partial α-ordering after the new vertex v^* has been added to the hypergraph. By using the given property, we can see that for all $1 \leq i < n-3$:*

$$\begin{aligned}
\alpha d_\omega(v^*, V_i) + (1-\alpha) d'_\omega(v^*, V_i) &= \alpha d_\omega(L, V_i) + (1-\alpha) d'_\omega(L, V_i) \\
&= \alpha d_\omega(\{v_{n-1}\}, V_i) + (1-\alpha) d'_\omega(\{v_{n-1}\}, V_i) \\
&\leq \alpha d_\omega(\{v_{i+1}\}, V_i) + (1-\alpha) d'_\omega(\{v_{i+1}\}, V_i)
\end{aligned}$$

In particular, this proves that V_{n-3} is a valid partial α-ordering in $H \,/\, L$. As v_{n-2} and v^* are the only vertices in $H \,/\, L$ that have not yet been ordered, it follows that either $(v_1, \ldots, v_{n-2}, v^*)$ or $(v_1, \ldots, v_{n-3}, v^*, v_{n-2})$ is a valid α-ordering of the vertices of $H \,/\, L$. In both cases, the last two vertices of the α-ordering are v_{n-2} and v^*. However, the trivial minimum v_{n-2}-v^* cut is in the first case $\{v^*\}$ and in the second case $\{v_{n-2}\}$. As every trivial cut is an upper bound of the (global) minimum cut, one way to account for this is to simply consider both trivial cuts $\{v_{n-2}\}$ and $\{v^*\}$ and store $\min(\lambda[\{v_{n-2}\}], \lambda[\{v^*\}])$ if it is lower than $\hat{\lambda}(H)$. Thus, the next iteration of the ordering-based algorithm can be simplified to contracting v_{n-2} and v^* directly, without having to recompute the α-ordering from the ground up. □

From Theorem 4.1, we can conclude that reusing the α-ordering in subsequent iterations is equivalent to performing multiple contractions in one iteration. In the following, we will stick to the concept of performing multiple contractions per iteration. Note that we still need to maintain the lowest trivial cut value found across all contractions and update $\hat{\lambda}(H)$ if necessary. This approach is more in line with the graph algorithm by Nagamochi et al. [59, 60], which also involves searching for multiple contractions in a single iteration.

For each variant of the ordering-based algorithm, we now define a recurrent rule that, if applicable, allows to perform an additional contraction in the current iteration. This means that, as long as the rule is applicable, we can add another vertex to the contraction of the current iteration. Before stating the rule, we first set the context. Let $U := \{v_j, \ldots, v_n\}$ be the set of all *eligible* vertices, i.e., the vertices that will be contracted in the current iteration. Note that v_j is the vertex with the lowest index in the α-ordering that will be contracted and every vertex coming after v_j in the α-ordering is also part of U. At the start of each iteration, we set $j = n-1$ and $U = \{v_{n-1}, v_n\}$, since the last two vertices of each α-ordering

form a pendent pair and are therefore always contracted. We observe that v_{j-1} can only be added to the set of eligible vertices U if contracting the vertices in U into a new vertex v^* would preserve the partial α-ordering V_{j-2}. In other words, either $(v_1, \ldots, v_{j-1}, v^*)$ or $(v_1, \ldots, v_{j-2}, v^*, v_{j-1})$ must be a valid α-ordering of the contracted hypergraph $H \,/\, U$, similar to the proof of Theorem 4.1. Otherwise, the vertices v_{j-1} and v^* do not necessarily form a pendent pair and we cannot add v_{j-1} to the set of eligible vertices U without the risk of destroying a non-trivial minimum cut.

We start by defining the recurrent contraction rule for the tight ordering in Theorem 4.2. As the tight ordering relies only on the containment contribution, we only need to ensure that the containment contribution of v^* is the same as that of v_j for all partial tight orderings up to V_{j-3}. Otherwise, V_{j-2} may not be preserved, as it is based on all previous partial orderings. The condition of Theorem 4.2 requires that every hyperedge incident to a vertex in $\{v_{j+1}, \ldots, v_n\}$ is also incident to either v_{j-2} or v_{j-1}. The intuition is that the containment contribution of v^* is larger than that of v_j for the partial tight ordering V_{j-3} if there is a hyperedge $e \in I(U)$ that is fully covered by $U \cup V_{j-3}$ but not by $\{v_j\} \cup V_{j-3}$ in H. In this case, we have $d'_\omega(U, V_{j-3}) = d'_\omega(\{v_j\}, V_{j-3}) + \omega(e)$. This means that v^* may be chosen earlier in the ordering, i.e., the partial tight ordering V_{j-2} may not be preserved, as shown in Figure 4.3. The condition of Theorem 4.2 ensures that this never happens for $1 \leq i < j-2$, i.e., the partial tight ordering V_{j-2} is preserved and the vertex v_{j-1} can safely be added to U.

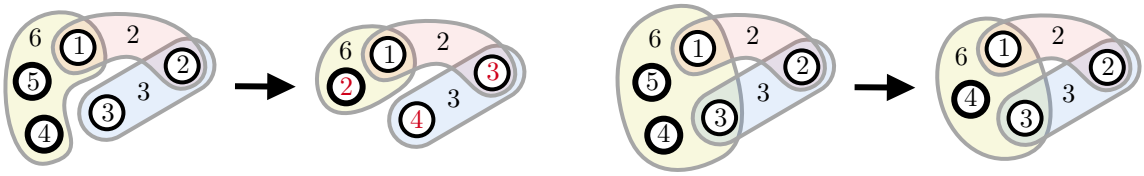
Theorem 4.2
Let $(v_1, \ldots, v_n)$ be a tight ordering and U be the set of eligible vertices of an iteration of the ordering-based algorithm. We can safely add v_{j-1} to U if the following condition holds:

$$I(\{v_{j+1}, \ldots, v_n\}) \subseteq I(\{v_{j-2}, v_{j-1}\})$$

Proof of Theorem 4.2 *For all $1 \leq i < j-2$, we see that neither v_{j-2} nor v_{j-1} is included in $U \cup V_i$. With the given condition, we conclude that $\forall\, e \in I(\{v_{j+1}, \ldots, v_n\}) : e \nsubseteq U \cup V_i$. Since we have $\alpha = 0$ for the tight ordering, it follows that for all $1 \leq i < j-2$:*

$$\begin{aligned}
\alpha d_\omega(U, V_i) + (1-\alpha) d'_\omega(U, V_i) &= d'_\omega(U, V_i) \\
&= \sum_{e \in I(U) \cap I(V_i) \wedge e \subseteq U \cup V_i} \omega(e) \\
&= \sum_{e \in I(v_j) \cap I(V_i) \wedge e \subseteq \{v_j\} \cup V_i} \omega(e) \\
&= d'_\omega(\{v_j\}, V_i) \\
&= \alpha d_\omega(\{v_j\}, V_i) + (1-\alpha) d'_\omega(\{v_j\}, V_i)
\end{aligned}$$

This means that, when contracting the vertices $v_j, \ldots, v_n$ into a new vertex v^*, we obtain a contracted hypergraph for which Theorem 4.1 holds. Note that we need to replace n with $j+1$ and L with U to be able to apply the theorem. It follows that v_{j-1} can be safely added to the set of eligible vertices U of the current iteration. □



(i): Theorem is not satisfied, we cannot add 3 to U

(ii): Theorem is satisfied, we can safely add 3 to U

Figure 4.3: Example of Theorem 4.2 with $j = 4$ and $U = \{4, 5\}$. On the left, the theorem is not satisfied, i.e., the tight ordering changes after contracting U to v^* and vertex $j-1 = 3$ cannot be added to U. On the right, it is satisfied, i.e., we can safely add vertex 3 to U.

For the MA ordering, the recurrent contraction rule is defined in Theorem 4.3. Note that the MA ordering only uses the adjacency contribution. Therefore, when contracting the vertices of U into a new vertex v^*, it is sufficient to ensure that the adjacency contribution of v^* is the same as that of v_j for all partial MA orderings up to V_{j-3}. Otherwise, V_{j-2} may not be preserved, as it is based on all previous partial orderings. For this, the condition of Theorem 4.3 requires that every hyperedge incident to a vertex in $\{v_{j+1}, \ldots, v_n\}$ is either incident to v_j or not incident to any vertex in $\{v_1, \ldots, v_{j-3}\}$. In other words, the condition guarantees that there is no hyperedge $e \in I(U)$ in H that is incident to V_{j-3} but not to v_j. In this case, v_{j-1} can be safely added to the set of eligible vertices U. Otherwise, we have $d_\omega(U, V_{j-3}) = d_\omega(\{v_j\}, V_{j-3}) + \omega(e)$, meaning that v^* may be chosen earlier in the ordering, i.e., the partial MA ordering V_{j-2} may not be preserved, as shown in Figure 4.4.

Theorem 4.3
Let $(v_1, \ldots, v_n)$ be a maximum adjacency ordering and U be the set of eligible vertices of an iteration of the ordering-based algorithm. We can safely add v_{j-1} to U if the following condition holds:

$$I(\{v_{j+1}, \ldots, v_n\}) \subseteq I(v_j) \cup (E \setminus I(v_1, \ldots, v_{j-3}))$$

Proof of Theorem 4.3 *For all $1 \leq i < j-2$, we see that a hyperedge cannot be included at the same time in $E \setminus I(v_1, \ldots, v_{j-3})$ and in $I(V_i)$. With the given condition, we conclude that $\forall\, e \in I(\{v_{j+1}, \ldots, v_n\}) : e \in I(v_j) \vee e \notin I(V_i)$. Since we have $\alpha = 1$ for the maximum adjacency ordering, it follows that for all $1 \leq i < j-2$:*

$$\begin{aligned}
\alpha d_\omega(U, V_i) + (1-\alpha) d'_\omega(U, V_i) &= d_\omega(U, V_i) \\
&= \sum_{e \in I(U) \cap I(V_i)} \omega(e) \\
&= \sum_{e \in I(v_j) \cap I(V_i)} \omega(e) \\
&= d_\omega(\{v_j\}, V_i) \\
&= \alpha d_\omega(\{v_j\}, V_i) + (1-\alpha) d'_\omega(\{v_j\}, V_i)
\end{aligned}$$

(i): Theorem is not satisfied, we cannot add 3 to U **(ii):** Theorem is satisfied, we can safely add 3 to U

Figure 4.4: Example of Theorem 4.3 with $j = 4$ and $U = \{4, 5\}$. On the left, the theorem is not satisfied, i.e., the MA ordering changes after contracting U to v^* and vertex $j - 1 = 3$ cannot be added to U. On the right, it is satisfied, i.e., we can safely add vertex 3 to U.

Similar to Theorem 4.2, this means that we can apply Theorem 4.1 to the hypergraph obtained by contracting the vertices $v_j, \ldots, v_n$ into a new vertex v^*. Thus, v_{j-1} can be safely added to the contracted vertices of the current iteration. □

For every other α-ordering, i.e., $0 < \alpha < 1$, the containment contribution as well as the adjacency contribution influence the ordering. Therefore, the vertex v_{j-1} can only be added to the set of eligible vertices U if both conditions of Theorem 4.2 and Theorem 4.3 are satisfied at the same time. Note that this is much more restrictive, which means that the tight ordering and the MA ordering are expected to be better suited for applying multiple contractions per iteration.

4.3.4 Pseudocode

The pseudocode of HeiCut is provided in Algorithm 1. The boolean $useLP$ indicates whether the algorithm uses label propagation to perform the heuristic reduction in every round, as described in Section 4.3.2. For each reduction rule, we use a separate *union-find* data structure [28] to efficiently keep track of which vertices must be contracted together. Tarjan [74] showed that, when improving the union-find data structure with techniques such as path compression and union by rank, each operation runs in almost constant time. More specifically, an operation uses $O(\alpha(n))$ amortized time where $\alpha(n)$ is the inverse Ackermann function [1]. To determine the minimum cut of the kernel, HeiCut uses one of the variants of the exact ordering-based algorithm, for which we describe the pseudocode in Algorithm 2. In particular, the multiple contraction approach described in Section 4.3.3 is implemented by the lines 7 to 10. When excluding these lines, we obtain the original ordering-based algorithm with a single contraction per iteration, as defined in previous work [15, 52, 56, 66]. For the computation of the α-ordering, we use a priority queue to store and update the adjacency and containment contribution of each vertex with respect to the already-ordered vertices. In particular, we use a bucket priority queue if the hypergraph is unweighted and a binary heap if the hypergraph is weighted.

Algorithm 1: HEICUT

Input: Hypergraph $H = (V, E)$, parameter $\alpha \in [0, 1]$, boolean $useLP$
Output: Minimum cut value $\lambda(H)$

$\hat{\lambda}(H) \leftarrow \delta_\omega(H)$
while *first round or reduced in previous round* **do** `// Kernelization rounds`
  if $useLP$ **then** `// Optional heuristic`
    $H \leftarrow$ LABELPROPAGATION(H)
  foreach *exact reduction rule* r **do** `// Fixed order of rules`
    $H \leftarrow$ APPLYEXACTREDUCTION$(H, r, \hat{\lambda}(H))$
    if $|V| > 1$ **then**
      $\hat{\lambda}(H) \leftarrow \min(\hat{\lambda}(H), \delta_\omega(H))$ `// Update upper bound`
    if $|E| = 0 \vee |V| = 1 \vee \hat{\lambda}(H) = 0$ **then**
      return $\lambda(H) \leftarrow \hat{\lambda}(H)$ `// Stop early`
$\hat{\lambda}(H) \leftarrow \min(\hat{\lambda}(H),$ ORDERINGBASEDSOLVER$(H, \alpha))$ `// Solve kernel`
return $\lambda(H) \leftarrow \hat{\lambda}(H)$

Algorithm 2: ORDERINGBASEDSOLVER

Input: Hypergraph $H = (V, E)$, parameter $\alpha \in [0, 1]$
Output: Minimum cut value $\lambda(H)$

$n \leftarrow |V|$
while $n > 1$ **do** `// Stop if one vertex remains`
  $(v_1, \ldots, v_n) \leftarrow$ COMPUTEORDERING(H, α)
  $\hat{\lambda}(H) \leftarrow \min(\hat{\lambda}(H), \lambda[\{v_n\}])$ `// Update upper bound`
  $U \leftarrow \{v_{n-1}, v_n\}$
  $j \leftarrow n - 1$
  while $j \geq 2$ *and can add* v_{j-1} *to* U **do** `// Use Theorem 4.2 and/or 4.3`
    $\hat{\lambda}(H) \leftarrow \min(\hat{\lambda}(H), \lambda[U], \lambda[\{v_{j-1}\}])$ `// Update upper bound`
    $U \leftarrow \{v_{j-1}\} \cup U$
    $j \leftarrow j - 1$
  $H \leftarrow$ CONTRACT(H, U) `// Contract eligible vertices`
  $n \leftarrow n - |U| + 1$
return $\lambda(H) \leftarrow \hat{\lambda}(H)$

4.4 Parallelization

In this section, we describe how our proposed algorithms can be parallelized in a shared-memory context. This involves running multiple threads in parallel on the same machine, with all threads sharing a global memory space. More specifically, we only focus on the parallelization of HEICUT, given that there already exists extensive work on the shared-memory parallelization of BIP and MILP solvers [10, 24, 70]. First, we demonstrate that the kernelization phase of HEICUT, including the exact reduction rules and the label propagation heuristic, is *embarrassingly parallel*. This means that it can be parallelized with little effort. Afterwards, we demonstrate that the exact ordering-based algorithm can be parallelized by computing a different α-ordering for each thread. To the best of our knowledge, this is the first ever parallel implementation of the ordering-based algorithm.

4.4.1 Parallel Kernelization

The kernelization phase of HEICUT is embarrassingly parallel. The reason for this is that the exact reduction rules and the label propagation heuristic can both be parallelized with little effort, since almost no dependency or communication between the threads is required. Note that we will not describe how to parallelize the contraction step, since we use the method of MT-KAHYPAR [33] for this, which mainly uses easily parallelizable operations. For more information, we refer to Section 6.3.

We observe that each reduction rule in Section 4.3.1 first identifies all structures in the hypergraph to which the reduction rule applies before performing the contraction. This is because all of the identified structures are independent of each other and can thus be contracted simultaneously before applying the next reduction rule. Therefore, parallelizing each reduction rule is as simple as identifying the structures simultaneously rather than sequentially. In other words, instead of looping sequentially over every structure to check whether it satisfies the reduction rule, the different threads perform this check in parallel. The only common dependency of the threads is the union-find data structure, which keeps track of the vertices that must be contracted together. Similar to VIECUT, we avoid race conditions by using the parallel union-find data structure of Anderson et al. [4]. They propose a wait-free implementation that replaces critical operations, such as path compression and rank updates, with *compare-and-swap (CAS)* instructions.

Some of the graph-based reduction rules are parallelized differently compared to their original description in VIECUT. In particular, the authors of VIECUT claim that Reduction Rule 5 can only be parallelized if it is ensured that none of the identified edges share a common vertex. In other words, there should not exist a vertex $v \in V$ for which two incident edges e_{uv} and e_{vw} will be contracted by Reduction Rule 5. When contracting the two edges separately, the contraction of the edge e_{uv} merges the vertices u and v into a new vertex v^*, meaning that the edge e_{vw} becomes e_{v^*w}. They argue that the weighted vertex degree of v^* differs from that of v, which could invalidate Reduction Rule 5 for

the edge e_{v^*w}, even if it could previously be applied to the edge e_{vw}. However, we find this restriction to be unnecessary in our scenario, as the strict inequality mentioned in the proof of Reduction Rule 5 guarantees that this issue will never arise. For a more detailed derivation, we refer to Proof A.1. Besides, we allow each vertex to scan a slightly higher number of triangles for Reduction Rules 6 and 7 by handling marked vertices differently. In particular, we still mark the vertices and only scan the neighborhood of non-marked vertices. However, we now allow the neighbors to be already marked if they have not already scanned their own neighborhood themselves. In the sequential case, this change would lead to a non-linear time complexity. In the parallel scenario, it is a common practice to assign each thread a slightly higher workload due to the parallel execution. Note that we still enforce the restriction that each vertex can only participate in at most one contraction by using a CAS instruction. This modification of Reduction Rules 6 and 7 is already part of the parallel implementation of VIECUT, but it has never been explicitly mentioned in the papers [40, 41] to the best of our knowledge.

The simplest way to parallelize the label propagation heuristic is to determine the next label for multiple vertices simultaneously, rather than looping over the vertices one by one in each iteration. As this straightforward approach does not rely on any communication between the threads, race conditions might occur. This means that a vertex might update its label so quickly that neighboring vertices use this new label instead of that from the previous iteration to determine their own new label. However, Staudt and Meyerhenke [71] show that these race conditions do not affect the quality of the algorithm and even introduce another source of randomness, which may increase the clustering diversity. Therefore, we take the same approach as VIECUT and simply ignore the race conditions.

4.4.2 Parallel Ordering-Based Solver

A key observation for parallelizing the ordering-based algorithm is that Equation 3.1 can be applied to different vertex pairs of the hypergraph H at the same time. This means that we can compute, in each thread i, the minimum s_i-t_i cut value for a thread-specific pair $s_i, t_i \in V$. We then store $\min_i\{\lambda(H, s_i, t_i)\}$ if it is lower than the current upper bound $\hat{\lambda}(H)$ and contract each pair of vertices before starting the next iteration. As in the sequential case, the most straightforward way to implement the parallel algorithm is to make use of α-orderings. In particular, each thread computes its own α-ordering to obtain a thread-specific pendent pair $(v_{i,n-1}, v_{i,n})$ where $\{v_{i,n}\}$ is a minimum $v_{i,n-1}$-$v_{i,n}$ cut. Since each α-ordering starts at a random vertex, it is likely that $v_{i,n}$ is different for every thread, even if all threads use the same value α. However, there is no guarantee that the threads will compute different pendent pairs. In the worst case, the performance will be the same as for the sequential algorithm. Therefore, to increase diversification, we can assign a different value α_i to each thread i. For example, when using a uniform distribution, we set $\alpha_i = \frac{i-1}{N-1}$ for $1 \leq i \leq N$. In this case, the first thread uses the tight ordering, the last thread uses the MA ordering and all other threads use a different combination of the adjacency and the containment contributions. Nevertheless, in the worst

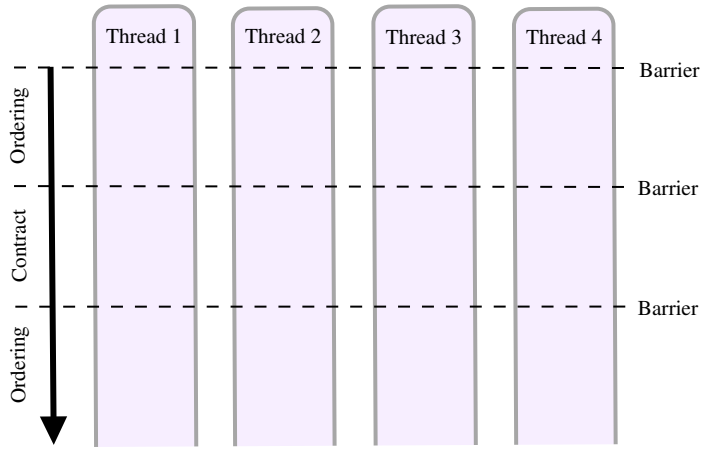


Figure 4.5: A timeline diagram for the execution of the ordering-based algorithm with $N = 4$ threads. For every iteration, two separate barriers ensure that the threads are synchronized before computing the α-ordering or contracting the pendent pair in each thread.

case, the algorithm still performs identically to the sequential implementation. In the best case, each thread would compute a different pendent pair in each iteration, meaning that the algorithm would only require $\lceil \frac{n-1}{N} \rceil$ iterations. To the best of our knowledge, this approach for parallelizing the ordering-based algorithm has never been suggested in any published work before. Note that the technique of performing multiple contractions per iteration as described in Section 4.3.3 also works in the parallel scenario. More specifically, each thread can independently search for multiple contractions based on its α-ordering, leading to an even smaller number of iterations.

One way to implement the parallel ordering-based algorithm is to use N persistent threads. This means that we initialize the threads at the start of the algorithm and only destroy them after having performed all iterations. The reason for this is that we want to use the maximum degree of parallelization in each iteration, which is exactly the permitted number N of threads. Since N remains the same for every iteration, re-initializing the threads in each iteration would only lead to more overhead. In addition, when each thread updates the current upper bound $\hat{\lambda}(H)$ based on its pendent pair, we use a CAS instruction to avoid race conditions. An important aspect of parallelizing the ordering-based algorithm is that all threads must be synchronized. The reason for this is that all threads must finish the computation of their α-ordering before any thread can contract its pendent pair. Otherwise, a contraction could invalidate the α-orderings of other threads, resulting in incorrect behavior. Similarly, each thread must wait for all the other threads to have finished contracting their pendent pair before it can compute the next α-ordering. Both synchronizations are achieved by inserting a barrier, as shown in Figure 4.5. When a thread reaches a barrier, it waits until all other threads have also reached the barrier. The first barrier would be positioned after line 10 of Algorithm 2 and the second after line 12.

CHAPTER 5

Hypercactus

In the previous chapter, we described in detail how to efficiently find a minimum cut in a hypergraph H. However, in general, a hypergraph can have many cuts of the same value. Therefore, a natural extension of the hypergraph minimum cut problem is to search for *all* minimum cuts in H, rather than just one. More specifically, the goal is to construct a so-called *hypercactus* H^*, i.e., a compact representation of size $O(n)$ that preserves all minimum cuts of H. Chekuri and Xu [14, 15] show that computing a hypercactus H^* of the hypergraph H can be achieved in the same time complexity as that for finding a single minimum cut by using a so-called split oracle. In this chapter, we will provide several improvements to the algorithm of Chekuri and Xu as well as its first ever implementation, to the best of our knowledge. For this, we first outline the idea behind the algorithm of Chekuri and Xu in Section 5.1. Afterwards, we demonstrate in Section 5.2 that we can improve the hypercactus algorithm by performing kernelization in a preprocessing phase, similar to HEICUT. In Section 5.3, we describe several modifications to the split oracle of Chekuri and Xu that facilitate the implementation and improve the performance in practice. The pseudocode for the hypercactus algorithm, including the suggested improvements, is provided in Section 5.4.

5.1 Algorithm Description

We briefly outline the hypercactus algorithm of Chekuri and Xu to provide context for the improvements described in subsequent sections. The algorithm exploits the fact that each hypergraph H admits a so-called unique canonical decomposition D_H^c [19] that can be used to construct a hypercactus representation H^*, which is not necessarily unique [17]. For more information, we refer to Section 2.2.2. Chekuri and Xu describe in detail how the hypercactus H^* can be constructed from the canonical decomposition D_H^c of H. More specifically, each element of D_H^c is first transformed according to a fixed scheme, after which H^* is built by connecting all of the transformed elements together. As it is computationally intensive to directly compute the canonical decomposition of H, Chekuri and

Xu propose to first compute a prime decomposition of H. A prime decomposition is not necessarily unique and consists only of prime elements that do not contain any split, i.e., no prime element admits a non-trivial minimum cut. For example, an element with fewer than four vertices is always prime, since every minimum cut is guaranteed to be trivial. Next, the prime decomposition is transformed into the canonical decomposition by merging some of the prime elements. One straightforward way to compute a prime decomposition of H is to start with the decomposition $\{H\}$ and repeatedly eliminate a split from an element in the decomposition. This can be done by using a so-called *split oracle* in a recursive manner.

Definition 5.1 (Split Oracle)
A split oracle is a function that, given an element $Y = (V_Y, E_Y)$ in the decomposition of the input hypergraph H, outputs either a split in Y or two vertices $s, t \in V_Y$ such that there is no s-t split with value $\lambda(H)$ in Y.

In particular, if the split oracle outputs a split for a given element Y in the decomposition, we replace Y with the two elements obtained by cutting along the split. Otherwise, we reduce the size of the element by contracting the vertices s and t. Although coming up with any polynomial-time algorithm for the split oracle is simple, the near-linear time approach of Chekuri and Xu involves a series of advanced techniques and concepts. The overall idea of their algorithm is based on the digraph $\vec{G}_Y$, which we defined in Section 2.1. More specifically, any minimum s-t cut in Y corresponds to a minimum s-t cut in the associated digraph $\vec{G}_Y$ and can thus be computed from a maximum s-t flow in $\vec{G}_Y$ [54]. This means that, for two vertices $s, t \in V_Y$, there exists an s-t split in Y if we can enumerate at least three minimum s-t cuts from the maximum s-t flow of $\vec{G}_Y$. The reason for this is that $\{s\}$ and $\{t\}$ are the only two trivial minimum s-t cuts, which means that the third minimum cut must be an s-t split in Y. Note that if only one or two minimum s-t cuts can be enumerated, we must check individually whether they are non-trivial. In the case where no s-t split can be found or its value is greater than $\lambda(H)$, the oracle returns s and t, as the two vertices can safely be contracted. To compute the maximum s-t flow in the digraph $\vec{G}_Y$, Chekuri and Xu use an approach that first computes a tight ordering in Y. Afterwards, they construct a so-called *tight graph* $G'_Y = (V', E')$ where $V' = V_Y$ and each hyperedge $e \in E_Y$ is replaced by an edge $e' \in E'$ that connects the last two pins of e in the tight ordering. An example of a tight graph is given in Figure 5.1. Chekuri and Xu show that the containment contribution is entirely preserved, i.e., every tight ordering in Y is also a tight ordering in G'_Y. This means that $\{v_n\}$ is a trivial minimum v_{n-1}-v_n cut in both Y and G'_Y. It follows that the maximum v_{n-1}-v_n flow value in G'_Y is the same as that in $\vec{G}_Y$. Therefore, a maximum v_{n-1}-v_n flow in G'_Y can be easily transformed into a maximum v_{n-1}-v_n flow in $\vec{G}_Y$, i.e., we set $s = v_{n-1}$ and $t = v_n$. The only remaining task is to compute the maximum v_{n-1}-v_n flow in the tight graph G'_Y, which can be achieved by applying the algorithm of Arikati and Melhorn [5]. Note that, even though this algorithm requires an MA ordering as an input, we can substitute a tight ordering because all α-orderings collapse to the same vertex ordering on graphs.

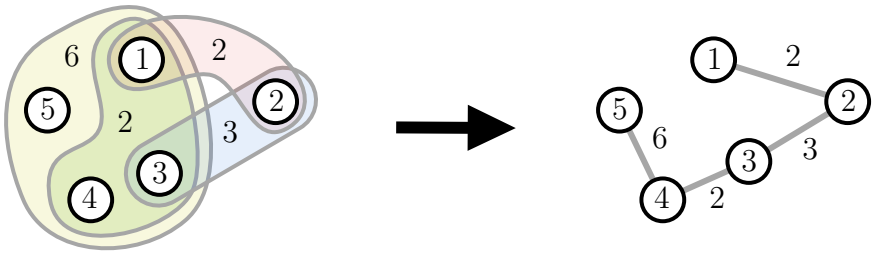


Figure 5.1: Example of building a tight graph from a hypergraph based on a specific tight ordering.

5.2 Kernelization

When taking a closer look at Definition 5.1, we observe that the vertices s and t of Y represent sets of vertices $S, T \subseteq V$ in the input hypergraph H. More specifically, we know that there is no split in H going through S or T, due to recursive application of the split oracle. Furthermore, if there is no s-t split with value $\lambda(H)$ in Y, the vertices s and t are contracted into a new vertex u and we know that there is no split in H going through $U := S \cup T$. This approach is very similar to the concept of the reduction rules defined in Section 4.3.1, since there we also identify structures in H for which it is proven that no minimum cut is going through them. Therefore, it is possible to improve the hypercactus algorithm by performing kernelization in a preprocessing phase before applying the split oracle to find the prime decomposition of the kernel. Afterwards, the prime decomposition of the input hypergraph H can be easily constructed from that of the kernel by uncontracting the vertices that were merged together during the kernelization.

An important detail to consider when computing the prime decomposition of the kernel is that some vertices of the kernel originate from the contraction of multiple vertices in the input hypergraph H. In other words, we already have contracted vertices even before applying the split oracle for the first time. The problem with this is that a split in the input hypergraph might become trivial in the kernel and will thus not be detected by the split oracle. To solve this issue, we perform an initial scan when starting to compute the prime decomposition of the kernel before applying the split oracle. In particular, we first identify the vertices in the kernel that represent multiple vertices in H. For each of these vertices, we check if isolating it results in a split in H, and if so, we cut the vertex out and put it into its own decomposition element. Afterwards, we apply the split oracle as usual. This additional scan ensures that, when uncontracting the vertices that were merged together during kernelization, we obtain a valid prime decomposition of the input hypergraph H.

As the purpose of a hypercactus is to represent *all* minimum cuts in H and not only the minimum cut value $\lambda(H)$, we need to slightly modify the reduction rules of Section 4.3.1. In particular, we must weaken the reduction rules to ensure that they do not destroy any minimum cut. For this, we first enforce a strict inequality in the estimate-based Reduction Rules 2, 3 and 7, as otherwise we would destroy minimum cuts if $\hat{\lambda}(H) = \lambda(H)$. The strict inequality guarantees that we only contract hyperedges that never cross a cut with

a value of $\hat{\lambda}(H)$ and therefore they never cross a minimum cut. Besides, we also require strict inequalities for Reduction Rule 6, as otherwise we could destroy a minimum cut by replacing it with another. In other words, we only want to replace a cut if we find another one with a strictly lower value. Note that this also applies to Reduction Rule 5 but it already uses a strict inequality due to an issue explained in Figure 4.2. Finally, we must add a constraint to Reduction Rules 5 and 6 so that the weighted vertex degree of both endpoints u and v of the contracted hyperedge e_{uv} is strictly larger than the current estimate, i.e., $d_\omega(u) > \hat{\lambda}(H)$ and $d_\omega(v) > \hat{\lambda}(H)$. The reason for this is that contracting e_{uv} could destroy a trivial minimum cut if the current estimate is already equal to the minimum cut value, i.e., $\hat{\lambda}(H) = \lambda(H)$. If we have for example $d_\omega(u) = \hat{\lambda}(H) = \lambda(H)$, then $\{u\}$ is a trivial minimum cut and it would be destroyed when e_{uv} satisfies Reduction Rule 5 or 6. The additional constraint assures that e_{uv} is not contracted if one of its endpoints forms a trivial minimum cut. Note that Reduction Rules 1 and 4 already preserve all minimum cuts and therefore do not require any modification. Furthermore, we disable the label propagation heuristic described in Section 4.3.2, as the goal of the kernelization phase is to preserve all minimum cuts, i.e., every reduction must be exact.

5.3 Improved Split Oracle

In the following, we describe several modifications to the split oracle of Chekuri and Xu that facilitate the implementation and improve the performance in practice. First of all, an important aspect of the split oracle is that when it finds two vertices s and t for which there is no s-t split, we can only contract them if the cut $\{s, t\}$ isolating the two vertices is not a split. This is because, after the contraction of s and t, this split would become trivial and would thus not be detected by the split oracle anymore. Chekuri and Xu handle this case via backtracking, i.e., they always contract s and t and then uncontract them when moving up the recursion stack to determine whether $\{s, t\}$ is a split or not. If it is, they put s and t into their own decomposition element. However, this approach is likely to be computationally expensive, as it involves uncontracting the vertices at a later stage. We propose a much simpler and more efficient approach by performing forward checking, i.e., we only contract the vertices if $\{s, t\}$ is not a split. Otherwise, they are directly put into their own decomposition element. More formally, we only contract s and t if $\lambda[\{s, t\}] > \lambda(H)$. Note that computing the value of $\{s, t\}$ can be achieved by starting with the sum of the weighted vertex degrees of s and t and properly handling all of the hyperedges that are incident to both vertices.

The core idea of the algorithm of Chekuri and Xu is to use a maximum s-t flow in the digraph $\vec{G}_Y$ to determine whether an element Y in the decomposition admits an s-t split, as outlined in Section 5.1. For this, they rely on the enumeration algorithm of Provan and Shier [64]. More specifically, they argue that there exists an s-t split in Y if we can enumerate at least three minimum s-t cuts from the maximum s-t flow of $\vec{G}_Y$. Although this approach is theoretically sound, implementing the complex enumeration algorithm of

Provan and Shier only to stop already at the third iteration introduces unnecessary complexity and goes beyond what is required in practice. In the following, we will describe an alternative approach that achieves the same goal while relying on simple techniques that can easily be implemented. Our approach is inspired by the most balanced minimum cuts heuristic of Sanders and Schulz [68], which is based on the work of Picard and Queyranne [63]. We first compute the residual graph $G_f = (V_f, E_f)$ of the digraph $\vec{G}_Y$ as described in Section 2.2.3. Picard and Queyranne show that every minimum s-t cut of $\vec{G}_Y$ is represented by a closed vertex set C containing s but not t in G_f. This leads us to the following theorem, for which we first introduce the notion of *original* vertices.

Definition 5.2 (Original Vertex)
For a hypergraph $H = (V, E)$ and its digraph $\vec{G}_H = (\vec{V}, \vec{E})$ with $\vec{V} := V \cup E^+ \cup E^-$, we call a vertex $v \in \vec{V}$ original if it is also included in H, i.e., $v \in V$. For a given set $A \subseteq \vec{V}$, we denote $o(A) \subseteq A$ as the subset of original vertices in A.

Theorem 5.1
For a given element $Y = (V_Y, E_Y)$ in the decomposition of the input hypergraph H, there exists an s-t split in Y if the residual graph G_f of the digraph $\vec{G}_Y$ admits a closed vertex set C that contains s but not t and for which $2 \leq |o(C)| \leq V_Y - 2$.

Proof of Theorem 5.1 *From Picard and Queyranne [63], we know that $(C, \vec{V}_Y \setminus C)$ is a minimum s-t cut in the digraph $\vec{G}_Y$ if and only if C is a closed vertex set that contains s but not t in the associated residual graph G_f. However, if $(C, \vec{V}_Y \setminus C)$ is a minimum s-t cut in $\vec{G}_Y$, then $(o(C), V_Y \setminus o(C))$ is a minimum s-t cut in Y [54]. If $2 \leq |o(C)| \leq V_Y - 2$, then $(o(C), V_Y \setminus o(C))$ is non-trivial and therefore an s-t split in Y.* □

Based on Theorem 5.1, we observe that, instead of enumerating the minimum s-t cuts in the digraph $\vec{G}_Y$, we can enumerate the closed vertex sets in the residual graph G_f to determine the existence of an s-t split in Y. One way to achieve this is to apply the dynamic programming algorithm of Schrage and Baker [69] to the vertices of G_f. For this, the residual graph G_f must be *acyclic*, i.e., it must not contain any cycles. A common way to achieve this is to contract the so-called *strongly connected components* of G_f.

Definition 5.3 (Strongly Connected Component)
A strongly connected component (SCC) of a directed graph G is a maximal subgraph where, for any two vertices u and v in the subgraph, there exists a path from u to v and a path from v to u within the subgraph. This means that u is reachable from v and vice-versa.

Picard and Queyranne show that an SCC can never partially overlap with a closed vertex set C, i.e., either all vertices of the SCC are included in C or none of them are [63]. Thus, we can safely contract the strongly connected components of G_f without destroying any closed vertex set. We use the path-based algorithm [18, 21, 27] to identify the strongly connected components of G_f in linear time by performing a single *depth-first search* (DFS).

So far, we have demonstrated that we can implement the split oracle by using the enumeration algorithm of Schrage and Baker rather than that of Provan and Shier. Although the former algorithm is easier to implement, it is still unnecessarily complex in our scenario, as we stop the enumeration process after the third iteration anyway. In particular, we observe that the algorithm of Schrage and Baker can be omitted in most cases by performing a series of *quick checks*. This is because we only need to find *any* s-t split in Y if it exists, which is a much weaker requirement than enumerating all minimum s-t cuts in Y. For this, we take a similar approach as Sanders and Schulz [68], i.e., we sort the contracted SCCs in reversed topological order. This means that all contracted SCCs are sorted such that, for every directed edge (u, v) in the contracted residual graph, the SCC v comes before the SCC u in the ordering. In other words, an SCC is preceded in the ordering by all its successors in the contracted residual graph. Note that a reversed topological ordering of the SCCs is implicitly computed when identifying the strongly connected components via DFS. The idea is that each cut in the ordering corresponds to a closed vertex set in the contracted residual graph, since every edge crossing the cut goes in the same direction. However, imposing a specific ordering on the SCCs and then separating them with a cut only preserves some of the closed vertex sets but not all of them. This is the reason why the most balanced minimum cut algorithm of Sanders and Schulz is heuristic. In our scenario, this is not an issue, since we will revert to the algorithm of Schrage and Baker if all quick checks fail. As we are only interested in closed vertex sets that separate s from t, we first prove in Theorem 5.2 that the SCC of the sink t can always be moved to the last position of the reversed topological ordering. Afterwards, we show in Theorem 5.3 that every SCC of the contracted residual graph contains at least one original vertex from Y.

Theorem 5.2
For a given reversed topological ordering of the contracted SCCs, the SCC of the sink t can safely be moved to the last position without destroying the reversed topological ordering.

Proof of Theorem 5.2 *We know that t is the last vertex in the tight ordering of the vertices of Y. This means that $\{t\}$ is a minimum s-t cut in Y. It follows that, for all incident hyperedges $e \in I(t)$ in Y, the corresponding edge (e^-, e^+) is saturated in the digraph $\vec{G}_Y$. Therefore, the SCC of the sink t has no incoming edge in the contracted residual graph and can be moved to the last position without destroying the reversed topological ordering.* □

Theorem 5.3
For the residual graph G_f of the digraph $\vec{G}_Y$, every SCC of G_f contains at least one original vertex, i.e., there exists at least one vertex $v \in V_Y$ that is part of the SCC.

Proof of Theorem 5.3 *We will prove this theorem by contradiction. Assume that there is an SCC of G_f that does not contain an original vertex. Then it must contain a vertex from $E^- \cup E^+$, since the SCC cannot be empty. This gives us two cases.*
Case 1: The SCC contains a vertex $e^- \in E^-$. Now e^- either receives some flow from a vertex $v \in V_Y$ or not. If it does, both edges (v, e^-) and (e^-, v) are present in G_f and there

is a cycle $e^- \to v \to e^-$ in G_f. If it does not, the edge $(e^-, e+)$ exists in G_f, i.e., there is a cycle $e^- \to e^+ \to v \to e^-$ in G_f. Thus, the original vertex v must be part of the SCC.
Case 2: The SCC contains a vertex $e^+ \in E^+$. Now e^+ either gives some flow to a vertex $v \in V_Y$ or not. If it does, both edges (v, e^+) and (e^+, v) are present in G_f and there is a cycle $e^+ \to v \to e^+$ in G_f. If it does not, the edge $(e^-, e+)$ exists in G_f, i.e., there is a cycle $e^+ \to v \to e^- \to e^+$ in G_f. Thus, the original vertex v must be part of the SCC. □

Based on Theorem 5.2, we always move the SCC of the sink t to the last position of the reversed topological ordering obtained from the DFS. Note that an implicit conclusion of the theorem is that the SCC of the sink t contains exactly one original vertex, i.e., the sink t itself. Afterwards, we apply four different quick checks that are applied in the order in which they are presented. If one of these quick checks is successful, we can output the result of the split oracle directly and do not need to run the enumeration algorithm of Schrage and Baker. This significantly speeds up the implementation in practice, as all quick checks can be applied in constant time. Figure 5.2 gives an illustration for all of the four different quick checks.

Quick Check 1. *If the SCC of the source s contains at least two original vertices and it does not come just before the SCC of the sink t in the ordering, cutting after the SCC of the source s in the ordering yields an s-t split in Y.*

Proof: *Every cut in the ordering corresponds to a closed vertex set C in G_f. Cutting after the SCC of the source s means that C contains s but not t, i.e., we have a minimum s-t cut in Y. As the SCC of the source s does not come just before the SCC of the sink t in the ordering, there are at least two SCCs in the partition of the sink t. With Theorem 5.3, it follows that the partition of the sink t contains at least two original vertices. Given that the partition of s contains at least two original vertices, Theorem 5.1 is satisfied and we have an s-t split in Y.* □

Quick Check 2. *If the SCC of the source s is not at the first position in the ordering and it does not come just before the SCC of the sink t in the ordering, cutting after the SCC of the source s in the ordering yields an s-t split in Y.*

Proof: *Same proof as for Quick Check 1. The only difference is that we now also apply Theorem 5.3 to the partition of the source s, i.e., there are at least two SCCs in the partition of the source s and therefore at least two original vertices.* □

Quick Check 3. *If there are at least two other SCCs between the SCC of the source s and the SCC of the sink t in the ordering, cutting after the SCC that follows the SCC of the source s in the ordering yields an s-t split in Y.*

Proof: *Same proof as for Quick Check 2. The only difference is that we do not cut directly after the SCC of the source s but one position later. Thus, there are still at least two SCCs in each partition, i.e., we can apply Theorem 5.3 to both partitions.* □

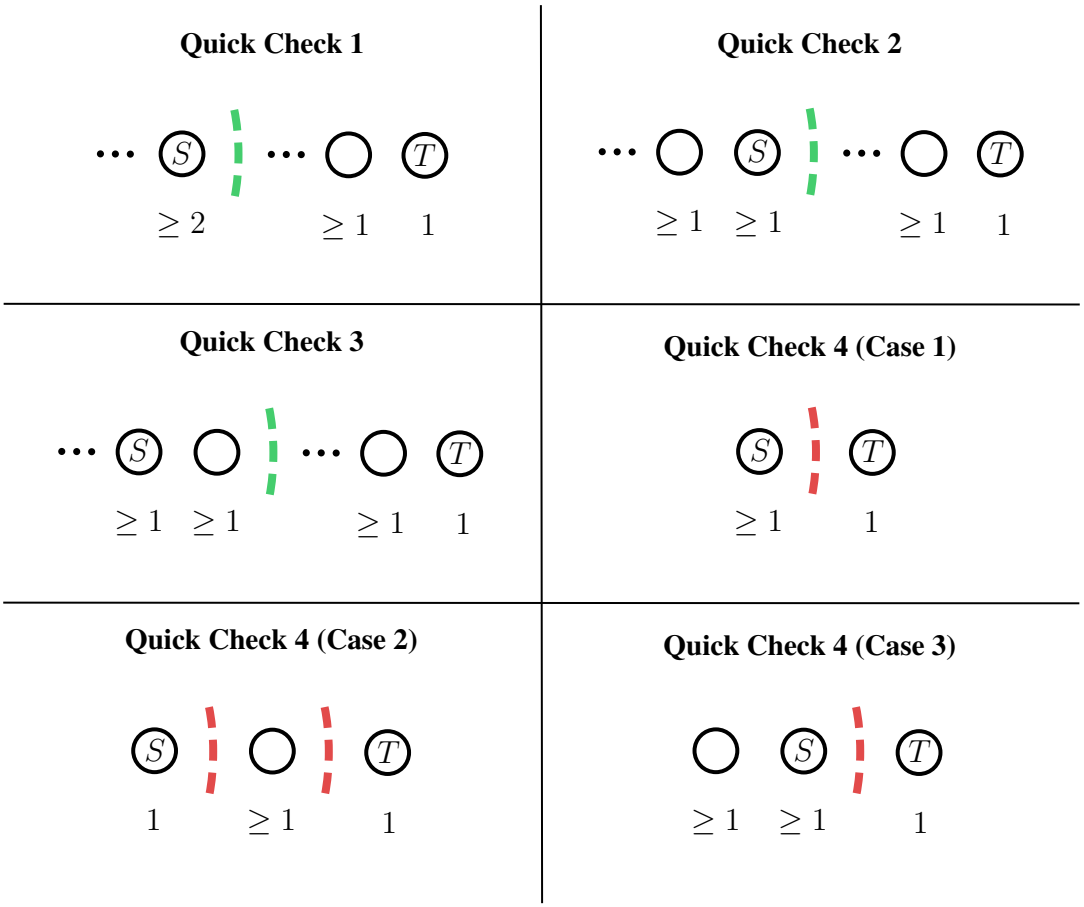


Figure 5.2: Illustration for all four quick checks, where S and T represent the respective SCCs of the source s and the sink t. For each SCC, we indicate the number of original vertices.

Quick Check 4. *If there are strictly less than four SCCs in the residual graph G_f, there does not exist an s-t split in Y.*

Proof: *There only exist three different cases for the ordering.*
Case 1: The ordering has a size of two, i.e., the SCC of the source s is followed by the SCC of the sink t. As the SCC of the sink t only contains one original vertex, the only possible cut is trivial, i.e., there is no s-t split in Y.
Case 2: There is a third SCC between the SCC of the source s and the SCC of the sink t. As Quick Check 1 failed, we know that the SCC of the source s contains only one original vertex, i.e., the source s itself. Besides, the SCC of the sink t also contains exactly one original vertex. Both possible cuts in the ordering are trivial, i.e., there is no s-t split in Y.
Case 3: There is a third SCC at the first position in the ordering, i.e., before the SCC of the source s. To separate s from t, we must cut after the SCC of the source s. However, we know that the SCC of the sink t contains exactly one original vertex. Therefore, the only possible cut in the ordering is trivial, i.e., there is no s-t split in Y. □

If none of the quick checks are successful, we still need to apply the enumeration algorithm of Schrage and Baker. However, we can make significant simplifications to the algorithm based on insights gained from the quick checks. In particular, we know from Quick Check 4 that the ordering contains at least four SCCs. Together with Quick Checks 2 and 3, we infer that, if all quick checks failed, the SCC of the source s must come just before the SCC of the sink t in the ordering. In other words, the SCC of the source s is at the second-to-last position in the ordering. As we are only interested in closed vertex sets that contain s but not t, we can ignore most iterations of the algorithm of Schrage and Baker. More specifically, since the algorithm relies on dynamic programming, the first closed vertex set C containing s but not t that is considered by the algorithm is the one that includes every SCC except the SCC of the sink t. We already know that this closed vertex set represents a trivial minimum s-t cut in Y, since the SCC of the sink t contains only one original vertex. Afterwards, the algorithm checks if it can remove any SCC from C so that the result is still a valid closed vertex set containing s but not t. In other words, it searches for an SCC $v \in C$ so that there is no directed edge (u, v) in the contracted residual graph with $u \in C \setminus \{v\}$. Note that v cannot be the SCC of the source s because otherwise the closed vertex set obtained by removing v from C does not contain s anymore. If any $v \in C$ can be found, we know from Theorem 5.3 that the resulting closed vertex set represents an s-t split in Y. Otherwise, we can conclude that there is no s-t split in Y. This means that the algorithm of Schrage and Baker can be simplified to a single pass over the SCCs in C, which can be implemented in linear time. Overall, we propose a much simpler approach for determining whether the element Y in the decomposition admits an s-t split based on a maximum s-t flow in the digraph $\vec{G}_Y$.

5.4 Pseudocode

To the best of our knowledge, we propose the first implementation of the hypercactus algorithm proposed by Chekuri and Xu. The pseudocode is provided in Algorithms 3, 4 and 5, which include all improvements described in Sections 5.2 and 5.3. The high-level overview of the implementation is given by Algorithm 3. More specifically, we first use HEICUT to compute the minimum cut value of the input hypergraph H. Afterwards, we reduce H to a smaller kernel by applying the weak definitions of the exact reduction rules in the lines 3 to 7 of Algorithm 3. We store a mapping k_H from the vertices of the input hypergraph H to the vertices of the kernel, which is necessary for uncontracting the vertices later. If the kernel is disconnected, i.e., $\lambda(H) = 0$ and $|E| = 0$, we can stop and return the kernel itself, since it is a valid hypercactus representation of H. The LARGEVERTICES method in line 8 identifies the vertices in the kernel that represent multiple vertices in H and adds them to the queue Q. This queue is necessary to ensure the correctness of the prime decomposition of the kernel computed in line 9, as described in Section 5.2. We then construct the prime decomposition of H by using the vertex mapping k_H to uncontract the vertices that were merged together during the kernelization. In line 11, we transform the prime decompo-

sition of H into the unique canonical decomposition D_H^c of H by merging some of the prime elements, as outlined in Section 5.1. Note that we store every decomposition in a tree to reduce the space usage to $O(p)$, as proposed by Chekuri and Xu. In line 12 we construct the hypercactus representation H^* based on the canonical decomposition D_H^c. More specifically, each element of D_H^c is first transformed according to a fixed scheme, after which H^* is built by connecting all of the transformed elements together. For a more detailed explanation of the different steps, we refer to Section 5.1 and the original papers of Chekuri and Xu [14, 15].

Algorithm 4, provides a detailed pseudocode of the PRIME method. In the first five lines, we use the vertex queue Q to check whether isolating one of the vertices in Q leads to a split in H, as these splits will not be detected by the split oracle. Besides, we improve the PRIME method so that it does not rely on backtracking and only contracts s and t if $\{s, t\}$ is not a split, as described by the lines 12 to 14. The details of the SPLITORACLE method are provided in Algorithm 5, including the improvements described in Section 5.3. In particular, we construct the residual graph G_f of the digraph $\vec{G}_Y$ in line 8, after which we identify the SCCs of G_f and sort them in reversed topological order in line 9. We apply Theorem 5.2 in line 10, followed by the quick checks in the lines 11 to 18. The remaining lines implement the single iteration of the algorithm of Schrage and Baker.

Algorithm 3: HYPERCACTUS

Input: Hypergraph $H = (V, E)$, parameter $\alpha \in [0, 1]$
Output: Hypercactus H^*, vertex mapping $\phi_{H^*} : V \to V^*$

$k_H \leftarrow \mathrm{id}_V$ `// Store contraction mapping of kernel`
$\lambda(H) \leftarrow$ HEICUT$(H, \alpha, \text{false})$
while *first round or reduced in previous round* **do** `// Kernelization rounds`
  foreach *exact weak reduction rule* r **do** `// Fixed order of rules`
    $H, k_H \leftarrow$ APPLYEXACTWEAKREDUCTION$(H, r, \lambda(H))$
if $\lambda(H) = 0 \wedge |E| = 0$ **then**
  return H, k_H `// Stop early`
$Q \leftarrow$ LARGEVERTICES(H, k_H) `// Find large vertices of kernel`
$D_H \leftarrow$ PRIME$(H, \lambda(H), Q)$ `// Prime decomposition of kernel`
$D_H \leftarrow$ UNCONTRACT(D_H, k_H) `// Prime decomposition of input`
$D_H^c \leftarrow$ CANONICAL(D_H) `// Canonical decomposition of input`
$H^*, \phi_{H^*} \leftarrow$ BUILDHYPERCACTUS$(D_H^c, \lambda(H))$
return H^*, ϕ_{H^*}

Algorithm 4: PRIME

Input: Element $Y = (V_Y, E_Y)$, minimum cut value $\lambda(H)$, large vertices $Q \subseteq V_Y$
Output: Prime decomposition D_Y

while $|Q| > 0$ **do** `// Check for split by isolating large vertices`
  $v \leftarrow$ Remove top element of Q
  if $\lambda[\{v\}] = \lambda(H)$ **then**
    $\{Y_1, Y_2\} \leftarrow$ REFINE$(Y, \{v\})$ `// WLOG we have` $v \in V_{Y_2}$
    return PRIME$(Y_1, \lambda(H), Q) \cup$ PRIME$(Y_2, \lambda(H), \{\})$
if $|V_Y| < 4$ **then**
  return $\{Y\}$ `// Stop early`
$s, t, S \leftarrow$ SPLITORACLE$(Y, \lambda(H))$ `// Call split oracle on element`
if *split oracle finds split* S **then**
  $\{Y_1, Y_2\} \leftarrow$ REFINE(Y, S)
  return PRIME$(Y_1, \lambda(H), \{\}) \cup$ PRIME$(Y_2, \lambda(H), \{\})$
else if $\{s, t\}$ *is a split* **then** `// Check to avoid backtracking`
  $\{Y_1, Y_2\} \leftarrow$ REFINE$(Y, \{s, t\})$
  return PRIME$(Y_1, \lambda(H), \{\}) \cup$ PRIME$(Y_2, \lambda(H), \{\})$
return PRIME$(Y \,/\, \{s, t\}, \lambda(H), \{\})$ `// We can safely contract` s `and` t

Algorithm 5: SPLITORACLE

Input: Element $Y = (V_Y, E_Y)$, minimum cut value $\lambda(H)$
Output: Source s, sink t, optional split S

$n \leftarrow |V_Y|$
$(v_1, \ldots, v_n) \leftarrow$ TIGHTORDERING(Y)
$G'_Y \leftarrow$ TIGHTGRAPH$(Y, v_1, \ldots, v_n)$
$s \leftarrow v_{n-1} \wedge t \leftarrow v_n$
$f' \leftarrow$ MAXFLOW(G'_Y, s, t) `// Use Arikati and Melhorn [5]`
$\vec{G}_Y \leftarrow$ DIGRAPH(Y)
$f \leftarrow$ CONVERT$(\vec{G}_Y, f')$ `// Convert flow of` G'_Y `to flow of` $\vec{G}_Y$
$G_f \leftarrow$ RESIDUALGRAPH$(\vec{G}_Y, f)$
$\text{SCC}_1, \ldots, \text{SCC}_j \leftarrow$ REVERSEDTOPOLOGICALORDERING(G_f)
Move SCC of t to last position `// Apply Theorem 5.2`
if *Quick Check 1 or Quick Check 2 applies* **then**
    $S \leftarrow$ Cut after the SCC of s
    return s, t, S
else if *Quick Check 3 applies* **then**
    $S \leftarrow$ Cut after the SCC that follows the SCC of s
    return s, t, S
else if *Quick Check 4 applies* **then**
    return $s, t, \{\}$
$C \leftarrow$ All SCCs except the SCC of t
if *there is an SCC* $v \in C$ *with no predecessor in* C *and* v *is not the SCC of* s **then**
    $S \leftarrow$ Separate v and the SCC of t from the rest
    return s, t, S
return $s, t, \{\}$ `// No split could be found`

CHAPTER 6

Experimental Evaluation

In this chapter, we start by outlining the hardware of the machine on which we run our experiments in Section 6.1. Afterwards, we describe the instances used for our experiments in Section 6.2. In Section 6.3, we outline our methodology in detail, including a description of the state-of-the-art competitors and of the measured metrics. Finally, we perform the experiments and analyze their results in Section 6.4.

6.1 Hardware

Our experiments are performed on a machine containing an `AMD EPYC 7702P` CPU and 996 GB of available RAM. The CPU consists of 64 physical and 128 logical cores, since each core can handle two threads. The clock speed can reach a minimum of 1.5 GHz as well as a maximum of 2 GHz. The machine contains an L2-Cache of 64 MiB (0.5 MiB per thread) and is based on the `x86_64` architecture. It runs `Ubuntu 20.04.1 LTS` with the Linux kernel version `5.4.0-187-generic`.

6.2 Instances

For our experiments, we use two main datasets provided by Gottesbüren et al. [33]. The M_{HG} dataset consists of 488 medium-sized hypergraphs, ranging from a few hundred to 13 million hyperedges. The L_{HG} dataset consists of 94 large-sized hypergraphs with up to 139 million hyperedges. The datasets cover three application domains originating from four sources, namely the ISPD98 VLSI Circuit Benchmark Suite [2], the DAC 2012 Routability-Driven Placement Contest [76], the SuiteSparse Matrix Collection [20] and the International SAT Competition 2014 [8]. Besides, to test the algorithms specifically on hypergraphs where the minimum cut value is not equal to the smallest weighted vertex degree, we construct a new dataset of $(k, 2)$-core hypergraphs. These instances are harder to solve as they do not admit a trivial minimum cut and provide a benchmark for

future work on the hypergraph minimum cut problem. Inspired by VIECUT, we perform a $(k, 2)$-core decomposition on the L_{HG} dataset by repeatedly removing, for each hypergraph, all vertices with an unweighted degree less than k as well as all hyperedges with a size less than two. For each hypergraph $H \in L_{HG}$, we store the $(k, 2)$-core with the lowest possible $k \geq 2$ where $\lambda(H) < \delta_\omega$. Note that not every hypergraph admits a $(k, 2)$-core that satisfies this condition. In total, we obtain a dataset of 44 different $(k, 2)$-cores. All hypergraphs from all datasets were originally created in their unweighted version. For the weighted instances, we assign each vertex and each hyperedge uniformly at random a weight between 1 and 100. The weighted datasets are all publicly available [1].

6.3 Methodology

Since our main focus lies in finding the exact minimum cut value, we configure HEICUT to not use the label propagation heuristic of Section 4.3.2. However, to assess the performance of our algorithm when no optimality is guaranteed, we also test an inexact variant named HEICUTLP that uses a single label propagation iteration per kernelization round. For both variants, we use the tight ordering ($\alpha = 0$) for the underlying ordering-based solver, as it incurs the fewest `increaseKey` operations in the priority queue, making it particularly efficient in practice. Furthermore, both variants implement the multiple contractions technique described in Section 4.3.3.

To put the performances of HEICUT and HEICUTLP into perspective, we compare them with four different competitors: RELAXEDBIP, RELAXEDMILP, TIGHT and TRIMMER. The algorithms RELAXEDBIP and RELAXEDMILP respectively implement the BIP and the MILP formulations described in Sections 4.1 and 4.2. We refer to both as *relaxed*, since they use a rounding heuristic that does not guarantee exact solutions. In particular, both formulations are solved by *Gurobi* 11.0.3 [35] with `IntFeasTol` and `FeasibilityTol` set to 10^{-7}. The TIGHT competitor is the exact ordering-based algorithm with the tight ordering but without the multiple contractions, i.e., it is identical to the algorithm proposed Mak and Wong [56]. However, as the original implementation is not publicly available, we use our own implementation. The idea is that comparing HEICUT with TIGHT allows us to easily assess the benefits of adding the kernelization phase and the multiple contractions technique. The TRIMMER competitor is the state-of-the-art exact algorithm proposed by Chekuri and Xu [14, 15], which only works for unweighted hypergraphs. To ensure a fair comparison, we also use a tight ordering for the underlying ordering-based solver of TRIMMER. Since we could not find a public implementation of the algorithm of Chekuri and Xu, we implement TRIMMER based on the algorithmic description of the papers. The pseudocode of our implementation is provided in Section A.2 of the appendix. The code of all implemented algorithms is available in a public repository [2]. We describe the command-line arguments used for all algorithms in Section A.1 of the appendix.

[1] `https://doi.org/10.5281/zenodo.17142170`
[2] `https://github.com/HeiCut/HeiCut`

All algorithms are implemented in C++14 and compiled with g++ version `11.4.0` while using full optimization (`-O3`). The hypergraph data structure of all algorithms is the same as that designed for MT-KAHYPAR [33]. One reason for this is that MT-KAHYPAR comes with a sophisticated hypergraph contraction implementation, which we directly use in HEICUT and HEICUTLP for both the sequential and the parallel context. We require the input hypergraph to be in the HMETIS file format [50], which was originally created for the hypergraph partitioning software of the same name [51]. In addition, all algorithms support the graph-specific file format METIS [49], which was also originally created for the graph partitioning software of the same name [48]. When testing the parallel version of our algorithm as described in Section 4.4 we allow each instance of the experiment to use multiple threads. For this, we use *oneTBB* [3], since it is a fast threading library and it is already integrated in MT-KAHYPAR. To prevent conflicts between instances caused by simultaneous multithreading (SMT), we allocate a fixed range of physical cores to each instance. This means that all logical cores of the same physical core are only used by one instance at a time. As a pseudorandom number generator (PRNG), we use `MT19937`, which is the standard implementation of *Mersenne Twister* [57]. Note that we assign to each thread its own PRNG, as Mersenne Twister is not thread safe.

In an experiment, we measure, for every instance, the found cut value, the running time and the peak memory usage. For the running time, we exclude the I/O time and set for each instance a limit of two hours. To determine the peak memory usage, we use GNU time and extract the maximum resident set size, which is given in kilobytes. For the medium dataset M_{HG}, we use *GNU Parallel* [73] to run 13 independent instances in parallel on our machine, where each instance is given a hard virtual memory limit of 100 GB. For the large dataset L_{HG} and the $(k, 2)$-core dataset, we only run four instances in parallel with a virtual memory limit of 300 GB per instance. When testing the parallel versions of our algorithms, we ensure that we run only as many instances as the machine can accommodate. For example, if each instance uses 16 threads, we run only eight instances at a time for the medium dataset M_{HG}. Instances that exceed the time or memory limit before reporting a solution are considered to have failed. However, since we consider RELAXEDBIP and RELAXEDMILP to be inexact solvers, they are allowed to return the best feasible solution found so far if time runs out.

For the evaluation of the experiments, we plot most of the results in the form of a so-called *performance profile* [22]. This type of plot is widely used to compare different metrics such as the cut value, the running time or the peak memory usage. Each metric is visualized in a two-dimensional plot where the y-axis indicates a fraction of all the instances of an algorithm while the x-axis represents an increasing variable τ starting at one. The idea is to plot, for every algorithm A, a line indicating the fraction of instances for which the performance of A is within a factor τ of the best-performing algorithm on the same instance. More formally, every point (f, τ) on the plotted line of algorithm A indicates the fraction f of instances of A whose metric is smaller or equal to τ times the

[3] `https://github.com/uxlfoundation/oneTBB`

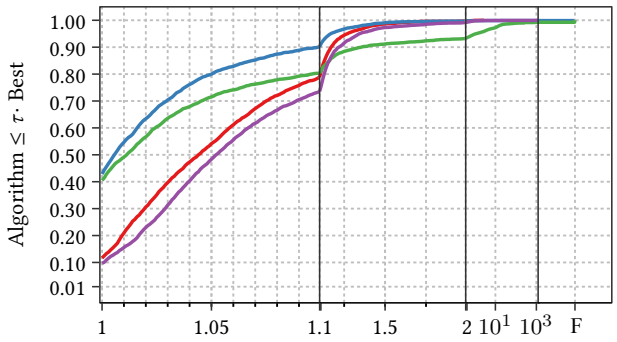


Figure 6.1: An example of a performance profile, which was taken from [13] and slightly modified. Four algorithms are plotted in different colors. The algorithm plotted in blue has the best performance for 40% of the instances. Around 90% of its instances are no more than 1.1 times worse than the best algorithm on a per-instance level.

metric of the best algorithm for the same instance. In other words, a point (f, τ) indicates that algorithm A is for a fraction f of its instances never more than τ times worse than the best algorithm on a per-instance level. An example of a performance plot is given by Figure 6.1. Performance profiles give a broader view on the results as they accurately depict comparisons between algorithms, even if some of them are unable to solve an instance. More specifically, failed instances simply enter into the failed (F) region of the plot.

6.4 Experiments

We run a series of experiments to evaluate the algorithms and improvements proposed in Chapters 4 and 5. More specifically, we have identified six different key questions that we aim to answer through our experiments:

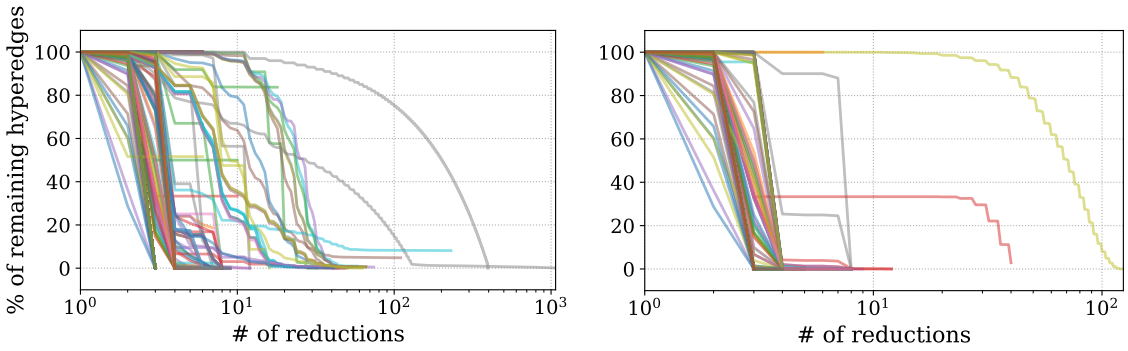
- **KQ1**: How effective are the proposed exact reduction rules in shrinking the hypergraphs while preserving the minimum cut value?
- **KQ2**: How effective is it to search for multiple contractions per iteration in the ordering-based algorithm?
- **KQ3**: How does HEICUT compare to the state-of-the-art hypergraph minimum cut algorithms in terms of solvability, runtime and memory usage?
- **KQ4**: What is the trade-off between accuracy and efficiency when enabling heuristic reduction via label propagation?
- **KQ5**: What are the performance benefits when parallelizing the exact reduction rules and the ordering-based solver?
- **KQ6**: What is the speed of the hypercactus algorithm of Chekuri and Xu when using the proposed improvements?

To answer the first key question **KQ1**, we evaluate the effectiveness of our exact reduction rules in Section 6.4.1. Afterwards, we address **KQ2** in Section 6.4.2 by assessing the effectiveness of searching for multiple contractions in the ordering-based algorithm. We tackle **KQ3** and **KQ4** by comparing HEICUT to all four competitors and to HEICUTLP over all datasets in Section 6.4.3. We then answer **KQ5** in Section 6.4.4 by measuring how the running time varies when increasing the number of threads for HEICUT. Finally, **KQ6** is addressed in Section 6.4.5, where we perform the first ever evaluation of the hypercactus algorithm of Chekuri and Xu, including our proposed improvements.

6.4.1 Effectiveness of Exact Reduction Rules

We first address **KQ1** by evaluating the effectiveness of our exact reduction rules defined in Section 4.3.1. For this, we measure after each reduction the percentage of remaining hyperedges relative to the original input. Note that we specifically focus on the exact variant HEICUT, i.e., we omit the label propagation heuristic of Section 4.3.2. Figure 6.2 visualizes the effectiveness of the reductions of HEICUT on the medium dataset M_{HG}. More specifically, each line represents a medium-sized hypergraph instance. The line of a hypergraph instance ends if it cannot be reduced further. We observe that most of the unweighted and weighted instances of M_{HG} are fully reduced by the exact reduction rules in fewer than ten reductions. Later reductions yield progress in the remaining harder instances. In particular, 85% of the unweighted and 87% of the weighted hypergraphs are fully reduced, i.e., $|E| = 0$ or $|V| = 1$. For most of the other instances, we detect during the reductions that $\hat{\lambda}(H) = 0$ and already terminate before reducing the hypergraph fully. Therefore, we call the underlying ordering-based solver only for 7% of the weighted instances and only for a single weighted instance. An even stronger effect can be observed for the large dataset L_{HG}, where 89% of the unweighted and 87% of the weighted instances are fully reduced. Surprisingly, all large-sized instances can already be solved during the kernelization phase and do not call the underlying ordering-based solver, except for a single unweighted instance. These results demonstrate the strong practical effectiveness of our exact reductions on real-world hypergraphs, even without the label propagation heuristic.

To analyze the effectiveness of each individual reduction rule, we measure for each rule the average time usage over all rounds of all medium-sized instances. In addition, we measure the average hyperedge reduction for each rule, which is defined as the percentage of hyperedges contracted by the rule relative to the size before applying the rule. In other words, if a reduction rule has a hyperedge reduction of 0%, it never contracts any hyperedge and if it has a hyperedge reduction of 50%, it always halves the number of hyperedges whenever it is applied. The results for the medium dataset M_{HG} are provided in Table 6.1. Note that we omit Reduction Rule 1 because it is implicitly implemented and we cannot measure its performance directly. Besides, Reduction Rules 6 and 7 are evaluated together because they are implemented within the same loop. We observe that the effectiveness of a reduction rule correlates with its position in the order, i.e., the first reduction rules are the most effective. As the time used by the first and last reduction rules is almost the same, we



(i): Unweighted medium dataset M_{HG} **(ii):** Weighted medium dataset M_{HG}

Figure 6.2: Remaining size of each medium-sized hypergraph relative to the number of exact reductions performed by HEICUT. Each line shows one hypergraph and ends if the hypergraph cannot be reduced further. The remaining size is given as a percentage of the original number of hyperedges.

can conclude that the proposed order of the reduction rules is appropriate. However, we observe that Reduction Rule 4 is approximately 18 times slower than the second slowest reduction rule, while contracting only around one percent of the hyperedges. The same can be observed for the large dataset L_{HG}. An explanation for this is that real-word hypergraphs do not admit many nested substructures. Therefore, to improve the performance of our algorithm further, we will not apply Reduction Rule 4 in the following experiments, as it offers little to no benefit in practice.

Answer to KQ1. *The proposed exact reduction rules are highly effective, since at least* 85% *of the real-world instances can be maximally reduced, thus providing the exact minimum cut without the need to run the underlying ordering-based algorithm. Reduction Rule 4 can be omitted to improve the performance further, since it is the least effective reduction rule and has the highest average time usage.*

6.4.2 Effectiveness of Multiple Contractions

To answer **KQ2**, we evaluate the effectiveness of performing multiple contractions per iteration for the ordering-based algorithm, as described in Section 4.3.3. In particular, as we only focus on the tight ordering, we are interested in the increase in performance when applying the recurrent contraction rule defined in Theorem 4.2. For this, we first measure the effectiveness of the multiple contractions when running the ordering-based solver as a standalone algorithm. For the unweighted medium-sized instances, the multiple contractions lead to 7.0% less iterations and a speedup of 1.05 in the geometric mean. Similarly, we obtain 7.2% less iterations and a speedup of 1.09 for the weighted medium-sized in-

Reduction Rule	Number	Time (s)	Hyperedge Reduction (%)
Unweighted medium dataset M_{HG}			
HeavyHyperedge	2	0.11	28.80
HeavyOverlap	3	0.28	26.75
NestedSubstructure	4	5.31	1.02
ImbalancedVertex	5	0.04	5.72
ImbalancedTriangle	6	0.02	3.85
HeavyNeighborhood	7		
Weighted medium dataset M_{HG}			
HeavyHyperedge	2	0.33	83.12
HeavyOverlap	3	0.84	87.56
NestedSubstructure	4	5.89	1.30
ImbalancedVertex	5	0.15	37.25
ImbalancedTriangle	6	0.21	7.98
HeavyNeighborhood	7		

Table 6.1: Average time usage and average hyperedge reduction for each reduction rule over all rounds of all medium-sized instances. Reduction Rules 6 and 7 are evaluated together.

stances. Overall, using the multiple contractions technique is faster for 70% of the instances in M_{HG}. Note that we cannot report results for the large dataset L_{HG}, as the standalone ordering-based solver does not finish for any large-sized instance within the given time limit. In addition, we measure the benefit from performing multiple contractions in the underlying ordering-based solver of HEICUT to solve the kernels that cannot be reduced further. For both the medium dataset M_{HG} and the large dataset L_{HG}, we observe that the minimum cut value of the kernel can be found around 1.35 times faster when performing multiple contractions per iteration. This means that Theorem 4.2 is much more effective when being combined with the kernelization phase, making it an important component of HEICUT. In fact, every kernel of HEICUT that is passed to the underlying ordering-based solver is solved faster when searching for multiple contractions per iteration.

Answer to KQ2. *Searching for multiple contractions per iteration is faster for* 70% *of the instances if the ordering-based solver is used as a standalone algorithm. When using it as an underlying solver for the kernels of* HEICUT*, the multiple contractions technique is always faster with an* 1.35 *speedup in the geometric mean.*

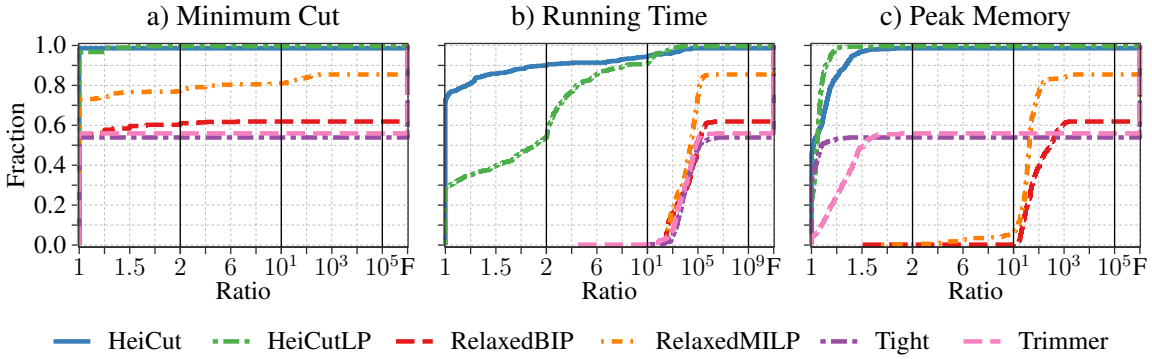
6.4.3 Comparison against State-of-the-Art

We address **KQ3** and **KQ4** by comparing HEICUT to the four competitors described in Section 6.3 and to HEICUTLP, which uses the heuristic reduction via label propagation as described in Section 4.3.2. For this, we evaluate all algorithms in terms of cut value, running time and peak memory on three datasets. We first start with the medium dataset M_{HG}, as it contains a large number of hypergraphs and gives a good overall comparison between the algorithms. Afterwards, we use the large dataset L_{HG} to asses how the algorithms scale to hypergraphs with hundreds of millions of hyperedges. Finally, we compare the algorithms on the new $(k, 2)$-core dataset to see how they perform when the instances do not admit a trivial minimum cut.

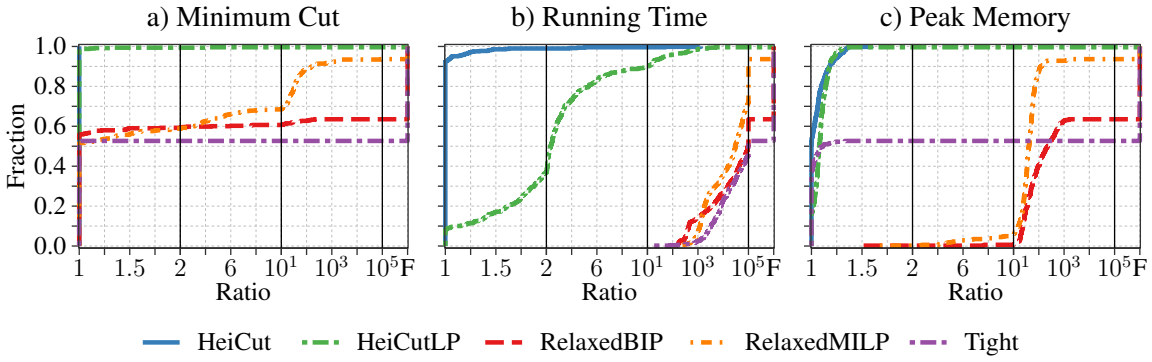
Medium Dataset $\mathbf{M_{HG}}$

Figure 6.3 shows the performance profiles on the unweighted and weighted instances of the medium dataset M_{HG}. We observe that HEICUT dominates the competition on all instances by a large margin. In particular, on the unweighted instances, HEICUT solves 98.6% of the instances, while TRIMMER is the next best exact algorithm with 55.9% of solved instances. On all instances that TRIMMER solved, HEICUT is much faster. More specifically, HEICUT is at least 1 000 times faster than TRIMMER on 85% of the instances solved by TRIMMER. On 95% of those instances, HEICUT is more memory efficient. We also observe that TIGHT solves 53.9% of the instances and thus performs similar to TRIMMER, meaning the sparsification approach used for TRIMMER has no significant benefit in practice. Surprisingly, our simple BIP and MILP formulations perform slightly better than TIGHT and TRIMMER on most unweighted instances. In particular, RELAXEDBIP returns the optimal solution for 55.9% of the instances, while RELAXEDMILP finds the exact minimum cut in 73.1% of the instances and yields an inexact solution for 10.6% of the instances. This means that the linear number of constraints in the MILP formulation results in a noticeable increase in quality. Nevertheless, HEICUT is not only much faster but also more memory efficient than RELAXEDBIP and RELAXEDMILP on all unweighted instances of M_{HG}. While the failed instances of TIGHT and TRIMMER most often exceed the time limit, RELAXEDBIP and RELAXEDMILP mostly fail because they run out of memory.

A similar observation can be made for the weighted instances of M_{HG}. HEICUT solves all instances, while TIGHT is the second-best exact algorithm with 52.7% solved instances. On all instances that TIGHT solved, HEICUT is always faster and at least 1 000 times faster on 95% of those instances. HEICUT is more memory efficient on 60% of those instances. Note that TRIMMER does not work on weighted hypergraphs. Our relaxed BIP and MILP formulations have a similar quality than TIGHT. In particular, RELAXEDBIP returns the optimal solution for 55.9% of the instances, while RELAXEDMILP finds the exact minimum cut in 51.5% of the instances and yields an inexact solution for 44.6% of the instances. On all weighted instances of M_{HG}, RELAXEDBIP and RELAXEDMILP are dominated by HEICUT for all metrics.



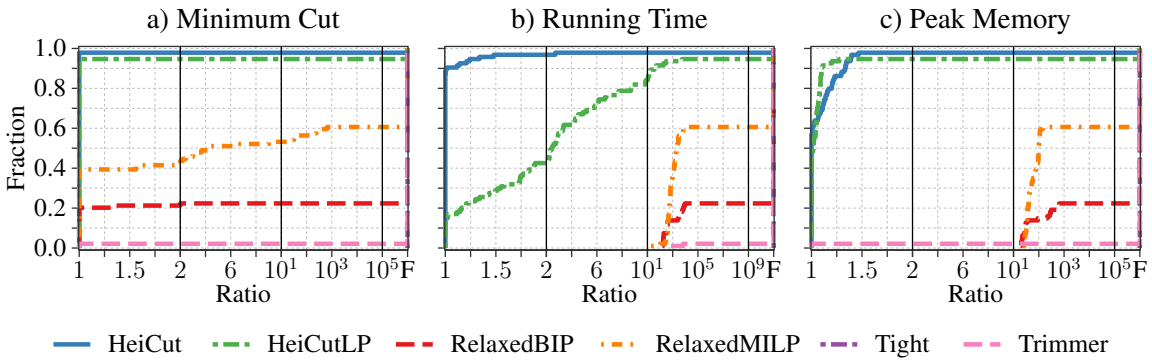
(i): Unweighted medium dataset M_{HG}



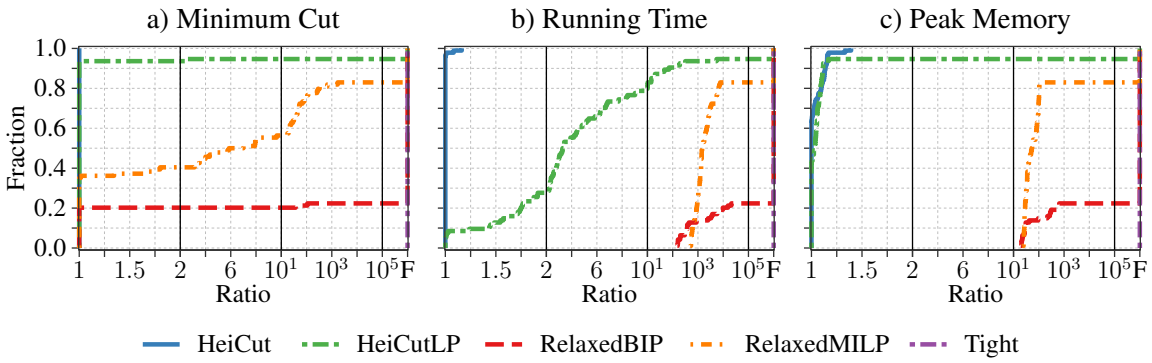
(ii): Weighted medium dataset M_{HG}

Figure 6.3: Performance profiles of all algorithms on the unweighted and weighted instances of the medium dataset M_{HG}. Note that TRIMMER only runs on unweighted hypergraphs.

When analyzing the results of HEICUTLP, we observe that HEICUTLP still finds the exact minimum cut value for approximately 97% of both unweighted and weighted instances of M_{HG}, despite using the label propagation heuristic. This means that the solution quality of HEICUTLP is almost as good as that of HEICUT. Besides, both use roughly the same memory. However, HEICUTLP is in general slower than HEICUT. This is because, although label propagation results in more aggressive contractions, performing the label propagation itself takes long and dominates the overall running time on most instances. Nevertheless, HEICUTLP is significantly faster on hard instances. More specifically, there exist eleven unweighted and one weighted medium-sized instance where the exact reduction rules of HEICUT have no effect and the hypergraphs must be solved in their original form by the underlying ordering-based solver. In contrast, the label propagation heuristic of HEICUTLP manages to fully reduce all of these hard instance, so that it can stop early and the underlying ordering-based algorithm is never called. When taking the geometric mean over all these hard instances, HEICUTLP is approximately 3 000 times faster than HEICUT, while always finding the exact minimum cut value. Therefore, the label propagation heuristic proves to be useful in the few cases where the exact reduction rules have no effect on the input hypergraph.



(i): Unweighted large dataset L_{HG}



(ii): Weighted large dataset L_{HG}

Figure 6.4: Performance profiles of all algorithms on the unweighted and weighted instances of the large dataset L_{HG}. Note that TRIMMER only runs on unweighted hypergraphs.

Large Dataset $\mathbf{L_{HG}}$

The performance profiles on the unweighted and weighted instances of the large dataset L_{HG} are given in Figure 6.4. Similar to the medium dataset M_{HG}, we observe that HEICUT dominates the competition on all instances by a large margin. HEICUT solves 97.9% of the unweighted instances, while TRIMMER is the next best exact algorithm with only two solved instances. Note that TIGHT does not solve any instance within the given time limit. This means that both TRIMMER and TIGHT struggle to solve any large-sized instance in less than two hours, while HEICUT solves nearly all of them by using only 3.1 seconds in the geometric mean. These results demonstrate the significant performance gap between HEICUT and the current state-of-the-art algorithms. We also observe that RELAXEDBIP returns the optimal solution for around 20.4% of the instances, whereas RELAXEDMILP finds the exact minimum cut in 38.7% of the instances and yields an inexact solution for 21.5% of the instances. As for the medium dataset M_{HG}, this means that the linear number of constraints in the MILP formulation results in a noticeable increase in solution quality. Nevertheless, RELAXEDBIP and RELAXEDMILP are outper-

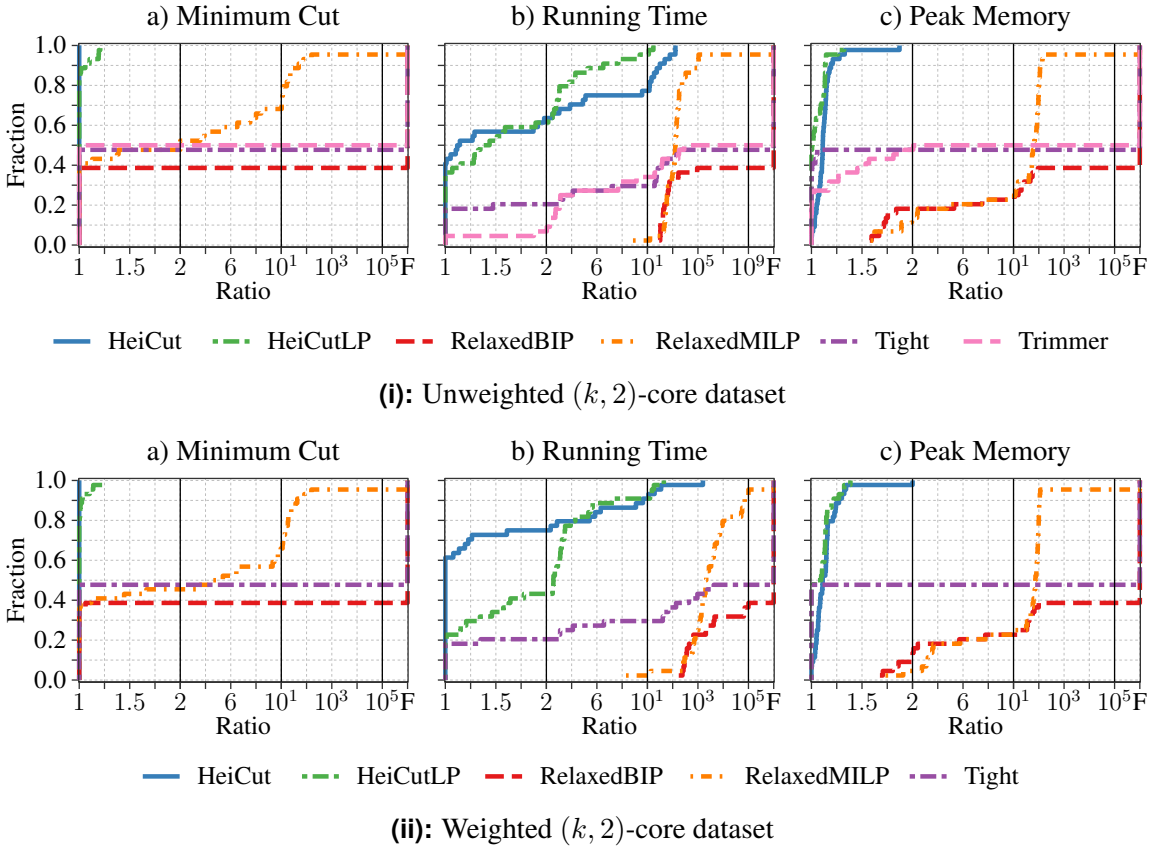
formed by HEICUT for each metric. More specifically, HEICUT is at least 1 000 times faster than RELAXEDMILP on around 40% of the instances for which RELAXEDMILP returns a solution. On all of those instances, HEICUT uses significantly less memory. All failed instances of TIGHT and TRIMMER exceed the time limit, while RELAXEDBIP and RELAXEDMILP mostly fail because they run out of memory.

We obtain similar results for the weighted instances of L_{HG}. In particular HEICUT solves all weighted large instances, while TIGHT fails to solve any instance within the constraints. Note that TRIMMER does not work on weighted hypergraphs. RELAXEDBIP returns the optimal solution for 19.4% of the instances, while RELAXEDMILP finds the exact minimum cut in 34.4% of the instances and yields an inexact solution for 48.4% of the instances. On all weighted instances of L_{HG}, HEICUT is at least 1 000 times faster than RELAXEDMILP on approximately 70% of the instances for which RELAXEDMILP returns a solution. Besides, HEICUT is more memory efficient than RELAXEDMILP on all of those instances.

Regarding the label propagation heuristic, we observe that HEICUTLP still manages to find the exact minimum cut value for approximately 93% of both unweighted and weighted instances of L_{HG}. In other words, the solution quality of HEICUTLP nearly matches that of HEICUT. In addition, both algorithms use roughly the same memory. However, we observe that HEICUTLP is slower than HEICUT on most instances. The reason is the same as for the medium dataset M_{HG}, i.e., although label propagation results in more aggressive contractions, performing the label propagation itself takes long and dominates the overall running time on most instances. Contrary to the medium dataset M_{HG}, there exist fewer hard large-sized instances on which HEICUTLP outperforms HEICUT. This is because the exact reduction rules of HEICUT are more effective on the instances of L_{HG}. In particular, only a single unweighted instance cannot be fully reduced during the kernelization phase. For this instance, the label propagation heuristic of HEICUTLP manages to fully reduce the hypergraph, so that it can stop early and the underlying ordering-based algorithm is not called. Therefore, HEICUTLP is 1.34 times faster than HEICUT on this hard instance. Nevertheless, we conclude that the label propagation heuristic is less effective on L_{HG} than on M_{HG} as there are fewer instances for which the exact reduction rules are ineffective.

$(k, 2)$-Core Dataset

We also test the algorithms on the new $(k, 2)$-core dataset, where each hypergraph has a minimum cut value that is strictly lower than the smallest weighted vertex degree. These instances are harder to solve, since they do not admit a trivial minimum cut. The performance profiles on the unweighted and weighted instances of the $(k, 2)$-core dataset are provided in Figure 6.5. The results show that HEICUT dominates the competition even on the new $(k, 2)$-core dataset. HEICUT solves all of the unweighted instances, while TRIMMER finds the optimal solution for 50% and TIGHT for 47.73% of the unweighted instances. Furthermore, HEICUT is faster than TRIMMER and TIGHT on approximately 60% of their respective successfully finished unweighted instances. This means that, on the instances



(i): Unweighted $(k, 2)$-core dataset

(ii): Weighted $(k, 2)$-core dataset

Figure 6.5: Performance profiles of all algorithms on the unweighted and weighted instances of the $(k, 2)$-core dataset. Note that TRIMMER only runs on unweighted hypergraphs.

where TRIMMER and TIGHT finish, the dominance of HEICUT is not as pronounced as for the medium or large dataset. Nevertheless, TRIMMER and TIGHT fail to find the optimal solution within two hours for half of the unweighted $(k, 2)$-core instances, while HEICUT solves all unweighted instances using only 1.64 seconds in the geometric mean. This means that HEICUT still outperforms TRIMMER and TIGHT by a large margin. We also observe that RELAXEDBIP and RELAXEDMILP perform slightly worse than TRIMMER and TIGHT on the unweighted $(k, 2)$-core instances. RELAXEDBIP returns the optimal solution for 37.2% of the instances, while RELAXEDMILP finds the exact minimum cut in 39.5% of the instances and yields an inexact solution for 55.8% of the instances. HEICUT is faster and uses less memory than RELAXEDMILP on all unweighted instances.

A similar observation can be made for the weighted $(k, 2)$-core instances. More specifically, HEICUT solves all weighted instances, while TIGHT finds the optimal solution for 47.73% of the weighted instances. We observe that HEICUT is faster than TIGHT on 60% of the instances solved by TIGHT. RELAXEDBIP returns the optimal solution for 34.9% of the instances, while RELAXEDMILP finds the exact minimum cut in 32.6% of the instances and yields an inexact solution for 62.7% of the instances. RELAXEDBIP and RELAXEDMILP are dominated by HEICUT on all weighted $(k, 2)$-core instances.

The results show that HEICUTLP still finds the exact minimum cut value for approximately 80% of both unweighted and weighted $(k, 2)-$instances. Although HEICUTLP performs slightly worse in terms of solution quality compared to the medium and large datasets, it is nearly as fast as HEICUT on the $(k, 2)-$instances. In particular, HEICUTLP is faster than HEICUT for around 57% of the unweighted and around 40% of the weighted instances. Both algorithms have roughly the same memory usage. There are thirteen unweighted and seven weighted instances that are not fully reduced by the kernelization phase of HEICUT. The label propagation heuristic of HEICUTLP manages to fully reduce all of these hard instance, so that it can stop early and the underlying ordering-based algorithm is never called. When taking the geometric mean over all these hard instances, HEICUTLP is approximately five times faster than HEICUT, while finding the exact minimum cut value in 65% of those instances. Thus, compared to the medium and large datasets, HEICUTLP can overall compete more with HEICUT in terms of running time, but it is less effective on the hard instances, as it does not always find the optimal solution.

Answer to KQ3. HEICUT *solves twice as many medium-sized instances as the next best competitor. On the large dataset,* HEICUT *solves nearly all instances, while all of the competitors solve almost none of them.* HEICUT *also dominates its competitors on the new* $(k, 2)$*-core dataset. Overall,* HEICUT *is three to four orders of magnitude faster than the current state-of-the-art algorithms.*

Answer to KQ4. *The label propagation heuristic proves to be effective on specific medium-sized hypergraphs where no exact reduction rule applies.* HEICUTLP *greatly reduces the size of those harder instances, while still finding the exact solution for most of them. The heuristic is less effective on large and* $(k, 2)$*-core hypergraphs.*

6.4.4 Parallelization

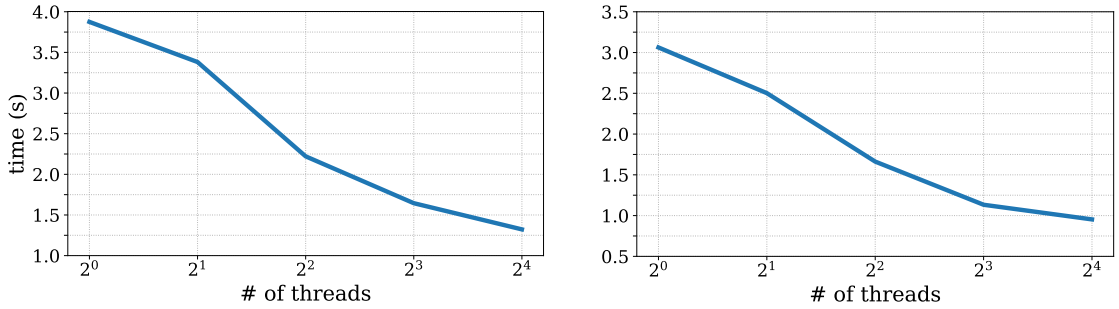
We answer **KQ5** by first assessing how much the exact ordering-based algorithm benefits from the parallelization described in Section 4.4.2. For this, we run the ordering-based solver with $N = 16$ threads on the medium dataset M_{HG} and compare it to the sequential case. Note that we do not use the large dataset L_{HG}, as the ordering-based algorithm does not finish for any large-sized instance in the sequential case, which would make the comparison less insightful. However, testing how many large-sized instances can be solved with parallelization could be future work. We run two variants of the parallel algorithm. For the first one, we ensure that each thread i computes a tight ordering at a random start, i.e., we set $\alpha_i = 0$ for all $1 \leq i \leq N$. For the second variant, we enforce more diversification by using a uniform distribution, i.e., we set $\alpha_i = \frac{i-1}{N-1}$ for all $1 \leq i \leq N$. None of the variants searches for multiple contractions per iteration. The results are provided in Table 6.2. We observe that both parallel variants lead to more finished runs and a better running time

Variant	Threads	Finished (%)	Time (s)	Contractions (per Iteration)
Unweighted medium dataset M_{HG}				
Tight	1	53.89	439.69	1.00
Tight	16	**67.62**	**105.32**	12.29
Uniform	16	59.84	153.22	**14.15**
Weighted medium dataset M_{HG}				
Tight	1	52.66	501.73	1.00
Tight	16	**64.55**	**125.62**	9.98
Uniform	16	63.93	167.04	**12.33**

Table 6.2: Comparison of the sequential case and both parallel variants of the ordering-based solver on the medium dataset M_{HG}. We report the percentage of finished runs as well as the geometric mean of time usage and contractions per iteration.

compared to the sequential case. The reason for this is that the parallel execution allows each thread to find a different ordering, resulting in more than one contraction per iteration. However, we observe that the tight variant outperforms the uniform variant in terms of finished runs and time usage, even though it performs less contractions per iteration. This is because most of the threads of the uniform variant use the adjacency contribution, which leads to many `increaseKey` operations in the priority queue during the computation of the ordering. In contrast, if all threads use the tight ordering, they all rely only on the containment contribution and can compute the ordering much faster. This means that even though a single iteration of the uniform variant is more effective, the tight variant can compute many more iterations, so that it is overall more effective. Thus, for measuring the performance of the parallel version of HEICUT in the following, we will ensure that each thread of the underlying ordering-based solver computes a tight ordering.

We want to evaluate the increase in performance obtained from parallelizing HEICUT as described in Section 4.4, which consists of the parallelization of the exact reduction rules and of the underlying ordering-based solver. For the latter, we also enable that each thread searches for multiple contractions per iteration. As the parallelization should have the most impact on large hypergraphs, we run HEICUT on the large dataset L_{HG} with $1, 2, 4, 8$ and 16 threads and take for each run the geometric mean of the running time over all instances. Figure 6.6 shows the so-called *strong scalability plot* of HEICUT, which indicates how the running time varies with increasing number of threads for the same fixed dataset L_{HG}. As expected, the time usage decreases with the level of parallelization for both the unweighted and weighted instances. In particular, the running time of the sequential case is roughly divided by $\log(N)$ if we use N threads. This logarithmic relationship means that the performance increases only slowly with the number of threads. One reason for this is that the sequential version of HEICUT already manages to solve almost all large instances in under



(i): Unweighted large dataset L_{HG} **(ii):** Weighted large dataset L_{HG}

Figure 6.6: Strong scalability plot of HEICUT, which indicates how the running time varies with increasing number of threads for the same fixed dataset L_{HG}. Note we take for each parallelization level the geometric mean of the running times over all large instances.

four seconds, which does not leave much room for improvement. Therefore, it would be more insightful to measure the benefit of parallelizing HEICUT on a harder dataset, which could be done in future work. Nonetheless, when using $N = 16$ threads, HEICUT manages to solve one more unweighted instance compared to the sequential case. This means that only a single unsolved unweighted instance remains, which will most probably be solved by increasing the number of threads even further.

Answer to KQ5. *The performance of* HEICUT *can be improved further by parallelizing the exact reduction rules and the ordering-based solver. However, we obtain only a logarithmic speedup, since the sequential version already runs in under four seconds for most large instances. Thus, a harder dataset is needed for a better comparison.*

6.4.5 Hypercactus

We tackle **KQ6** by performing the first ever evaluation of the hypercactus algorithm of Chekuri and Xu, which includes the kernelization phase described in Section 5.2 and the improvements to the split oracle as outlined in Section 5.3. In particular, we test how the algorithm performs when computing the hypercactus representation of every unweighted and weighted instance in the medium dataset M_{HG}. The results show that the algorithm manages to construct the hypercactus representation for 67.41% of the unweighted and all but one weighted instances of M_{HG} within the given constraints, i.e., two hours time limit and 100 GB virtual memory limit. The gap in the success rate can be explained by the fact that the weak reduction rules are less effective on the unweighted instances, as shown in Table 6.3. More specifically, we observe that, on average, the weak reduction rules shrink the input hypergraph to a kernel with only around 56 hyperedges for the weighted instances. In contrast, the average kernel of the unweighted instances has approximately 26 100 hyperedges. In particular, all but one weighted instances are solved

Hypergraph	Number of Hyperedges	Number of Vertices
Unweighted medium dataset M_{HG}		
Input	871 796.25	642 154.85
Kernel	26 094.60	32 186.17
Hypercactus	135.66	9 291.50
Weighted medium dataset M_{HG}		
Input	871 796.25	642 154.85
Kernel	55.91	6 216.41
Hypercactus	17.41	6 189.85

Table 6.3: Average number of hyperedges and vertices for the input hypergraph, the weak kernel and the final hypercactus representation.

within less than three minutes, with a geometric mean of 0.34 seconds. The only unsolved instance is `Trefethen_20000`, for which none of the weak reduction rules apply. The solvable unweighted instances use 0.71 seconds in the geometric mean. This means that the performance of the hypercactus algorithm is greatly boosted by the novel kernelization phase and the algorithm only struggles if the reduction rules are not effective. To put the performance into perspective, our proposed hypercactus algorithm finds all minimum cuts in more medium-sized hypergraphs than a single minimum cut is found by the current state-of-the-art algorithms TRIMMER and TIGHT. For both the unweighted and weighted instances of M_{HG}, the hypercactus algorithm uses less than one gigabyte of memory, which demonstrates the effectiveness of the memory-efficient decomposition tree proposed by Chekuri and Xu. The size distribution of the final hypercactus representations seems to follow a power law, although a more sophisticated analysis of the hypercactus structure could be done in future work.

Answer to KQ6. *With the proposed improvements, the hypercactus representation can be constructed for* 67.41% *of the unweighted and all but one weighted medium-sized instances within the two hours time limit. The performance is greatly boosted by the proposed kernelization phase. In comparison, our implementation finds all minimum cuts in more medium-sized hypergraphs than a single minimum cut is found by current state-of-the-art algorithms.*

CHAPTER 7

Discussion

In Section 7.1, we provide a brief summary of the novel techniques described in this thesis and draw a conclusion based on the performed experiments. Section 7.2 then discusses the various approaches that could be explored in future work.

7.1 Conclusion

In this work, we present HEICUT, a lightweight and scalable algorithm that quickly finds a minimum cut in unweighted or weighted hypergraphs with hundreds of millions of hyperedges. HEICUT applies seven provably exact reduction rules to aggressively reduce the input hypergraph to a smaller kernel without losing information about the original minimum cut value. These exact reduction rules consist of novel hypergraph-specific rules as well as generalizations of already proven rules from graph algorithms. Besides, HEICUT supports the option of performing a heuristic reduction by identifying and contracting a clustering via label propagation. After shrinking the hypergraph, HEICUT applies an existing ordering-based solver to find an exact minimum cut in the reduced hypergraph. For this, we extend the ordering-based algorithm by defining novel recurrent rules that allow to search for multiple contractions per iteration. This multiple contractions technique can also be applied to any other algorithm that uses the ordering-based solver in a subroutine, directly improving its performance in practice.

We perform extensive experiments on more than 500 real-world and synthetic hypergraphs. First, we show that defining and solving simple BIP and MILP formulations of the hypergraph minimum cut problem is already sufficient to match the performance of current state-of-the-art algorithms on most instances. This demonstrates the unsatisfactory performance of the state-of-the-art competitors in practice. In addition, we observe that HEICUT solves twice as many medium-sized instances as the next best competitor. On the large dataset with up to hundreds of millions of hyperedges, HEICUT solves nearly all instances, while all of the competitors solve almost none of them. Overall, HEICUT is three to four orders of magnitude faster than the current state-of-the-art algorithms. More

specifically, HEICUT achieves near-linear running time in practice, even if it has a non-linear time complexity in theory. This is because 85% of all instances can be maximally reduced by the exact reduction rules, eliminating the need to run the underlying ordering-based solver. Combined with further stopping criteria, HEICUT can already solve 95% of all instances with the exact reduction rules. Nevertheless, the label propagation heuristic proves to be effective on very specific hypergraphs where the exact reduction rules have no effect. In particular, the heuristic manages to aggressively reduce the size of those harder instances, while still finding the optimal solution for most of them. We also introduce a new dataset of $(k, 2)$-core instances, where the minimum cut value is guaranteed to be different from the smallest weighted vertex degree. These instances are harder to solve as they do not admit a trivial minimum cut and provide a benchmark for future research on the hypergraph minimum cut problem.

We improve the performance of HEICUT further by parallelizing the exact reduction rules and the ordering-based solver. For the latter, we present the first ever parallel implementation, to the best of our knowledge. For this, we use persistent threads with barriers to perform thread-specific contractions in each iteration. Our new parallel approach benefits any algorithm that uses the ordering-based solver in a subroutine. The greatest speedup is obtained when combining the parallel execution with the proposed multiple contractions technique, i.e., all threads search independently for multiple contractions in each iteration.

Finally, we describe how a weaker version of the exact reduction rules can be applied to the hypercactus algorithm of Chekuri and Xu, which aims to find all minimum cuts in a hypergraph. Together with several practical improvements to their split oracle, we manage to propose the first ever implementation of their hypercactus algorithm, to the best of our knowledge. The experiments show that our implementation highly benefits from the proposed kernelization. In particular, it finds the hypercactus representation within less than a second for 67.41% of the unweighted and all but one weighted medium-sized instances. This means that our proposed hypercactus algorithm finds all minimum cuts in more medium-sized hypergraphs than a single minimum cut is found by the current state-of-the-art algorithms. We plan to publish the source code of our hypercactus algorithm in the same repository that contains the code for HEICUT [1].

7.2 Future Work

We propose a series of novel techniques and implementations that can be built upon in a variety of ways in future work. First, the kernelization phase of HEICUT can be improved even further by finding more exact hypergraph-specific reduction rules or generalizing all remaining graph rules from VIECUT to hyperedges of arbitrary size. In addition, we are interested in testing both the sequential and the parallel variant HEICUT on even harder instances to gain a comprehensive understanding of its potential. Next, we are convinced that the novel technique of performing multiple contractions per iteration in the ordering-

[1] `https://github.com/HeiCut/HeiCut`

based solver can be optimized even further by extending the proposed theorems. One way to achieve this is to allow some changes in the adjacency and containment contributions after the contraction of the eligible vertices, if it can be guaranteed that these changes still preserve the ordering. Our first ever parallel implementation of the ordering-based solver, along with the new concept of computing different orderings in each thread, is worthy of a more thorough analysis. This includes testing it on the larger dataset, to fully measure its capabilities compared to the sequential case.

Another straightforward possibility is to improve the performance of other hypergraph algorithms that rely on minimum cuts by replacing the existing minimum cut algorithm with HEICUT. This covers fields such as cybersecurity [78], hypergraph expansion [65] and quantum computing [55]. In the same way, our work can be used to extend graph-based algorithms for clustering [36], network reliability [45], community detection [11] and VLSI design [3] to the hypergraph scenario. Besides, our novel implementation of the hypercactus algorithm paves the way for a multitude of hypergraph algorithms in related fields, such as incremental minimum cuts [34], enumerating all minimum cuts [7] and connectivity augmentation [17, 25]. Lastly, a more sophisticated structural analysis of the generated hypercactus representations can provide new useful insights.

APPENDIX A

Appendix

A.1 Command-Line Arguments

In the following, we list the command-line arguments used for every tested algorithm. Note that `PATH` is only a placeholder and must be replaced with the real path to the hypergraph of the given instance.

Command of HeiCut:

```
$ ./kernelizer PATH --ordering_type=tight --ordering_mode=multi
```

Command of HeiCutLP:

```
$ ./kernelizer PATH --ordering_type=tight --ordering_mode=multi
   ↪ --lp_num_iterations=1
```

Command of RelaxedBIP:

```
$ ./ilp PATH --ilp_mode=BIP
```

Command of RelaxedMILP:

```
$ ./ilp PATH --ilp_mode=MILP
```

Command of Tight:

```
$ ./submodular PATH -ordering_type=tight --ordering_mode=single
```

Command of Trimmer:

```
$ ./trimmer PATH -ordering_type=tight --ordering_mode=single
```

A.2 Pseudocode of TRIMMER

The TRIMMER algorithm by Chekuri and Xu [14, 15] is a sparsification-based method for computing the minimum cut in unweighted hypergraphs. It constructs a sequence of sparsified hypergraphs, known as k-trimmed certificates, each of which preserves all local connectivities up to a threshold k. These certificates contain only $O(kn)$ hyperedges and are passed to an exact minimum cut solver. The algorithm iteratively doubles k and computes the minimum cut of each certificate until the cut value of the original hypergraph is recovered. For more information, we refer to Section 3.2 and to the original papers.

We implement TRIMMER as described by the Chekuri and Xu. To construct a k-trimmed certificate H_k, the algorithm first computes an MA ordering of the vertices in H, starting from a random vertex. Next, we define the *head* of each hyperedge e as its first pin in the MA ordering and sort the hyperedges by the position of their heads, breaking ties using their original order. This produces the *hyperedge head ordering*, with which we compute a list of *backward edges* for each vertex $v \in V$. More specifically, the backward edges of v are constructed by taking all hyperedges containing v but for which v is not the head and sorting them by the hyperedge head ordering. This allows us to efficiently construct H_k for any k. The algorithm proceeds in iterations, starting with $k = 2$. In each iteration, we construct H_k and compute its minimum cut using the exact ordering-based solver with the tight ordering [56]. If the computed minimum cut value $\lambda(H_k)$ is strictly less than k, then $\lambda(H) = \lambda(H_k)$ and the algorithm terminates. Otherwise, k is doubled and the process continues. A pseudocode of our implementation is provided in Algorithm 6.

Algorithm 6: TRIMMER

Input: Hypergraph $H = (V, E)$
Output: Minimum cut value $\lambda(H)$

```
m ← |E|
(e_1, ..., e_m) ← HEADORDERING(H)  // Based on MA ordering of vertices
foreach hyperedge e_i in head ordering do
    foreach pin v ∈ e_i where v is not the head of e_i do
        Add e to backward edges of v
k ← 2
while true do
    H_k ← BUILDTRIMMEDCERTIFICATE(H, k)      // Using backward edges
    λ(H_k) ← ORDERINGBASEDSOLVER(H, α = 0)   // Using tight ordering
    if k > λ(H_k) then
        return λ(H) ← λ(H_k)
    k ← 2 · k
```

A.3 Further Proofs

Proof A.1 (Parallelization of Reduction Rule 5 without the restriction of VIECUT) *Let $v \in V$ be a vertex with two hyperedges $e_{uv} = \{u, v\} \in E$ and $e_{vw} = \{v, w\} \in E$ that both satisfy Reduction Rule 5. The contraction of the hyperedges is performed in two steps, i.e., first contracting e_{uv} and then e_{vw}. After contracting e_{uv}, the vertices u and v are merged into a new vertex v^*, meaning that e_{vw} becomes e_{v^*w}. The authors of* VIECUT *claim that Reduction Rule 5 may be invalid for e_{v^*w}, even if it was previously satisfied by e_{vw}. We will show that this is not the case in our scenario. First, we observe that the hyperedge e_{vw} must satisfy the reduction rule via the inequality $d_\omega(v) < 2\omega(e_{vw})$, as the other inequality is independent of $d_\omega(v)$, which means that the contraction of e_{uv} could never invalidate the reduction rule for e_{v^*w}. However, this implies that hyperedge e_{uv} satisfies the reduction rule via $d_\omega(u) < 2\omega(e_{uv})$. If not, we would have $d_\omega(v) < 2\omega(e_{uv})$, which leads to the following contradiction:*

$$\begin{aligned} 2d_\omega(v) &= d_\omega(v) + d_\omega(v) \\ &< 2\omega(e_{uv}) + 2\omega(e_{vw}) \\ &= 2\left(\omega(e_{uv}) + \omega(e_{vw})\right) \\ &\leq 2d_\omega(v) \end{aligned}$$

Therefore, we have $d_\omega(u) < 2\omega(e_{uv})$. Since the weighted vertex degree of v^* is given by $d_\omega(v^*) = d_\omega(v) + d_\omega(u) - 2\omega(e_{uv})$, we see that $d_\omega(v^*) < d_\omega(v)$. This leads to the following chain of inequalities:

$$d_\omega(v^*) < d_\omega(v) < 2\omega(e_{vw}) \leq 2\omega(e_{v^*w})$$

Thus, we conclude that if Reduction Rule 5 can be applied to e_{vw}, then it can also be applied to e_{v^*w} after the contraction of e_{uv}. More generally, it follows that the reduction rule can be applied simultaneously to hyperedges that share a common vertex, making the restriction of VIECUT *unnecessary in our scenario.*

Zusammenfassung

Eine zentrale Aufgabe vieler Anwendungen wie Netzwerkzuverlässigkeit, Community-Identifizierung oder VLSI-Design ist das Lösen des Hypergraph-Minimum-Cut-Problems, dessen Ziel es ist, die Knoten eines Hypergraphen in zwei nicht leere Partitionen zu unterteilen, sodass das Gesamtgewicht der geschnittenen Hyperkanten minimiert wird. Obwohl in den letzten Jahren viele Fortschritte für Graphen erzielt wurden, bleibt die Erweiterung von Minimum-Cut-Algorithmen auf Hypergraphen eine Herausforderung. Aus diesem Grund stellen wir mehrere neuartige Algorithmen vor, beginnend mit einem Binary Integer Program (BIP) und einem Mixed-Integer Linear Program (MILP), die als einfache Baselines dienen. Anschließend präsentieren wir HEICUT, einen skalierbaren und schnellen Algorithmus zur Identifizierung eines minimalen Schnitts in Hypergraphen mit bis zu Hunderten von Millionen Hyperkanten. HEICUT verwendet sieben nachweislich exakte Reduktionsregeln, die die Größe des Hypergraphen reduzieren und dennoch die Ermittlung der optimalen Lösung ermöglichen. Diese Reduktionsregeln bestehen aus neuartigen hypergraphspezifischen Regeln sowie Verallgemeinerungen von bereits bewährten Regeln aus Graphen-Algorithmen. Optional kann HEICUT eine heuristische Reduktion auf Basis von Label-Propagation anwenden, um komplexe Strukturen zu verkleinern. Die reduzierte Instanz wird dann durch einen exakten, ordnungsbasierten Algorithmus gelöst, für den wir neuartige Regeln vorschlagen, die pro Iteration nach mehreren Kontraktionen suchen. Wir beschreiben auch, wie die unterschiedlichen Bestandteile von HEICUT parallelisiert werden können, inklusive der ersten parallelen Implementierung des ordnungsbasierten Algorithmus, nach aktuellem Wissensstand. Unsere umfangreiche Auswertung auf über 500 realen Hypergraphen zeigt, dass die Kombination von exakten Reduktionsregeln und vorzeitigem Stoppen bereits in über 95% der Fälle einen minimalen Schnitt identifizieren kann. HEICUT löst doppelt so viele Instanzen wie der nächstbeste State-of-the-Art-Algorithmus und ist dabei drei bis vier Größenordnungen schneller. Aufgrund dieser Resultate verwenden wir einige der Techniken von HEICUT, um den Hypercactus-Algorithmus von Chekuri und Xu zu erweitern und zu verbessern, der darauf abzielt, alle minimalen Schnitte in einem Hypergraphen zu finden. Nach aktuellen Wissensstand präsentieren wir die erste Implementierung des Hypercactus-Algorithmus. In den Experimenten gelingt es uns, alle minimalen Schnitte in mehr Hypergraphen zu finden, als ein einziger minimaler Schnitt von den aktuellen State-of-the-Art-Algorithmen gefunden wird.

Bibliography

[1] Wilhelm Ackermann. Zum hilbertschen aufbau der reellen zahlen. *Mathematische Annalen*, 99(1):118–133, 1928.

[2] Charles J. Alpert. The ISPD98 circuit benchmark suite. In Majid Sarrafzadeh, editor, *Proceedings of the 1998 International Symposium on Physical Design, ISPD 1998, Monterey, CA, USA, April 6-8, 1998*, pages 80–85. ACM, 1998. doi: 10.1145/274535.274546. URL https://doi.org/10.1145/274535.274546.

[3] Charles J. Alpert and Andrew B. Kahng. Recent directions in netlist partitioning: a survey. *Integr.*, 19(1-2):1–81, 1995. doi: 10.1016/0167-9260(95)00008-4. URL https://doi.org/10.1016/0167-9260(95)00008-4.

[4] Richard J. Anderson and Heather Woll. Wait-free parallel algorithms for the union-find problem. In Cris Koutsougeras and Jeffrey Scott Vitter, editors, *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, May 5-8, 1991, New Orleans, Louisiana, USA*, pages 370–380. ACM, 1991. doi: 10.1145/103418.103458. URL https://doi.org/10.1145/103418.103458.

[5] Srinivasa Rao Arikati and Kurt Mehlhorn. A correctness certificate for the stoer-wagner min-cut algorithm. *Inf. Process. Lett.*, 70(5):251–254, 1999. doi: 10.1016/S0020-0190(99)00071-X. URL https://doi.org/10.1016/S0020-0190(99)00071-X.

[6] Niklas Baumstark, Guy E. Blelloch, and Julian Shun. Efficient implementation of a synchronous parallel push-relabel algorithm. In Nikhil Bansal and Irene Finocchi, editors, *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, volume 9294 of *Lecture Notes in Computer Science*, pages 106–117. Springer, 2015. doi: 10.1007/978-3-662-48350-3_10. URL https://doi.org/10.1007/978-3-662-48350-3_10.

[7] Calvin Beideman, Karthekeyan Chandrasekaran, and Weihang Wang. Deterministic enumeration of all minimum cut-sets and k-cut-sets in hypergraphs for fixed k. *Math. Program.*, 207(1):329–367, 2024. doi: 10.1007/S10107-023-02013-8. URL https://doi.org/10.1007/s10107-023-02013-8.

[8] Anton Belov, Daniel Diepold, Marijn Heule, and Matti Järvisalo. The sat competition 2014. 2014.

[9] Austin R. Benson, David F. Gleich, and Jure Leskovec. Higher-order organization of complex networks. *CoRR*, abs/1612.08447, 2016. URL http://arxiv.org/abs/1612.08447.

[10] Rochelle L. Boehning, Ralph M. Butler, and Billy E. Gillett. A parallel integer linear programming algorithm. *European Journal of Operational Research*, 34 (3):393–398, 1988. ISSN 0377-2217. doi: https://doi.org/10.1016/0377-2217(88)90160-9. URL https://www.sciencedirect.com/science/article/pii/0377221788901609.

[11] Deng Cai, Zheng Shao, Xiaofei He, Xifeng Yan, and Jiawei Han. Mining hidden community in heterogeneous social networks. In Jafar Adibi, Marko Grobelnik, Dunja Mladenic, and Patrick Pantel, editors, *Proceedings of the 3rd international workshop on Link discovery, LinkKDD 2005, Chicago, Illinois, USA, August 21-25, 2005*, pages 58–65. ACM, 2005. doi: 10.1145/1134271.1134280. URL https://doi.org/10.1145/1134271.1134280.

[12] Ümit V. Çatalyürek and Cevdet Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. Parallel Distributed Syst.*, 10(7):673–693, 1999. doi: 10.1109/71.780863. URL https://doi.org/10.1109/71.780863.

[13] Ümit V. Çatalyürek, Karen D. Devine, Marcelo Fonseca Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. More recent advances in (hyper)graph partitioning. *ACM Comput. Surv.*, 55(12):253:1–253:38, 2023. doi: 10.1145/3571808. URL https://doi.org/10.1145/3571808.

[14] Chandra Chekuri and Chao Xu. Computing minimum cuts in hypergraphs. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 1085–1100. SIAM, 2017. doi: 10.1137/1.9781611974782.70. URL https://doi.org/10.1137/1.9781611974782.70.

[15] Chandra Chekuri and Chao Xu. Minimum cuts and sparsification in hypergraphs. *SIAM J. Comput.*, 47(6):2118–2156, 2018. doi: 10.1137/18M1163865. URL https://doi.org/10.1137/18M1163865.

[16] Chandra Chekuri, Andrew V. Goldberg, David R. Karger, Matthew S. Levine, and Clifford Stein. Experimental study of minimum cut algorithms. In Michael E.

Saks, editor, *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 5-7 January 1997, New Orleans, Louisiana, USA*, pages 324–333. ACM/SIAM, 1997. URL http://dl.acm.org/citation.cfm?id=314161.314315.

[17] Eddie Cheng. Edge-augmentation of hypergraphs. *Math. Program.*, 84(3):443–465, 1999. doi: 10.1007/S101070050032. URL https://doi.org/10.1007/s101070050032.

[18] Joseph Cheriyan and Kurt Mehlhorn. Algorithms for dense graphs and networks on the random access computer. *Algorithmica*, 15(6):521–549, 1996. doi: 10.1007/BF01940880. URL https://doi.org/10.1007/BF01940880.

[19] William H. Cunningham. Decomposition of submodular functions. *Combinatorica*, 3(1):53–68, 1983. doi: 10.1007/BF02579341. URL https://doi.org/10.1007/BF02579341.

[20] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, 2011. doi: 10.1145/2049662.2049663. URL https://doi.org/10.1145/2049662.2049663.

[21] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. ISBN 013215871X. URL https://www.worldcat.org/oclc/01958445.

[22] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Math. Program.*, 91(2):201–213, 2002. doi: 10.1007/S101070100263. URL https://doi.org/10.1007/s101070100263.

[23] Shaddin Dughmi. Submodular functions: Extensions, distributions, and algorithms. A survey. *CoRR*, abs/0912.0322, 2009. URL http://arxiv.org/abs/0912.0322.

[24] Jonathan Eckstein. Parallel branch-and-bound algorithms for general mixed integer programming on the CM-5. *SIAM J. Optim.*, 4(4):794–814, 1994. doi: 10.1137/0804046. URL https://doi.org/10.1137/0804046.

[25] Marcelo Fonseca Faraj, Ernestine Großmann, Felix Joos, Thomas Möller, and Christian Schulz. Engineering weighted connectivity augmentation algorithms. In Leo Liberti, editor, *22nd International Symposium on Experimental Algorithms, SEA 2024, July 23-26, 2024, Vienna, Austria*, volume 301 of *LIPIcs*, pages 11:1–11:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. doi: 10.4230/LIPICS.SEA.2024.11. URL https://doi.org/10.4230/LIPIcs.SEA.2024.11.

[26] Lester R. Ford Jr. and Delbert R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956. doi: 10.4153/CJM-1956-045-5. URL https://doi.org/10.4153/CJM-1956-045-5.

[27] Harold N. Gabow. Path-based depth-first search for strong and biconnected components. *Inf. Process. Lett.*, 74(3-4):107–114, 2000. doi: 10.1016/S0020-0190(00)00051-X. URL https://doi.org/10.1016/S0020-0190(00)00051-X.

[28] Bernard A. Galler and Michael J. Fischer. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, 1964. doi: 10.1145/364099.364331. URL https://doi.org/10.1145/364099.364331.

[29] Michel X. Goemans and V. S. Ramakrishnan. Minimizing submodular functions over families of sets. *Comb.*, 15(4):499–513, 1995. doi: 10.1007/BF01192523. URL https://doi.org/10.1007/BF01192523.

[30] Andrew V. Goldberg and Robert Endre Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988. doi: 10.1145/48014.61051. URL https://doi.org/10.1145/48014.61051.

[31] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961. ISSN 03684245. URL http://www.jstor.org/stable/2098881.

[32] Lars Gottesbüren, Tobias Heuer, and Peter Sanders. Parallel flow-based hypergraph partitioning. In Christian Schulz and Bora Uçar, editors, *20th International Symposium on Experimental Algorithms, SEA 2022, July 25-27, 2022, Heidelberg, Germany*, volume 233 of *LIPIcs*, pages 5:1–5:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi: 10.4230/LIPICS.SEA.2022.5. URL https://doi.org/10.4230/LIPIcs.SEA.2022.5.

[33] Lars Gottesbüren, Tobias Heuer, Nikolai Maas, Peter Sanders, and Sebastian Schlag. Scalable high-quality hypergraph partitioning. *ACM Trans. Algorithms*, 20(1):9:1–9:54, 2024. doi: 10.1145/3626527. URL https://doi.org/10.1145/3626527.

[34] Rahul Raj Gupta and Sushanta Karmakar. Incremental algorithm for minimum cut and edge connectivity in hypergraph. In Charles J. Colbourn, Roberto Grossi, and Nadia Pisanti, editors, *Combinatorial Algorithms - 30th International Workshop, IWOCA 2019, Pisa, Italy, July 23-25, 2019, Proceedings*, volume 11638 of *Lecture Notes in Computer Science*, pages 237–250. Springer, 2019. doi: 10.1007/978-3-030-25005-8_20. URL https://doi.org/10.1007/978-3-030-25005-8_20.

[35] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2025. URL https://www.gurobi.com.

[36] Erez Hartuv and Ron Shamir. A clustering algorithm based on graph connectivity. *Inf. Process. Lett.*, 76(4-6):175–181, 2000. doi: 10.1016/S0020-0190(00)00142-3. URL https://doi.org/10.1016/S0020-0190(00)00142-3.

[37] Zhongtian He, Shang-En Huang, and Thatchaphol Saranurak. Cactus representations in polylogarithmic max-flow via maximal isolating mincuts. In David P. Woodruff, editor, *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024*, pages 1465–1502. SIAM, 2024. doi: 10.1137/1.9781611977912.60. URL https://doi.org/10.1137/1.9781611977912.60.

[38] Zhongtian He, Shang-En Huang, and Thatchaphol Saranurak. Cactus representation of minimum cuts: Derandomize and speed up. In David P. Woodruff, editor, *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024*, pages 1503–1541. SIAM, 2024. doi: 10.1137/1.9781611977912.61. URL https://doi.org/10.1137/1.9781611977912.61.

[39] Vitali Henne. Label propagation for hypergraph partitioning. Master's thesis, 2015. URL https://doi.org/10.5445/IR/1000063440.

[40] Monika Henzinger, Alexander Noe, Christian Schulz, and Darren Strash. Practical minimum cut algorithms. *ACM J. Exp. Algorithmics*, 23, 2018. doi: 10.1145/3274662. URL https://doi.org/10.1145/3274662.

[41] Monika Henzinger, Alexander Noe, and Christian Schulz. Shared-memory exact minimum cuts. In *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, pages 13–22. IEEE, 2019. doi: 10.1109/IPDPS.2019.00013. URL https://doi.org/10.1109/IPDPS.2019.00013.

[42] Monika Henzinger, Alexander Noe, Christian Schulz, and Darren Strash. Finding all global minimum cuts in practice. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 173 of *LIPIcs*, pages 59:1–59:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi: 10.4230/LIPICS.ESA.2020.59. URL https://doi.org/10.4230/LIPIcs.ESA.2020.59.

[43] Michael Jünger, Giovanni Rinaldi, and Stefan Thienel. Practical performance of efficient minimum cut algorithms. *Algorithmica*, 26(1):172–195, 2000. doi: 10.1007/S004539910009. URL https://doi.org/10.1007/s004539910009.

[44] David R. Karger. Minimum cuts in near-linear time. *J. ACM*, 47(1):46–76, 2000. doi: 10.1145/331605.331608. URL https://doi.org/10.1145/331605.331608.

[45] David R. Karger. A randomized fully polynomial time approximation scheme for the all-terminal network reliability problem. *SIAM Rev.*, 43(3):499–522, 2001. doi: 10.1137/S0036144501387141. URL https://doi.org/10.1137/S0036144501387141.

[46] David R. Karger and Clifford Stein. A new approach to the minimum cut problem. *J. ACM*, 43(4):601–640, 1996. doi: 10.1145/234533.234534. URL https://doi.org/10.1145/234533.234534.

[47] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. doi: 10.1007/978-1-4684-2001-2_9. URL https://doi.org/10.1007/978-1-4684-2001-2_9.

[48] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998. doi: 10.1137/S1064827595287997. URL https://doi.org/10.1137/S1064827595287997.

[49] George Karypis and Vipin Kumar. Metis 4.0: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. *Army HPC Research Center, Department of Computer Science & Engineering, University of Minnesota*, 38:7–1, 1998.

[50] George Karypis and Vipin Kumar. hMetis 1.5.3: A hypergraph partitioning package. *Army HPC Research Center, Department of Computer Science & Engineering, University of Minnesota*, 2:1–20, 1998.

[51] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: applications in VLSI domain. *IEEE Trans. Very Large Scale Integr. Syst.*, 7(1):69–79, 1999. doi: 10.1109/92.748202. URL https://doi.org/10.1109/92.748202.

[52] Regina Klimmek and Frank Wagner. A simple hypergraph min cut algorithm. 1996.

[53] Kishore Kothapalli, Sriram V. Pemmaraju, and Vivek Sardeshmukh. On the analysis of a label propagation algorithm for community detection. In Davide Frey, Michel Raynal, Saswati Sarkar, Rudrapatna K. Shyamasundar, and Prasun

Sinha, editors, *Distributed Computing and Networking, 14th International Conference, ICDCN 2013, Mumbai, India, January 3-6, 2013. Proceedings*, volume 7730 of *Lecture Notes in Computer Science*, pages 255–269. Springer, 2013. doi: 10.1007/978-3-642-35668-1_18. URL https://doi.org/10.1007/978-3-642-35668-1_18.

[54] Eugene L. Lawler. Cutsets and partitions of hypergraphs. *Networks*, 3(3):275–285, 1973. doi: 10.1002/NET.3230030306. URL https://doi.org/10.1002/net.3230030306.

[55] Chenghua Liu, Minbo Gao, Zhengfeng Ji, and Mingsheng Ying. Quantum speedup for hypergraph sparsification. *CoRR*, abs/2505.01763, 2025. doi: 10.48550/ARXIV.2505.01763. URL https://doi.org/10.48550/arXiv.2505.01763.

[56] Wai-Kei Mak and D. F. Wong. A fast hypergraph min-cut algorithm for circuit partitioning. *Integr.*, 30(1):1–11, 2000. doi: 10.1016/S0167-9260(00)00008-0. URL https://doi.org/10.1016/S0167-9260(00)00008-0.

[57] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998. doi: 10.1145/272991.272995. URL https://doi.org/10.1145/272991.272995.

[58] David W. Matula. A linear time 2+epsilon approximation algorithm for edge connectivity. In Vijaya Ramachandran, editor, *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, pages 500–504. ACM/SIAM, 1993. URL http://dl.acm.org/citation.cfm?id=313559.313872.

[59] Hiroshi Nagamochi and Toshihide Ibaraki. Computing edge-connectivity in multiple and capacitated graphs. In Tetsuo Asano, Toshihide Ibaraki, Hiroshi Imai, and Takao Nishizeki, editors, *Algorithms, International Symposium SIGAL '90, Tokyo, Japan, August 16-18, 1990, Proceedings*, volume 450 of *Lecture Notes in Computer Science*, pages 12–20. Springer, 1990. doi: 10.1007/3-540-52921-7_51. URL https://doi.org/10.1007/3-540-52921-7_51.

[60] Hiroshi Nagamochi, Tadashi Ono, and Toshihide Ibaraki. Implementing an efficient minimum capacity cut algorithm. *Math. Program.*, 67:325–341, 1994. doi: 10.1007/BF01582226. URL https://doi.org/10.1007/BF01582226.

[61] Hiroshi Nagamochi, Yoshitaka Nakao, and Toshihide Ibaraki. A fast algorithm for cactus representations of minimum cuts. *Japan Journal of Industrial and Applied Mathematics*, 17(2):245, 2000. doi: 10.1007/BF03167346. URL https://doi.org/10.1007/BF03167346.

[62] Manfred Padberg and Giovanni Rinaldi. An efficient algorithm for the minimum capacity cut problem. *Math. Program.*, 47:19–36, 1990. doi: 10.1007/BF01580850. URL https://doi.org/10.1007/BF01580850.

[63] Jean-Claude Picard and Maurice Queyranne. On the structure of all minimum cuts in a network and applications. *Math. Program.*, 22(1):121, 1982. doi: 10.1007/BF01581031. URL https://doi.org/10.1007/BF01581031.

[64] J. Scott Provan and Douglas R. Shier. A paradigm for listing (s, t)-cuts in graphs. *Algorithmica*, 15(4):351–372, 1996. doi: 10.1007/BF01961544. URL https://doi.org/10.1007/BF01961544.

[65] Li Pu and Boi Faltings. Hypergraph learning with hyperedge expansion. In Peter A. Flach, Tijl De Bie, and Nello Cristianini, editors, *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2012, Bristol, UK, September 24-28, 2012. Proceedings, Part I*, volume 7523 of *Lecture Notes in Computer Science*, pages 410–425. Springer, 2012. doi: 10.1007/978-3-642-33460-3_32. URL https://doi.org/10.1007/978-3-642-33460-3_32.

[66] Maurice Queyranne. Minimizing symmetric submodular functions. *Math. Program.*, 82:3–12, 1998. doi: 10.1007/BF01585863. URL https://doi.org/10.1007/BF01585863.

[67] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76 (3), 2007. doi: 10.1103/physreve.76.036106. URL https://link.aps.org/doi/10.1103/PhysRevE.76.036106.

[68] Peter Sanders and Christian Schulz. Engineering multilevel graph partitioning algorithms. In Camil Demetrescu and Magnús M. Halldórsson, editors, *Algorithms - ESA 2011 - 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings*, volume 6942 of *Lecture Notes in Computer Science*, pages 469–480. Springer, 2011. doi: 10.1007/978-3-642-23719-5_40. URL https://doi.org/10.1007/978-3-642-23719-5_40.

[69] Linus Schrage and Kenneth R. Baker. Dynamic programming solution of sequencing problems with precedence constraints. *Oper. Res.*, 26(3):444–449, 1978. doi: 10.1287/OPRE.26.3.444. URL https://doi.org/10.1287/opre.26.3.444.

[70] Yuji Shinano, Stefan Heinz, Stefan Vigerske, and Michael Winkler. Fiberscip - A shared memory parallelization of SCIP. *INFORMS J. Comput.*, 30(1):11–30, 2018. doi: 10.1287/IJOC.2017.0762. URL https://doi.org/10.1287/ijoc.2017.0762.

[71] Christian Staudt and Henning Meyerhenke. Engineering high-performance community detection heuristics for massive graphs. In *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*, pages 180–189. IEEE Computer Society, 2013. doi: 10.1109/ICPP.2013.27. URL https://doi.org/10.1109/ICPP.2013.27.

[72] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *J. ACM*, 44(4):585–591, 1997. doi: 10.1145/263867.263872. URL https://doi.org/10.1145/263867.263872.

[73] Ole Tange. GNU parallel: The command-line power tool. *login Usenix Mag.*, 36(1), 2011. URL https://www.usenix.org/publications/login/february-2011-volume-36-number-1/gnu-parallel-command-line-power-tool.

[74] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975. doi: 10.1145/321879.321884. URL https://doi.org/10.1145/321879.321884.

[75] Nate Veldt, Austin R. Benson, and Jon M. Kleinberg. Hypergraph cuts with general splitting functions. *SIAM Rev.*, 64(3):650–685, 2022. doi: 10.1137/20M1321048. URL https://doi.org/10.1137/20m1321048.

[76] Natarajan Viswanathan, Charles J. Alpert, Cliff C. N. Sze, Zhuo Li, and Yaoguang Wei. The DAC 2012 routability-driven placement contest and benchmark suite. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, pages 774–782. ACM, 2012. doi: 10.1145/2228360.2228500. URL https://doi.org/10.1145/2228360.2228500.

[77] Michael Walter and Freek Witteveen. Hypergraph min-cuts from quantum entropies. *Journal of Mathematical Physics*, 62(9), 2021. doi: 10.1063/5.0043993. URL https://doi.org/10.1063/5.0043993.

[78] Yutaro Yamaguchi, Anna Ogawa, Akiko Takeda, and Satoru Iwata. Cyber security analysis of power networks by hypergraph cut algorithms. *IEEE Trans. Smart Grid*, 6(5):2189–2199, 2015. doi: 10.1109/TSG.2015.2394791. URL https://doi.org/10.1109/TSG.2015.2394791.