# Engineering Scalable Algorithms for Isolating Cut

Olga Sergeyeva

March 26, 2026

4208424

Bachelor Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

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

Co-Referee:
M. Sc. Adil Chhabra

# Acknowledgments

I would like to express my sincere gratitude to Prof. Dr. Christian Schulz, who introduced me to the field of Algorithm Engineering and has thus been a clear inspiration in his field of research. It was a pleasure learning from him in various lectures and practicals, and has thus been a clear highlight during my degree. I would also like to expressively thank him for the opportunity of writing this thesis under his supervision. I would like to thank Adil Chhabra for his consistent, reliable and excellent help during the process of the project, which has been an invaluable addition to this thesis. Additionally I want to express gratitude to everyone who has helped me throughout the process of both my thesis, be it proofreading or just sharing a laugh over a cup of coffee, as well as my degree. It would not have been as fun nor as rewarding without all the people along the way.

I hereby confirm, Olga Sergeyeva, 4208424, that this thesis has been written independently and without unauthorised external assistance. I have not used any sources or aids other than those specified, and I have identified all passages that have been taken verbatim or paraphrased from published or unpublished works, including those from digital or AI-based sources. I confirm that any use of AI tools was discussed in advance with the supervisor of the thesis and that the agreed rules were followed. I take full responsibility for the academic quality and content of this paper, the methodology chosen and the process of its creation, as well as the literature cited. I agree that my thesis will be checked for plagiarism and possible misuse of automated text and code generation as part of university procedures.

Heidelberg, March 26, 2026

Olga Sergeyeva

# Abstract

The MULTITERMINAL-CUT PROBLEM asks for a minimum-weight set of edges whose removal separates a given set of terminals pairwise. Since the problem is NP-hard for three or more terminals, this thesis focuses on the scalable computation of minimum isolating cuts in weighted undirected graphs as a practical basis for multiterminal separation.

We study three maximum-flow based approaches for computing the family of isolating cuts for a terminal set $R$ of size $k$. As a direct baseline, one minimum cut is computed for each terminal, resulting in $k$ maximum-flow computations on the full graph. Building on the structural ideas behind the Isolating Cut Lemma of Li and Panigrahi, a second approach first computes $\lceil \log_2 k \rceil$ group cuts, uses them to localise each terminal to a reduced instance, and then recovers the isolating cuts. A third approach extends this idea by combining the final recovery stage into one disjoint-union maximum-flow computation.

All approaches are implemented in sequential and parallel form and evaluated for terminal counts from $2$ to $128$ on benchmark instances ranging from small graphs with only a few thousand vertices to very large real-world instances. The results show a clear trade-off between simple direct methods and more structured localisation methods. For small terminal sets, the direct approach is often fastest as it avoids additional overhead. For larger terminal counts, however, the localisation-based variants scale substantially better. Across the complete high-$k$ benchmark cases with $k \in 32, 64, 128$, the better of the two structured variants is faster than the direct baseline on $98.1\%$ of the cases, and at $k = 128$ improves runtime by a factor of $3.53$ on average. Batching the recovery phase provides an additional but smaller benefit, while the parallel variants further reduce runtime, achieving mean speedups from one to eight threads at $k = 128$ of $3.83$ for PTERMCUT, $2.08$ for PISOCUT, and $2.06$ for PBATCHISOCUT. Overall, the experiments show that no single method dominates across all settings, instead, the most suitable algorithm depends on the terminal count, the graph instance and whether the main objective is low sequential overhead or stronger shared-memory scalability.

# Contents

CHAPTER 1

# Introduction

## 1.1 Motivation

Graphs have become a pivotal tool often used to model versatile and complex interconnected systems. Be it telecommunications, protein structures or critical system architecture, there is more often than not a need to be able to work with specific points of interest, and influence them as needed. Having the ability to completely separate these focal points from each other without leaving any path to one another is thus a recurring problem. Consider a subset of critical entities in a system of sensitive data. If an unauthorised actor were to breach that system and gain access to said sensitive data, the operators would want to contain the damage as soon as the breach went noticed. Therefore being able to isolate the system into mutually unreachable components is an important mechanism for keeping a critical situation contained. The *multiterminal-cut* deals with exactly that problem.

Given a graph $G = (V, E)$ and a subset $R \subseteq V$ of vertices of size $k$, called *terminals*, we intend to separate all terminals pair-wise whilst keeping the weight of the removed edges minimal. For $|R| = 2$ this is equivalent to the classical $s - t$ minimum-cut problem, and is a quite thoroughly researched topic, while being solvable in polynomial time. For $|R| \geq 3$, however, the problem becomes NP-hard, making it far more difficult to solve efficiently.

In this thesis, we focus on isolating cuts as a practical basis for terminal separation. A direct approach computes one minimum isolating cut for each terminal and therefore requires $k$ maximum-flow computations. This becomes increasingly expensive as the number of terminals grows. To address this problem, we use the Isolating Cut Lemma as the structural basis of our algorithms. Informally, the lemma states that the family of isolating cuts for a terminal set can be derived by using only a logarithmic number of maximum-flow computations, followed by a recovery step that reconstructs the individual cuts. This makes it possible to reduce the amount of full-graph cut computations while preserving correctness.

## 1.2 Our Contribution

In this thesis, we study the practical computation of minimum isolating cuts in weighted undirected graphs $G = (V, E, \omega)$ for a given terminal set $R \subseteq V$ with $|R| = k$. Rather than designing a new exact algorithm for the minimum multiterminal-cut problem itself, we focus on isolating cuts as the central algorithmic object and investigate how they can be computed efficiently in practice.

To this end, we implement and compare three maximum-flow-based approaches. The first approach serves as a direct baseline and computes one minimum isolating cut for each terminal independently on the full graph. The second approach follows the structural ideas of Li and Panigrahi [21] by first computing a logarithmic number of group cuts, using them to localise each terminal to a reduced instance, and then recovering the isolating cuts on these smaller graphs. The third approach builds on the same localisation phase but combines the final recovery step into one batched maximum-flow computation on a disjoint-union instance.

All three approaches are integrated into the KAHIP framework [17] and implemented in both sequential and shared-memory parallel form. Based on these implementations, we conduct an extensive experimental evaluation across a broad benchmark set, analysing runtime scaling, phase-wise behaviour, shared-memory speedup, thread imbalance, peak memory usage and the practical trade-offs between the different algorithmic designs.

## 1.3 Structure

The remainder of this thesis is organized as follows. Chapter 2 introduces the theoretical backbone of this work, by first going into detail of all algorithmic and graphical foundations needed, followed by an introduction into the formal definition of the problem to be solved and concluding with the concepts used throughout. Chapter 3 reviews related work and places this thesis into context of current research. Chapter 4 presents the main theoretical body of the thesis, explaining the intuition behind the Isolating Cut Lemma, its role in presented algorithms and corresponding correctness proofs. Chapter 5 then deals with the methods used to engineer the scalability of the isolating cut algorithms, by focusing on each variation of code, comparing their effectiveness and theory behind them, as well as highlighting important differences. Chapter 6 presents the experimental evaluation of the proposed methods. Chapter 7 concludes the work by discussing the results, as well as outlining directions for future work.

CHAPTER 2

# Fundamentals

This chapter starts out by introducing the fundamentals and notation used, followed by giving a breakdown of all necessary algorithmic foundations and problem statements which are considered and worked with further on.

## 2.1 General Definitions

**Graph Structure.** Let $G = (V, E, \omega)$ be an *undirected weighted graph*, with $V$ a finite set of vertices, $E \subseteq \{\{u, v\} \mid u, v \in V,\ u \neq v\}$ defining an edge set of unordered pairs of distinct vertices and $\omega : E \to \mathbb{N}_0$ being a non-negative cost function. We denote for the rest of this work $|V| =: n$ as the number of vertices and $|E| =: m$ as the number of edges in $G$. For all edges $e \in E$ with $e = \{u, v\}$, we declare $u, v$ as vertices incident to the edge $e$ with them being the endpoints of $e$. The degree of a vertex $v \in V$, formally written as $\deg(v)$, equals the amount of edges incident to $v$. Any two vertices are called *neighbours*, if there exists an edge directly connecting them. Neighboured vertices are often also referred to as being adjacent to one another. Following above definition, we limit ourselves to working with simple graphs. Thus we eliminate the possibility of parallel edges running between vertices and self-loops, i.e. connecting a vertex to itself with an edge.

For any sequence $p = (v_1, ..., v_k)$, we declare $p$ a *path*, if the vertices are connected via a subset of edges with $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k - 1$. Additionally, when considering a *simple path*, the vertices and edges are considered to be distinct. If not further specified, we restrict the paths talked about to simple paths. The length of a path $p$ equals the amount of edges along the path. For a path $p = (u, ..., v)$, with $u$ and $v$ respectively being either start- or endpoint, we refer to it as a $u - v$-path. If there exists a $u - v$-path for every pair of vertices $u, v \in V$, then $G$ is considered a *connected* graph.

A graph $G' = (V', E')$ is a subgraph of $G$, if $V' \subseteq V$ and $E' \subseteq E$, with every edge having both its endpoints in $V'$. If $E'$ consists of all edges in $E$ with both endpoints in $V'$, then $G[V'] := (V', E')$ is an *induced subgraph* of $G$. An induced subgraph is considered

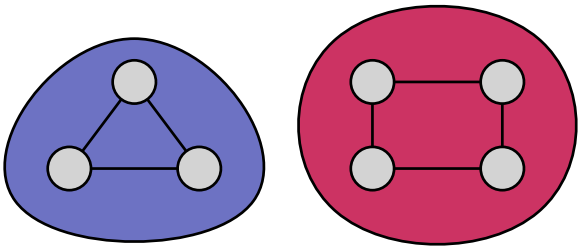

**Figure 2.1:** Example of a graph consisting of two connected components. Each component forms a maximally connected induced subgraph, in which each pair of vertices is connected by a path.

*maximally connected*, if there is no further vertex in $G$ which can be added without losing connectedness. Any induced subgraph that also happens to be maximally connected is called a *connected component* of $G$.

We define a *contraction* on a graph as a graph operation in which a set of vertices is replaced by a single new super-vertex. More precicely, let $G = (V, E, \omega)$ be an undirected weighted graph and let $X \subseteq V$ be non-empty. When contracting $X$ we thus remove the vertices in $X$, and introduce the new vertex $x_X$. For an edge with one endpoint in $X$, it gets replaced by an edge incident to $x_X$. Edges with both endpoints in $X$ become self-loops and are therefore discarded after contraction. If multiple edges connect the same pair of vertices after contraction, then they are merged into one edge whose weight is the sum of the original edge weights.

As a special case of this contraction for a subset $U \subseteq V$, we define a $U$-contracted Graph $G_U = (V_U, E_U)$ as follows. All Vertices $v \in U$ remain unchanged and thus belong to $V_U$. For every connected component $C_i$ of the induced subgraph $G[V \setminus U]$, we introduce a super-vertex $v_{C_i}$, thus resulting in the contracted vertex set.

$$V_U := U \cup \{v_{C_i} \mid C_i \text{ is a connected component of } G[V \setminus U]\}.$$

For $E_U$, we insert any edge with both endpoints being in $U$. If there exists an edge with only one endpoint in $U$ and the other endpoint being in a connected component $C_k$, we connect the vertex in $U$ with $v_{c_k}$. Edges which have neither endpoint in $U$ lie within some connected component, therefore becoming self-loops on that component after contraction. Any self-loops are discarded after contraction, as we are working with simple graphs. Thus resulting in following edge set

$$E_U := \{\{u, v\} \mid \{u, v\} \in E \text{ with } u, v \in U\} \cup \{\{u, v_{C_k}\} \mid u \in U,\ \exists w \in C_k : \{u, w\} \in E\}.$$

If, following the contraction, the graph contains multiple edges inbetween any pair of nodes, the sum of the edge-weights is added up and those multiple edges are replaced by one edge connecting the components.

A *directed graph* $G = (V, E, \omega)$ is defined analogously to an undirected graph, with the exception that the edge set now consists of ordered pairs of vertices. Self-loops and

parallel edges are still not permitted, however there may exist an edge $(u, v)$ and $(v, u)$ for any two pairs of vertices. If all edges $(u, v)$ have a counter-part $(v, u)$ the directed graph can be represented as an undirected graph. When talking about graphs, we mean undirected graphs if not differently specified.

**Flow Networks.** A *flow network* is defined as $F = (G, c, s, t)$, with $G$ as an undirected Graph, $c$ a capacity-function $c : E \to \mathbb{N}_0$ and $s, t \in V$ of $G$ as dedicated source and sink vertices. When defining flows on an undirected graph, for each undirected edge $e = \{u, v\}$ we treat it as two directed edges $(u, v)$ and $(v, u)$ with the same capacity.

We define a *flow* on the network as a function $f : E \to \mathbb{N}_0$ with following properties. For any flow $f$ of an edge $e$, it must uphold the *capacity constraint*, defined as

$$0 \leq f_e \leq c_e.$$

Per *the conservation constraint* the incoming flow ($f_v^-$) must equal the outgoing flow ($f_v^+$),

$$f_v^- := \sum_{(u,v)\in E} f_{(u,v)} \quad \text{and} \quad f_v^+ := \sum_{(v,w)\in E} f_{(v,w)}$$

of any vertex $v \in V \backslash \{s, t\}$, such that

$$f_v^- - f_v^+ = 0.$$

The *value* of a flow $f$ is defined as

$$\text{val}(f) := \sum_{(s,v)\in E} f_{(s,v)}$$

with $s$ being the source vertex of $F$. A flow is a *maximum flow*, if there is no higher feasible flow value.

Given a flow on a flow network $F$, we define the *residual graph* $G_f = (V, E_f)$ derived from $F$ by representing the available capacity for modifying the current flow. For any directed edge $(u, v) \in E$ the *residual capacity* is defined as

$$c_f(u, v) := c(u, v) - f(u, v)$$

and being non-negative by capacity constraint. If $c_f(u, v) > 0$ then $(u, v)$ is in $E_f$. Furthermore, the residual graph has an opposing edge $(v, u) \in E_f$ with

$$c_f(v, u) := f(u, v),$$

for any edge with $f(u, v) > 0$ in the original network $F$. The residual graph is used by max-flow algorithms to iteratively improve the flow value by either increasing or decreasing the current flow of an edge.

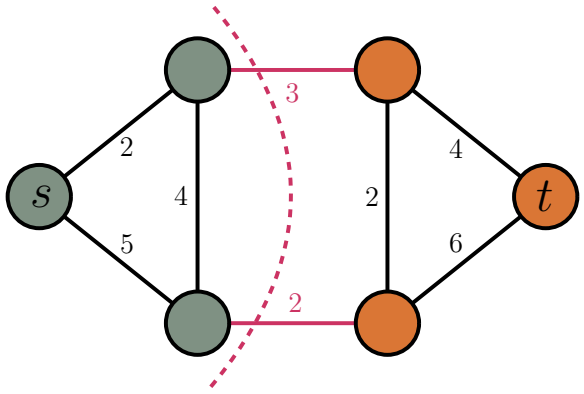


**Figure 2.2:** An Illustration featuring an $s-t$ cut on a flow network. The vertex set is partitioned into $S$ (green) and $\bar{S}$ (orange) such that $s \in S$ and $t \in \bar{S}$. Edges crossing the dashed partition form the cut-set $\delta(S)$ with cut capacity 5, represented in pink.

**Cuts.** Let $G = (V, E)$ be a graph, we define a *cut* $C = (S, \bar{S})$ on $G$, as a bipartition of the vertex set $V$ into two disjoint sets $S$ and $\bar{S} = V \setminus S$. The Cut-Set is the set of edges with

$$\delta(S) := \{(u, v) \in E \mid u \in S, v \in \bar{S}\}.$$

If $s \in S$ and $t \in \bar{S}$ are specifically defined vertices, we call the cut an $s-t$ cut. The weight of a cut in an unweighted graph is defined as the number of edges in the cut-set. If $G$ has edge weights, we consider

$$\omega(\delta(S)) := \sum_{e \in \delta(S)} \omega(e)$$

to be the cut capacity. A *minimum cut* is a cut which is considered minimal in its cut capacity in comparison to any other possible cut.

The minimum cut problem is closely related to the maximum flow problem. Given a flow network $F = (G, c, s, t)$, we can use any such flow algorithm to compute the min-cut of $s-t$.

**Theorem 1 (Max-Flow Min-Cut)**
*For any flow network, the value of the maximum $s-t$ flow of $F$ equals the capacity of the minimum $s-t$ cut.*

Computing maximum flows therefore provides an efficient way to determine minimum cuts in graphs as proven by Ford and Fulkerson in 1968 [15]. The residual graph of a finished max-flow computation can be used to determine the min-cut of the graph. Let $S$ be the subset of vertices reachable from $s$ in the residual graph. Thus the cut $C = (S, V \setminus S)$ is a min-cut.

**Max-Flow Algorithms.** Over the years numerous algorithms have been developed to compute maximum flow in networks. Early methods include Ford and Fulkerson's [15]

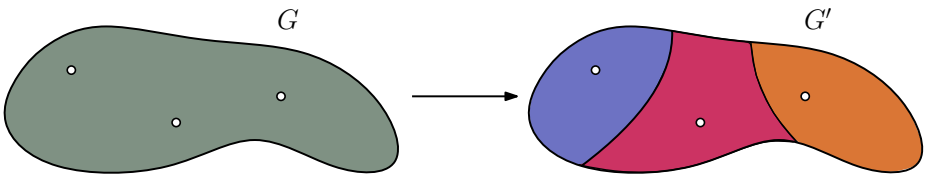


**Figure 2.3:** Example of a multiterminal cut on $G$, that separates the graph into three connected components, each containing exactly one terminal. The edges running in-between the partitions form the multiterminal cut.

idea of sending flow along any path between the source and the sink that still has available capacity. These paths are called *augmenting paths*. For integral capacities, this method runs in $O(m \cdot v_{\text{mf}})$ with $m$ being the amount of edges and $v_{\text{mf}}$ being the value of the maximum-flow. Edmonds and Karp later proposed a fully formulated idea of such an augmenting-path algorithm, by choosing the shortest possible augmenting path, making it feasible in polynomial time [14]. This yields a runtime of $O(nm^2)$, with $n$ the number of vertices and $m$ the number of edges. Dinic improved this method further by allowing several augmenting paths to be processed within a computation phase, resulting in a runtime of $O(n^2m)$ in the general case[10, 12].

All these methods focus on consistently upholding the conservation constraint introduced earlier. However, there are further algorithms that can be used to compute max-flow, based on a different principle. Goldberg and Tarjan introduced an approach which allows the algorithm to temporarily violate the conservation constraint by allowing excess flow for intermediate vertices [16]. Instead of a *feasible* flow we therefore maintain a *preflow*. Additionally, we assign *distance labels* to vertices to approximate the distance from any vertex $v \in V$ to the sink vertex $t$. The preflow computations work by repeatedly sending flow to neighbouring vertices with a smaller label. When a vertex has excess flow it is then redistributed further on with *push* operations. If no further push operations are viable, we use *relabel* operations to increase vertex labels, to allow further push operations. As soon as no vertex other than the sink and source has any excess flow left, the algorithm is terminated and the maximum flow has been computed. In its generic form the push-relabel algorithm runs in $O(n^3)$.

## 2.2 Multiterminal Cut

**Multiterminal-Cut Problem.** Let $G = (V, E, \omega)$ be a weighted graph. We define a distinct subset of $k$ vertices with $R = \{r_1, ..., r_k\} \subseteq V$ and subsequently call them *terminals*. Then the *multiterminal-cut* defines a subset of edges $E'$ which separates all terminals pair-wise from one another, thus leaving no path between any two terminals. The problem can also be restated as a partitioning problem. For $k$ terminals we intend to find $k$ disjoint subsets $(V_1, ..., V_k)$ of the vertex-set $V$ such that each subset contains exactly one terminal.

The goal is then to minimize the weight of the cut-edges running in-between different parts of the partition. Equivalently, we can define

$$E_\times := \big\{\{u,v\} \in E \mid u \in V_i,\ v \in V_j,\ i \neq j\big\}$$

as the edge-set with endpoints lying in distinct subsets. The objective is then to minimize

$$\omega(E_\times) = \sum_{e \in E_\times} \omega(e).$$

Intuitively, the multiterminal cut problem intends to separate $k$ terminals from one another by removing edges of total minimum weight. For $k = 2$ the multiterminal-cut problem reduces to a classical minimum $s - t$ cut-problem, by assigning respectively one terminal as sink and the other terminal as source vertex. However, for $k \geq 3$, the problem becomes structurally more difficult, therefore requiring more sophisticated methods to compute such cuts.

**Isolating Cuts.** With the computational difficulty assigned to the multiterminal cut problem, many algorithms attempt to reduce the problem to a sequence of simpler minimum cut computations. One method for computing multiterminal-cuts is called *Isolating Cuts*. For a terminal $r_i \in R$ an *isolating cut* is defined as the minimum-cut separating the current terminal $r_i$ from all other terminals in $R \setminus r_i$. To compute such a cut we contract all remaining terminals into one super-terminal $t_\infty$ and compute a minimum-cut on $(r_i, t_\infty)$. Thus, the resulting problem can be reduced to an ordinary $s-t$ minimum-cut computation. Isolating Cuts form an important structural base in several terminal-separation algorithms and constitute the main focus and concept studied in this thesis.

CHAPTER 3

# Related Work

Graph cut problems embody an important subset of research in algorithm engineering and combinatorial optimization. Minimum cut and minimum $s - t$ cut problems have been studied extensively over the years, yielding efficient algorithms based on maximum-flow computations [11, 14, 15, 16]. However, in many practical settings it is necessary to separate more than just two designated vertices, resulting in more complex terminal-separation problems. These quickly become computationally difficult and require sophisticated and specifically adapted structural techniques to find a solution.

In this thesis, the main algorithmic object of interest is the scalable computation of isolating cuts, while multiterminal cuts serve as a natural application setting. This chapter therefore reviews related work on multiterminal cuts, isolating cuts, maximum-flow based methods and frameworks relevant to our implementation.

## 3.1 Multiterminal Cut and Terminal Separation

The multiterminal-cut problem asks for a minimum-weight set of edges whose removal separates all terminals pairwise from one another. In one of the foundational results of this area Dahlhaus et al. [8] showed that the multiterminal cut for two terminals is solvable in polynomial time, as it reduces to the classical minimum $s - t$ cut problem, while being NP-hard for three or more terminals. In the same work they established an $(2 - \frac{2}{k})$ approximation of the optimal cut, which remains one of the most prominent advancements in the field.

Consequently, later research has focused primarily on finding better approximation guarantees and deeper structural insights of multiterminal-cuts, rather than finding exact polynomial-time algorithms. On the approximation side, Călinescu et al. [7] improved the classical guarantee by proposing an $(\frac{3}{2} - \frac{1}{k})$ approximation factor based on a new linear programming relaxation. More specifically, they take advantage of the structural properties of the cut problem by introducing a suitable system of inequalities. Later work continued

to study this relaxation and its limitations, for example by improving the integrality-gap results for the Călinescu–Karloff–Rabani relaxation [2].

On the other hand, progress for exact solutions has mainly been achieved in the field of parametrized algorithms. Cao et al. [5] propose an $O^*(1.84^k)$ parametrized algorithm for the multiterminal cut problem, thus breaking its previous $2^k \cdot n^{O(1)}$ barrier. Essentially they use fundamental methods of *maximum volume minimum $s-t$ cuts* as introduced by Ford and Fulkerson in 1962 [15] and Isolating Cuts by Dahlhaus et al [8], coupled with submodular arguments and a redefined branching strategy.

Throughout this research area a recurring theme is reducing the multiterminal cut to repeated computations of minimum-cut. This thesis follows a similar approach. Rather than proposing a new exact algorithm for the minimum multiterminal-cuts problem, we study how isolating cut computations can be engineered efficiently and then used to construct multiterminal cuts in practice. We focus on ensuring that distinct terminals end up in disjoint components from one another, which is especially helpful in applications where a certain number of components or sensitive subsystems must end up completely separated from one another.

## 3.2 Practical Exact Solvers for Multiterminal Cut

While much of the theoretical work on multiterminal cut focuses either on approximation guarantees or parameterised complexity, there has also been progress on practically effective exact solvers. In particular, Velednitsky and Hochbaum [26, 27] introduced *Isolation Branching*, a branch-and-bound algorithm for the $k$-terminal cut problem. Their method does not rely on a mixed-integer programming formulation directly, but instead uses isolating cuts as a structural ingredient inside the branching process. Thus, isolating cuts are not only relevant as approximation tools or as subroutines in parameterised algorithms, but also appear in exact practical approaches for multiterminal separation.

This line of work is especially relevant for our evaluation, as it provides a practical exact baseline for comparison. In contrast to our approach, which focuses on computing the family of isolating cuts efficiently and using them to induce a feasible multiterminal cut, Isolation Branching aims directly at solving the minimum $k$-terminal cut problem exactly. Velednitsky and Hochbaum report that their implementation performs competitively in practice and scales more effectively than a branch-and-bound approach based on a strong integer-programming formulation [26].

Accordingly, this solver is a meaningful external point of reference for our experiments, however, the algorithmic objective remains different from the one pursued in this thesis. Our implementation is not designed as an exact solver for multiterminal cut, but rather as a scalable engineering study of isolating-cut computation itself. Therefore, when compared experimentally, the exact solver should be interpreted as a benchmark for solution quality and exact optimization, whereas our methods are intended as practical approaches for computing isolating cuts and the terminal-separating solutions induced by them.

## 3.3 Isolating Cuts

An isolating cut for a terminal $r_i$ is a minimum cut that separates $r_i$ from all other terminals in the graph. Intuitively, such a cut can be computed by contracting all remaining terminals into one singular super-terminal and then computing the resulting minimum cut. Thus reducing a multiterminal-problem to an ordinary two-terminal cut problem.

Isolating cuts appear as an important structural tool in approximation algorithms such as that of Dahlhaus et al., and they also play a role in later work for exact algorithms as with the earlier mentioned algorithm by Cao et al. [5]. More recently, Li and Panigrahi [21] further improved such methods by introducing the *Isolating Cut Lemma*, discussed in more detail in Chapter 4. This lemma shows that for a terminal set $R$ all minimum isolating-cuts can be computed in $O(\log |R|)$ maximum-flow computations, instead of having to compute it for each terminal. Accordingly, they were able to obtain deterministic algorithms for global minimum cut and Steiner cut with only polylogarithmically many maximum-flow computations. These results are particularly relevant as they shift the algorithmic perspective on isolating cuts. Primarily used as strategic tools and subroutines in algorithms of Dahlhaus et al. and Cao et al., the work of Li and Panigrahi shows that the computation of isolating cuts itself has additional structural properties which can be exploited algorithmically beyond its original setting. The isolating-cut procedure also appears in later work on faster Gomory–Hu tree construction by Abboud et al [1] and by Li and Panigrahi [20], Chekuri et al. [6] generalize the isolating-cut approach to the bisubmodular setting, thus giving way to a new look into graph connectivity problems.

Aligning with this line of work, we treat isolating cuts as the central algorithmic tool in this thesis. Moreover, we investigate how isolating-cut computations inspired by these results can be implemented efficiently, scaled in practice and then be evaluated.

## 3.4 Maximum Flow Computations

Although multiterminal cuts and terminal separation are the main focus of this work, maximum-flow algorithms provide the computational basis for all methods used. As minimum $s - t$ cuts can be computed by running maximum-flow algorithms, the practical performance of terminal-separation algorithms not only relies on their algorithmic structure, but also on the efficiency of the underlying flow routine.

Many classical approaches such as Ford-Fulkerson [15], Dinic [11, 12] and Edmonds-Karp [14] establish the theoretical foundation for flow-based cut computations. Push-relabel methods introduced by Goldberg and Tarjan [16] are among the most relevant approaches and are thereby often used in practice. Maximum-flow based computation is used as a fixed backend subroutine, instead of being a primary focus. Thus we use the push-relabel implementation provided in our framework and focus on the surrounding algorithmic structure, by organizing the computations efficiently and reducing their total number.

CHAPTER 4

# The Isolating Cut Lemma

This chapter lays out the structural foundations underlying the algorithms evaluated in this thesis. It aims to prepare for following chapters by exploring the Isolating Cut Lemma as defined by Li and Panigrahi [21] to keep this work self-contained.

## 4.1 Intuition

A direct way to compute the family of isolating cuts for a terminal set $R$ of size $k$ is to solve one minimum cut problem for each terminal separately. Essentially, for every terminal $v \in R$, one computes a minimum cut separating $v$ from remaining terminals $R \setminus \{v\}$. Executing this for all terminals yields the complete family of minimum isolating cuts, therefore requiring exactly $k$ maximum-flow computations. Consequently, if the running time of one such maximum-flow computation is denoted by $T_{\mathrm{mf}}$, the overall running time is of the order of $k \cdot T_{\mathrm{mf}}$, up to additional lower-order work of constructing and extracting the individual cut instances. As the amount of computations needed for solving this grows linear in accordance with the number of terminals $k$, this approach quickly becomes increasingly expensive for larger values of $k$ and thus is not well-suited for larger practical instances.

Li and Panigrahi addressed this arising bottleneck, by showing that the different minimum isolating cuts are not completely independent from each other. Instead they share structural information as they are all derived from the same underlying graph. When computing the minimum isolating cut for one terminal, we not only separate the current terminal from all other remaining terminals, but also reveal which regions of the graph cannot contain any more of them. Therefore, one cut computation already restricts the possible location of other isolating cuts. The key idea is therefore to reuse the structural information obtained from previous cuts instead of handling each terminal as a separate individual problem instance. The unions and intersections of these cuts satisfy useful properties and are captured formally through the *submodularity* of the cut function. This forms the basis of the Isolating Cut Lemma which will be discussed next.

## 4.2 Structural Properties

We start out by formalising the structural properties of minimum isolating cuts.

**Definition 1 (Minimum Isolating Cuts)**
*Let $G = (V, E, \omega)$ be an undirected weighted graph and let $R \subseteq V$ be a subset of vertices of size $k := |R| \geq 2$. For each $v \in R$, let $S_v$ denote the side containing $v$ of a minimum cut separating $v$ from $R \setminus \{v\}$. For a terminal $v \in R$ we denote by*

$$\lambda_v := \omega(\delta(S_v))$$

*the value of the corresponding minimum isolating cut. The collection*

$$\{S_v : v \in R\}$$

*is called the family of minimum isolating cuts for $R$.*

As defined, every set $S_v$ contains simply the terminal $v$ and no further terminal in $R$. Therefore, the cut set $\delta(S_v)$ separates $v$ from all terminals in $R \setminus \{v\}$ and has minimum weight among all such cuts. Immediately following above definition, we can ascertain these properties for any minimum isolating cut.

**Proposition 1 (Properties of Minimum Isolating Cuts)**
*For any terminal $v \in R$, the set $S_v$ satisfies following properties:*

*(i)* $v \in S_v$,

*(ii)* $S_v \cap R = \{v\}$,

*(iii)* $\omega(\delta(S_v)) = \lambda_v$.

*Proof.* The first property follows directly from the definition of $S_v$ as the side containing $v$. The second property holds as the set $S_v$ is defined upon a minimum isolating cut separating $v$ from any other terminal. Thus if $S_v \cap R \neq \{v\}$, the cut of $S_v$ would not have been a minimum isolating cut, thus contradicting our assumption. The third property follows from the definition of $\lambda_v$.

□

Hence, the family $\{S_v : v \in R\}$ contains one minimum terminal-separating side for each terminal in $R$. A direct way to compute this family is to solve one minimum cut problem for each terminal $v \in R$, where $v$ is separated from the remaining terminals $R \setminus \{v\}$. The Isolating Cut Lemma shows that this number can be reduced substantially by exploiting further structural properties of these cuts.

## 4.3 Submodularity

The Isolating Cut Lemma relies on a fundamental property of the cut function, specifically submodularity. The cut function often serves as a standard example of a submodular set function [22]. For completeness, we include a short self-contained proof adapted to our notation.

**Definition 2 (Submodular Set Function)**
*Let $U$ be a finite set. We define a set function as $f : 2^U \to R_{\geq 0}$. If for any $X, Y \subseteq U$ it holds that*

$$f(X) + f(Y) \geq f(X \cup Y) + f(X \cap Y)$$

*then $f$ is a submodular set function.*

We consider the cut function of a graph as the aforementioned set function here. For a subset $X \subseteq V$, let

$$\delta(X) := \big\{\{u, v\} \in E \mid u \in X,\ v \in V \setminus X\big\}$$

denote the set of edges crossing the cut induced by $X$. Its weight is given by

$$\omega(\delta(X)) = \sum_{e \in \delta(X)} \omega(e).$$

Let $c(X)$ denote the cut function of a set $X$ with $c(X) := \omega(\delta(X))$.

**Lemma 1 (Submodularity)**
*For any two sets $X, Y \subseteq V$, the cut function satisfies following inequality:*

$$c(X) + c(Y) \geq c(X \cup Y) + c(X \cap Y)$$

*Proof.* Let

$$A = X \cap Y, \quad B = X \setminus Y, \quad C = Y \setminus X, \quad D = V \setminus (X \cup Y).$$

denote four disjoint regions forming a partition of $V$. Hereby, $A$ represents the vertices in both subsets, $B$ the vertices in only $X$, $C$ only the vertices in $Y$ and $D$ the vertices in neither subset. With respect to the partition, the cuts induced by $X$, $Y$, $X \cup Y$ and $X \cap Y$ only differ in how edges between $B$ and $C$ are counted. Every edge in $E(B, C)$ is counted once for both cut functions $c(X)$ and $c(Y)$ but not counted at all for $c(X \cup Y)$ and $c(X \cap Y)$. Hence,

$$c(X) + c(Y) = c(X \cup Y) + c(X \cap Y) + 2\omega(E(B, C)).$$

As the edges were defined with non-negative weights, we know that $\omega(E(B, C)) \geq 0$. Therefore,

$$c(X) + c(Y) \geq c(X \cup Y) + c(X \cap Y).$$

Thus proving that the cut function is submodular.

□

Informally, this means that if two cuts overlap in a graph, then the weight of these cuts has to be at least of the weight of the cuts induced by their union and intersection.

## 4.4 Isolating Cut Lemma

The family of minimum isolating cuts can be obtained by solving, for each terminal $v \in R$, a minimum cut separating $v$ from $R \setminus \{v\}$. This requires $k = |R|$ maximum-flow computations. The Isolating Cut Lemma shows that this number can be reduced substantially.

**Lemma 2 (The Isolating Cut Lemma)**
*Let $G = (V, E, \omega)$ be an undirected weighted graph. For a subset $R \subseteq V$ of size at least 2, there exists an algorithm computing the minimum isolating cuts for $R$ using $\lceil \log |R| \rceil + 1$ maximum-flow computations for an $s - t$ problem. Outside of the maximum-flow calls, the algorithm runs in deterministic time $O(m \log n)$.*

Compared to the direct computation strategy described above, the lemma reduces the number of required maximum-flow computations from $k$ to $\lceil log_2\, k \rceil + 1$. As maximum-flow computations dominate the runtime in cut-based algorithms, the aforementioned reduction serves as key contributor as to why isolating-cut-based methods can be made scalable. The proof of this lemma relies on the submodularity of the cut function, as well as a decomposition of the terminal set into bipartitions.

## 4.5 Proof of the Isolating Cut Lemma

The following proof sketch closely follows Li and Panigrahi [21], while focusing only on the parts of the argument which are relevant for later algorithmic construction. The argument can be separated into two different phases, first a logarithmic number of group cuts is used to localise each terminal to a smaller region of the graph. Second, the recovery phase where the minimum isolating cuts are computed inside the reduced regions. To construct the group cuts, we fix an arbitrary ordering of the terminals and assign to each terminal its binary index of length $\ell = \lceil \log_2 |R| \rceil$. The purpose of these binary labels is to ensure that every pair of distinct terminals is separated by at least one of the group cuts. Two distinct binary strings differ in at least one bit position, which is why every pair of terminals is assigned to opposite sides of the bipartition at least once. Upon these binary labels we define two disjoint subsets of the terminal set for each bit position $i \in \{1, ..., \ell\}$.

$$R_i^0 := \{v \in R \mid i\text{-th bit of } v \text{ is } 0\} \quad R_i^1 := \{v \in R \mid i\text{-th bit of } v \text{ is } 1\}$$

For each bit position $i \in \{1, \ldots, \ell\}$ we compute one minimum cut separating the two defined terminal groups $R_i^0$ and $R_i^1$. We denote the resulting cut set by $C_i$, with each group cut being obtained by one maximum-flow computation.

**Definition 3 (Group Sets $U_v$)**
*For each terminal $v \in R$, we define $U_v$ as the connected component containing $v$, after removing all edges from the group cuts $C_1, \ldots, C_{\lceil \log_2 k \rceil}$.*

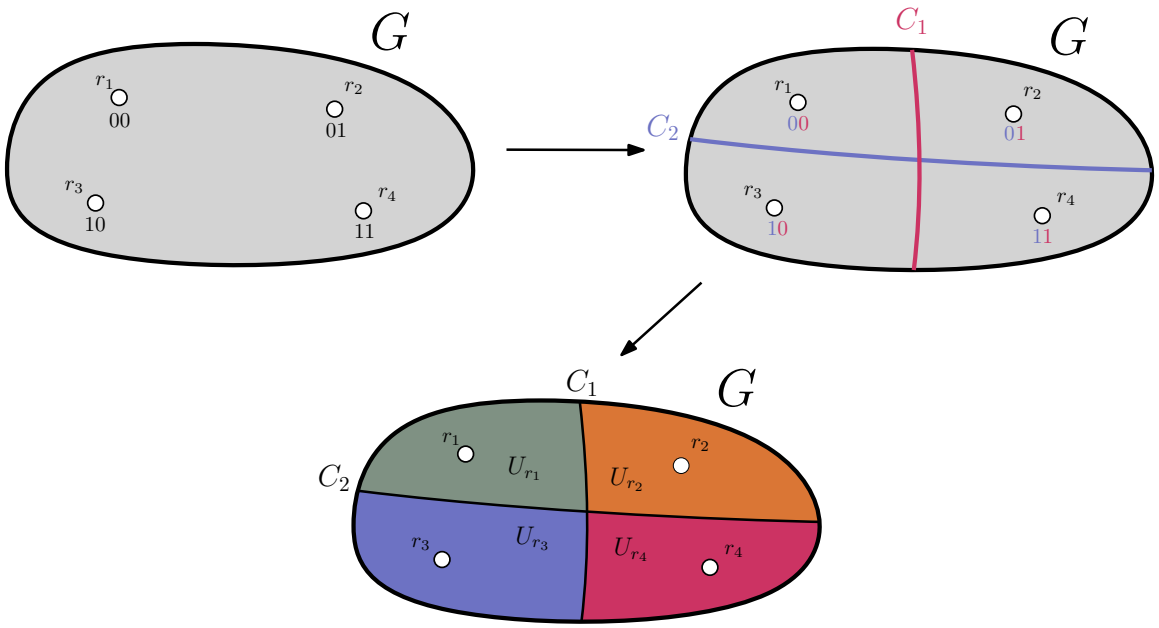


**Figure 4.1:** Illustration of the localisation idea behind the Isolating Cut Lemma. The left panel shows the original graph $G$ with terminal set $R$ of size 4, where each terminal is assigned a binary label. The right panel shows for each bit position, one group cut separating the corresponding terminal bipartition. The lower panel shows that after removing all group-cut edges, each terminal lies in a connected component $U_v$ containing no other terminal. The relevant minimum isolating cut is therefore localised inside this reduced region.

Once the group cuts have been removed from the graph it into connected components. For each terminal $v \in R$, the component $U_v$ serves as the local region in which the minimum isolating cut of $v$ can be recovered. We now establish two key properties of these components.

**Claim 1**
*For every terminal $v \in R$, the connected component $U_v$ contains no terminal other than $v$. In other words,*

$$U_v \cap R = \{v\}.$$

*Proof.* By definition, $v \in U_v \cap R$. We assume, for contradiction, that $U_v$ contains another terminal $u \neq v$. As $u$ and $v$ are distinct vertices in $R$, they are assigned different binary labels. Following this argument, we know that for some $j \in |R|$ it holds that $u$ and $v$ are divided by some bipartition $j$ into $R_j^0$ and $R_j^1$. Per definition, the group cut $C_j$ separates $u$ and $v$, hence, after removing the edges of $C_j$, the two vertices cannot be in the same connected component. This directly contradict previous assumption that there exists another terminal $u \in U_v$, proving $U_v \cap R = \{v\}$ for all terminals $v \in R$.

□

Continuing with the proof we also propose a second property. For each terminal $v \in R$, recall that $\lambda_v$ denotes the value of a minimum isolating cut of $v$. We define $S_v^*$ as the inclusion-wise minimal set satisfying

$$S_v^* \cap R = \{v\} \text{ and } \lambda_v = \omega(\delta(S_v^*))$$

meaning that there is no other smaller set $S_v' \subseteq S_v^*$, that is still a feasible minimum isolating cut of $v$. We propose that the minimum isolating cut for a terminal $v$ is contained in the respective group set $U_v$ for all terminals.

**Claim 2**
*For every terminal $v \in R$, the inclusion-wise minimal set $S_v^*$ defining a minimum isolating cut of $v$ is completely contained in the component $U_v$. That is,*

$$S_v^* \subseteq U_v.$$

*Proof.* We fix a terminal $v \in R$ and a bit position $i \in \{1, ..., l\}$. Let $T_v^i$ denote the side of the group cut $C_i$ that contains $v$. We claim that following property holds

$$S_v^* \subseteq T_v^i.$$

We assume for contradiction that $S_v^* \setminus T_v^i \neq \emptyset$, meaning that some part of the set $S_v^*$ lies outside $T_v^i$. Then the intersection $S_v^* \cap T_v^i$ is strictly smaller than only $S_v^*$, but still contains the terminal $v$ and no other terminal, making it a feasible isolating cut for $v$. As we chose the set $S_v^*$ as an inclusion-wise minimal set, the intersection of the two sets cannot have the same cut value. Therefore,

$$\omega(\delta(S_v^* \cap T_v^i)) > \omega(\delta(S_v^*)) = \lambda_v. \tag{4.1}$$

By applying submodularity to the two sets $S_v^*$ and $T_v^i$, we construct following inequality

$$\omega(\delta(S_v^*)) + \omega(\delta(T_v^i)) \geq \omega(\delta(S_v^* \cup T_v^i)) + \omega(\delta(S_v^* \cap T_v^i)). \tag{4.2}$$

Applying (4.1) to (4.2), it follows that

$$\omega(\delta(T_v^i)) > \omega(\delta(S_v^* \cup T_v^i)). \tag{4.3}$$

However, we chose $T_v^i$ to be the side of the minimum cut containing v while separating $R_i^0$ and $R_i^1$. Hence, $S_v^*$ must lie completely on the same side of every group cut alongside the terminal $v$. As $U_v$ is defined as the connected component that remains after removing all group cuts $C_i$, we conclude with

$$S_v^* \subseteq U_v.$$

□

Combining above claims, it shows that the group set $U_v$ of a terminal $v$ contains exclusively the terminal $v$ and the cut set $S_v^*$ of the inclusion-wise minimal minimum isolating cut. Inferring from that, it is sufficient to compute the minimum isolating cuts for each terminal in its respective component $U_v$, as the relevant cut structure is localised inside $U_v$.

**Analysis.** The proof yields the complexity improvement stated by the Isolating Cut Lemma. The first stage requires exactly $\lceil \log_2 k \rceil$ group cuts, and each of these is obtained by one maximum-flow computation. After identifying all group sets, the recovery stage suffices to obtain the family of minimum isolating cuts. Outside of these flow computations, the remaining work merely consists of constructing the terminal bipartitions, removing the group cuts and computing the connected components, which can be carried out in deterministic time $O(m \log n)$.

**Algorithmic Consequence.** These structural certainties allow for an immediate algorithmic translation. Once the group sets have been identified, the minimum isolating cuts can be recovered on the reduced instances, thus leading to a proposed two-phase strategy. First we localise the terminals with the help of group cuts, and then we compute the family of isolating cuts on the smaller components. This completes the proof sketch of the Isolating Cut Lemma and forms the theoretical basis for the algorithmic variants implemented in the next chapter.

CHAPTER 5

# Engineering Isolating Cut Algorithms

In the previous chapter, we established the Isolating Cut Lemma and derived the two-phase structure it induces: the group phase that localises the solution for each terminal and the recovery phase which computes the final isolating cuts inside the respective group sets. In this chapter, we describe how these methods are turned into concrete algorithms and implementations in the KAHIP [17] framework.

## 5.1 Overview

We consider three sequential approaches for computing the family of minimum isolating cuts. The direct baseline computes one isolating cut for each terminal on the full graph. The second approach uses the Isolating Cut Lemma to first localise each terminal in a logarithmic number of group cuts to then recover the isolating cuts on reduced instances. The third approach uses the same localisation phase, but combines the recovery step into one batched maximum-flow computation on a disjoint-union instance. For all three approaches, we additionally implement shared-memory parallel version. Table 5.1 summarises the three sequential methods alongside their main structural differences. Since the mathematical foundations were already introduced in previous chapters, the primary focus lies in the algorithms.

**Table 5.1:** Overview of the implemented algorithmic variants.

| Variant | Localisation | Isolating Cut Computation |
|---|---|---|
| TERMCUT | no | $k$ cuts on $G$ |
| ISOCUT | yes | $k$ cuts on $U_v$-contracted graphs |
| BATCHISOCUT | yes | one flow on a disjoint-union instance |

**Algorithm 1** TERMCUT

**Input :** Weighted graph $G = (V, E, \omega)$, terminal set $R \subseteq V$
**Output:** Isolating cuts $(C_r)_{r \in R}$
$C_\infty \leftarrow 1 + \sum_{e \in E} \omega(e)$
**foreach** $r \in R$ **do**
  $F_r \leftarrow$ BUILDTERMINALFLOW$(G, R, r, C_\infty)$
  $S_r \leftarrow$ SOURCESIDEOFMINCUT$(F_r, r, t)$
  $C_r \leftarrow \delta_G(S_r)$
**return** $(C_r)_{r \in R}$

The remainder of this chapter proceeds from the simplest to the most sophisticated variant. We start out with the direct baseline, then describe how the localisation phase refines the problem with the help of the Isolating Cut Lemma and conclude by showing how the recovery phase can be batched. After discussing the sequential versions, we focus on their parallelisation.

## 5.2 Terminal-wise Baseline

First we consider the direct baseline algorithm. The idea here is to iteratively and independently compute a minimum isolating cut for each terminal. This follows the most immediate reduction of the problem to compute the family of minimum isolating cuts and serves as a natural reference point for later variants. It does not take advantage of any additional structural properties, but starts from a clean slate for every terminal in $R$.

Algorithm 1 summarises the workflow of the method. For each terminal $v \in R$, all remaining terminals are connected to one artificial super sink node with edges of sufficiently large capacity to imitate an infinite value $C_\infty$. The minimum $v-$sink cut on this constructed instance then yields the minimum isolating cut of $v$. Repeating this step for every terminal in $R$ yields the full family of isolating cuts.

In the implementation, each terminal-specific instance is obtained from the shared base flow graph by adding one artificial super-sink and $|R| - 1$ terminal-to-sink edges. The minimum $v$-sink cut is then computed using the shared maximum-flow backend provided in KAHIP [17]. The returned source side is translated back to the original graph, and all crossing edges are marked as part of the isolating cut of the current terminal.

This builds the conceptually simplest variant computing the family of minimum isolating cuts. Every minimum isolating cut is computed independently from all others, thus keeping the method easy to understand and introducing little additional overhead beyond the repeated flow computations themselves. Every terminal induces its own cut problem on the full graph. The main drawback is that neither the number nor the size of the terminal-wise cut instances is reduced, therefore, performing $k$ minimum-cut computations on the full graph.

**Algorithm 2** IsoCut (Overview)

**Input :** Weighted graph $G = (V, E, \omega)$, ordered terminal set $R$
**Output:** Isolating cuts $(C_r)_{r \in R}$
$E_{\text{bit}} \leftarrow$ BitwiseSeparation$(G, R)$
compute the connected components of $G - E_{\text{bit}}$
**foreach** $r \in R$ **do**
  $U_r \leftarrow$ component of $G - E_{\text{bit}}$ containing $r$
  $C_r \leftarrow$ RecoverCut$(G, U_r, r)$
**return** $(C_r)_{r \in R}$

The main advantage remains in the simplicity involved in its theoretical background. As all terminal computations remain independent, the method exposes a large amount of task parallelism, which will be exploited later when discussing the parallel variants. Similarly, the independence also is the main limitation of the approach, as it does not exploit any additional structural information shared between different cuts.

## 5.3 Isolating Cut

Compared with the direct terminal-wise computation the second approach exploits additional structural information provided by the Isolating Cut Lemma. Instead of computing all isolating cuts directly on the full original graph, the method first each terminal to a smaller region by means of a logarithmic number of group cuts. The final isolating cuts are then recovered in these reduced instances by solving terminal-wise cut problems.

At a high level, this method extends the direct baseline by inserting a localisation step before the terminal-wise isolating-cut computations. As in the direct baseline approach, maximum-flow computations are still used to obtain cuts via auxiliary source and sink constructions. However, instead of immediately solving one isolating-cut problem for each terminal on the full graph, the algorithm first computes a logarithmic number of group cuts that separate the chosen terminal subsets from one another. The union of these group cuts partitions the graph into smaller connected components, called group sets $U_v$, each of which contains exactly one terminal $v$. In the second phase, the isolating cut for $v$ is then recovered only inside the corresponding region $U_v$. Thus, the method retains the same basic flow-based cut logic as the direct baseline, but uses an additional preprocessing phase to reduce the size of the instances on which the terminal-wise recovery problems are solved.

Algorithm 3 summarises the two-phase structure of this method. We start out by computing a logarithmic number of group cuts and use the resulting union of these cut edges to derive the terminal-specific group components $U_v$. The following phase then solves one reduced cut problem per terminal on the corresponding contracted graph.

**Algorithm 3** IsoCut

**Input :** Weighted graph $G = (V, E, \omega)$, terminal set $R = (t_0, \ldots, t_{k-1})$
**Output:** Set of cut edges $E_{\text{cut}}$
**Function** `IsoCut(`$G, R$`)`**:**

```
(F, C_∞) ← PREPROCESS(G, two reserved nodes)
E_group ← ∅
E_cut ← ∅
b ← ⌈log_2 |R|⌉
for i ← 0 to b − 1 do
    R_i^0 ← {t_j ∈ R | i-th bit of j is 0}
    R_i^1 ← {t_j ∈ R | i-th bit of j is 1}
    F_i ← copy of F
    connect the reserved source of F_i to all terminals in R_i^0 with capacity C_∞
    connect all terminals in R_i^1 to the reserved sink of F_i with capacity C_∞
    S_i ← source side of MAXFLOWMINCUT(F_i, source, sink)
    mark all edges in δ_G(S_i) in E_group
compute the connected components of G − E_group
let U_v be the component containing v for each v ∈ R
foreach v ∈ R do
    build a new flow graph F_v on U_v and one super-sink
    redirect every edge leaving U_v to the super-sink
    S_v ← source side of MAXFLOWMINCUT(F_v, v, sink_v)
    map S_v back to the original graph and mark all edges in δ_G(S_v) in E_cut
return E_cut
```

**Group Phase.** The first phase follows the construction introduced in the previous chapter. The terminal set $R$ is assigned an arbitrary but fixed order, and then identified via the binary representation of its corresponding index. For each position $i \in \{0, ..., \lceil \log_2 k \rceil - 1\}$, we induce a bipartition of the terminal set into two groups $R_i^0$ and $R_i^1$, depending on the binary index of the $i$-th bit currently considered. Based on these bipartitions, we compute a minimum-cut separating the two constructed terminal groups.

Algorithmically, this is realized by the same underlying flow representation as the direct baseline, but with two reserved auxiliary vertices. One of these serves as a super-source vertex while the other serves as a super-sink. The terminals in $R_i^0$ are connected to the induced source, while the terminals in $R_i^1$ are respectively connected to the sink with edge capacity of $C_\infty$. The minimum-cut computation is then run on the pair of super-source and super-sink, to separate the two terminal groups from one another. As before, the returned source set and its accompanying cut edges are then translated to the original graph and marked.

After all groups cuts have been computed, the union of these marked edges is removed implicitly from the graph. We then compute the connected components with the help of

breadth-first search on the remaining graph. For every terminal $v \in R$, the connected component defines the corresponding group set $U_v$ needed in the second phase. This concludes the localisation step of the algorithm and bounds the region where the isolating cut of $v$ may lie, as previously discussed in Chapter 4.

**Recovery of the Isolating Cut Family.** Once the group sets $U_v$ have been computed, the second phase of the algorithm intends to recover the actual isolating cuts of the terminals. For each such terminal $v \in R$, we construct a reduced flow instance that contains the vertices of $U_v$ explicitly, while representing all other vertices via an artificial super-sink. A local mapping assigns each vertex of $U_v$ to its corresponding vertex in the reduced instance. After this, all outgoing edges of vertices in $U_v$ are scanned and depending on whether both endpoints lie in $U_v$, we add the edge explicitly between the local vertices, or we redirect it to the super-sink, otherwise. Computing the minimum-cut separating $v$ and the artificial super-sink then yields the corresponding minimum isolating cut of $v$ on the reduced instance. Finally, the source side is translated back to the original graph and the corresponding cut edges are extracted.

Thus, the method still performs one recovery computation per terminal, however, these are carried out on components which are reduced in size rather than the full graph. Compared with the direct baseline, this introduces an additional logarithmically sized localisation phase, but substantially reduces the size of the final terminal-wise recovery instances.

Even though this approach performs an additional $\lceil \log_2 k \rceil$ group-cut computations, these are the only maximum-flow computations carried out on the full graph. The subsequent recovery phase still solves one cut problem per terminal, but each of these computations is performed on a reduced instance derived from the corresponding group set $U_v$. Specifically each vertex in the input graph appears in at most one group set $U_v$ and each reduced instance only introduces one additional super-sink vertex. Following this logic, each original edge can contribute to at most two reduced instances in the second phase, as an edge can run at most in-between two of the group sets. Hence, the implementation realizes the recovery phase as terminal-wise computations, however the total input size of the problems matches the amortized bound provided by the Isolating Cut Lemma.

This observation motivates the third variant, as Li and Panigrahi [21] also noted in their work. The recovery phase can be combined explicitly by forming a disjoint-union instance of the reduced instances and running one batched maximum-flow computation, which will be explored in the next section.

## 5.4 Batched Isolating Cut

The third variant keeps the same localisation phase as the previous two-phase approach, but changes the way the family of isolating cuts is recovered. Instead of solving one reduced minimum-cut instance for each terminal separately, it combines all recovery instances into a single batched flow graph obtained as the disjoint union of the terminal-specific reduced

**Algorithm 4** BATCHISOCUT

**Input :** Weighted graph $G = (V, E, \omega)$, ordered terminal set $R$
**Output:** Isolating cuts $(C_r)_{r \in R}$
$E_{\text{bit}} \leftarrow$ BITWISESEPARATION$(G, R)$
compute the connected components of $G - E_{\text{bit}}$
**foreach** $r \in R$ **do**
  $U_r \leftarrow$ component of $G - E_{\text{bit}}$ containing $r$
$(C_r)_{r \in R} \leftarrow$ BATCHEDRECOVERCUTS$(G, (U_r)_{r \in R}, R)$
**return** $(C_r)_{r \in R}$

**Algorithm 5** BATCHEDRECOVERCUTS

**Input :** Weighted graph $G = (V, E, \omega)$, terminal components $(U_r)_{r \in R}$, terminal set $R$
**Output:** Isolating cuts $(C_r)_{r \in R}$
construct a disjoint union of the recovery networks for all terminals
introduce a global source $s^*$ and a global sink $t^*$
**foreach** $r \in R$ **do**
  let $F_r$ be the recovery network induced by $U_r$ with local sink $t_r$
  connect $s^*$ to the copy of $r$ in $F_r$
  connect $t_r$ to $t^*$
$S^* \leftarrow$ SOURCESIDEOFMINCUT$(F, s^*, t^*)$
**foreach** $r \in R$ **do**
  $S_r \leftarrow$ vertices of $U_r$ whose copies lie in $S^*$
  $C_r \leftarrow \delta_G(S_r)$
**return** $(C_r)_{r \in R}$

instances. Here, we define disjoint union to be a graph where each reduced instance is copied with its own private set of vertices and edges, so that different instances do not share vertices with one another, while being connected only through one additional global source and global sink. The second phase is then solved by a single maximum-flow computation on this combined graph. Thus, this method follows more closely the final maximum-flow structure suggested by the Isolating Cut Lemma of Li and Panigrahi in [21].

Algorithm 4 summarises the method. The first phase remains unchanged with a logarithmic number of group cuts being computed and used to derive the terminal-specific group sets $U_v$. The only difference lies in the second phase, where the terminal-wise recovery computations are replaced by a singular batched computation on the disjoint-union graph.

**Batched Recovery on a disjoint-union instance.** After the group sets $U_v$ have been computed, we no longer solve one reduced minimum-cut problem per terminal. Instead, for every terminal $v \in R$, we copy the vertices of $U_v$ into one larger batched flow graph, alongside an additional local sink vertex representing the remainder of the graph. Thus, each terminal contributes one reduced recovery instance of size $|U_v|+1$ to the batched

graph.

The edges of the batched graph are constructed analogously to the reduced recovery instances of the previous method. If a copied edge has both endpoints in $U_v$, we explicitly add it in the batched graph. If the target of the edge leaves the terminal group set $U_v$, it is redirected to the respective local-sink of the group set. Additionally, the copied terminal vertex is connected to a global source and each local sink is connected to a global sink with capacity $C_\infty$. After processing all terminals in $R$, this yields a disjoint union of local recovery instances that are connected only through the global source and sink vertices.

One singular maximum-flow computation on this batched graph then yields the complete recovery phase in one computation, instead of having to compute terminal-wise minimum isolating cuts.

To translate the result back to the original graph, each copied vertex in the batched construction stores two pieces of information. For one it retains the original vertex of $G$ from which it was copied and it also stores the terminal $v \in R$ whose recovery instance it belongs to. After the minimum cut has been computed, we consider the source side of the cut in the batched graph. As the batched instance is a disjoint union of terminal-specific recovery graphs, the source side naturally decomposes into one part for each terminal $v$. For every terminal, we collect these copied vertices of the $U_v$-instance that remain reachable from the global source. Mapping these copied vertices back through the stored correspondence yields a vertex set $S_v \subseteq V$ in the original graph. The isolating cut of $v$ is obtained as the set of original edges with one endpoint in $S_v$ and the other outside $S_v$, that is, the cut $\delta(S_v)$.

Compared to the previous structured variant, this method differs only in the recovery phase. It replaces $k$ terminal-wise recovery computations with the one global flow computation on the batched graph as described in [21]. While reducing the number of flow operations, we increase the additional overhead in the second-phase by a more complicated construction of the flow graph and an additional mapping step needed to separate the global source set into its respective pieces. Thus, the method reduces the number of flow computations in the second phase, while increasing the complexity of the surrounding graph construction and extraction steps.

## 5.5 Parallelisation Strategies

All three algorithmic variants expose independent tasks that can be taken advantage of in a shared-memory parallel setting. Across all parallelisation strategies mentioned, we reuse the same principles. In particular, each thread maintains its own temporary data structures whenever possible. These include a private residual or working flow graph for the current maximum-flow computation, a local marker array indicating which vertices belong to the source side of the computed cut and a local array for the cut edges extracted from that result. Instead of updating shared global data structures during the thread-wise computations, each thread writes its intermediate results only to its own local memory. The thread-local results are combined only after the parallel tasks of the current phase have

**Table 5.2:** Overview of the parallel variants.

| Approach | Parallel Work | Sequential Bottleneck |
|---|---|---|
| Direct terminal-wise | terminal cuts | final merge |
| Localized recovery | group cuts, recovery cuts | components |
| Batched recovery | group cuts, extraction | components, batched flow |

finished. This design keeps the synchronisation overhead low. Specifically, threads do not compete while running the maximum-flow computations or while marking source-side vertices and cut edges. Synchronisation is therefore only required in the final merge step of a phase, when the thread-local results are combined into a single global representation. Thus avoiding race conditions and reducing contention on shared memory.

The difference between the three variants lies inherently in the amount and location of available parallel work. The direct terminal-wise baseline exposes one independent task per terminal. The localised approach exposes parallel work both in the group-cut phase and in the terminal-wise recovery phase, but still contains sequential bottlenecks between the two stages. The batched variant shares the same parallel group phase, but its second phase contains only one global maximum-flow computation and therefore offers less coarse-grained task parallelism. To address load imbalance, dynamic scheduling is used wherever task sizes are predicted to vary substantially, while resorting to static scheduling for the more balanced bitwise group-cut phases.

Table 5.2 shows the parallelisation strategies for all methods. In each case, independent cut computations are assigned to different threads. Each thread stores its intermediate source-set information and extracted cut edges locally and these local results are merged only after the parallel phase has completed. This setup avoids fine-grained contention on shared edge data while keeping the expensive flow computations thread-local.

**Parallel Direct Terminal-Wise Computation.** The parallelisation of the direct baseline is immediate, as the terminal-wise cut computations are completely independent from one another. Therefore, the loop over all terminals can be parallelized as is. Each thread starts from its own copy of the flow graph, inserting the required terminal-to-sink edges and solving the corresponding cut problem independently on the thread. The resulting source side and cut edges are written into thread-local data structures and are merged only after all terminal tasks have finished. We parallelise the work dynamically here one by one, as the runtime of the individual terminal cuts may vary with the structure of the graph and the position of each terminal. Synchronisation is therefore limited to the final merge of the thread-local cut-edge arrays as well as the accumulation of the cut values. Thus preserving the simplicity of the sequential baseline while exploiting the large amount of task parallelism exposed by the terminal-wise decomposition.

**Parallel Localised Recovery.** The localised variant exposes parallel work in both of its phases, but also contains sequential dependencies between them. In the first phase, the bitwise group cuts are completely independent of one another and can therefore be processed in parallel. However, this phase consists only of $\lceil \log_2 k \rceil$ many group-cut tasks, because exactly one minimum-cut computation is performed for each bit position in the binary encoding of the terminals. Hence, the amount of parallel work available in this phase is limited compared to the terminal-wise phases, especially when the number of threads is large. Each thread constructs the corresponding terminal bipartition, computes the group cut and stores the marked cut edges in a thread-local flag array.

After all group cuts have been computed, the thread-local arrays must first be merged to obtain the complete union of group-cut edges. The connected components defining the group sets $U_v$ cannot be computed earlier, because the component structure depends on the full set of edges removed by all group cuts together. If one group cut has not been computed yet, two vertices may still appear connected even though they are separated in the final reduced graph. For this reason, the connected components can only computed after all thread-local group-cuts have been merged into one global edge-removal structure.

The second phase also exposes task parallelism. Once all the group sets $U_v$ have been computed, the reduced recovery instances for all terminals are again independent from one another and can therefore be distributed across the threads, similar to the direct baseline. Therefore, for each terminal the thread constructs the corresponding $U_v$-contracted graph, computes the necessary minimum-cut, translates the resulting source set back to the original graph and marks the cut edges in a thread-local array. The work here is distributed dynamically, as the sizes of the sets $U_v$ may vary substantially and therefore leads to recovery tasks of different cost.

**Parallel Batched Recovery.** The batched variant shares the same localisation phase as the previous two-phase localisation-based method. Accordingly the bitwise group cuts are parallelised following the same logic. The group-cut tasks are distributed across different threads, their cut-edges are stored locally and then the results are merged before the connected components are computed sequentially.

The main difference lies in the recovery phase. As the batched recovery graph is constructed only once and solved by a singular global maximum-flow computation, it does not expose terminal-wise task parallelism. The available parallel work is shifted to the post-processing step only. After computation of the global source side, it is partitioned into terminal-specific source sets and thus parallelised over the final edge extraction. In this way, the batched variant combines this second-stage global maximum-flow computation with parallel post-processing of the extracted terminal-wise cuts.

CHAPTER 6

# Experimental Evaluation

In this chapter, we evaluate the practical performance of the implemented isolating-cut variants. We first describe the experimental setup, including the hardware environment, benchmark instances, and recorded measurements. Afterwards, we analyse the sequential variants with respect to runtime scaling, phase-wise behaviour, and memory consumption. We then study the parallel implementations and evaluate their scalability, thread imbalance, and the limits imposed by the available task parallelism.

## 6.1 Experimental Setup

**Hardware Environment.** Most benchmark instances were evaluated on Intel-based compute nodes equipped with an INTEL XEON SILVER 4216 processor at 2.10 GHz, providing 16 physical cores, 32 hardware threads, and 92 GiB of main memory. For the largest benchmark instances used, namely `youtube`, `eur`, `road_usa`, and `skkernel`, we used an AMD-based compute node equipped with an AMD EPYC 7702P processor at approximately 2.0 GHz, providing 64 physical cores, 128 hardware threads, and 995 GiB of main memory. In both cases, the experiments were executed on a single socket and one NUMA domain. All comparisons for a fixed graph instance were performed on the same machine class. Hence, although two different hardware classes were used overall, we ensure that direct runtime comparisons for a given instance are not affected by switching between architectures.

**Software Environment.** The implementation is written in C++11 and integrated into the KAHIP framework. All binaries were built with `CMake` in RELEASE mode, using the release flags `-O3 -DNDEBUG`. In addition, the build configuration enabled the compiler flags `-march=native`, `-funroll-loops`, and `-fno-stack-limit`. Parallel executables were compiled and linked with OPENMP support. The Intel-based nodes use UBUNTU 24.04.3 LTS and `g++ 13.3.0`. The AMD-based node also runs UBUNTU

24.04.3 LTS, using `g++ 12.2.0`. Since the build configuration employs architecture-specific optimizations, binaries were compiled separately for the Intel- and AMD-based machine classes. The source code used for the experiments in this thesis, together with the scripts for running the benchmarks and the raw result files used for plotting and aggregation, is available at `https://github.com/calvalre/k_initial_cuts`.

**Methodology.** We evaluate all six implemented variants. For every graph instance, experiments were performed for terminal counts of $2, 4, 8, 16, 32, 64, 128$.

In particular, this was chosen as the logarithmic scaling of the terminal count imitates the first phase of the structured variants which depends logarithmically on the number of terminals. To reduce the influence of how terminals are chosen in the vertex set of a graph, every experimental setup of GRAPH, TERMINAL AMOUNT, ALGORITHM, (THREAD COUNT) was repeated on three different random seeds. The terminal set for a run is chosen uniformly at random from the set of vertices of a graph using the corresponding seed. Unless stated otherwise the reported and discussed runtimes are aggregated over the three seeds by taking the median value.

All sequential variants were executed with one thread. The parallel variants were evaluated for thread counts of $1, 2, 4$ and $8$. Larger thread counts were not considered, as the structured variants expose only $\lceil \log_2 k \rceil$ many independent group-cut tasks in the first phase. Specifically, for the largest tested terminal count of $k = 128$, this limits the useful parallelism in the first phase to at most 7 independent tasks, with the parallelisation strategy chosen.

For every run the overall runtime was recorded alongside several phase-wise timings that depend on the algorithmic variant analysed. Among others these include but are not limited to the graph input, terminal selection, flow-graph preprocessing, bitwise group phase, component construction, recovery phase and flow computations. Additionally the resulting cut value, number of affected cut edges, peak memory consumption and, for the parallel runs, a thread-imbalance measure is recorded.

**Benchmark Instances.** To evaluate the implemented methods on a broad range of graph structures, we use benchmark instances from several established public graph collections. The benchmark set includes graph-partitioning and road-network instances from the 10th DIMACS Implementation Challenge [3], structural benchmark graphs from the Walshaw Graph Partitioning Archive [28], social, citation, collaboration, communication, and road-network graphs from SNAP [19], web graphs from the LAW/WebGraph dataset collection [18, 4], sparse matrix and circuit-derived graphs from the SuiteSparse Matrix Collection [9], university social-network graphs from the Facebook100 data set [25], and the `soc-lastfm` instance from the Network Repository [23]. In addition, the road instance `eur` is taken from the Western Europe road-network benchmark commonly used in route-planning research [24]. The resulting benchmark set covers mesh, geometric, structural, circuit, citation, web, road, and social graphs. Table 6.1 records the benchmark

instances used in our experiments. Overall, the instances range from small graphs with only a few thousand vertices to very large graphs with tens of millions of edges, allowing us to study both behaviour on smaller inputs and practical scalability on large real-world instances.

**Recorded Measurements.** As different algorithmic methods expose different phases, not all timings are shared across all implementations. Accordingly, in the evaluation following we only compare phase-wise measurements where the corresponding phases are present. The recorded data is used to analyse overall runtime behaviour, as well as how different phases distribute the work and runtime load, the effect of localisation and batching, memory consumption and the scalability of the parallel implementations.

## 6.2 Research Questions

The goal of this chapter is to evaluate the practical behaviour of the proposed isolating-cut algorithms and to understand in which situations the different design choices are most beneficial. In particular, we compare three variants introduced in the previous chapter: the direct terminal-wise approach, namely TERMCUT, the *Localised Recovery* approach, called ISOCUT, and the *Batched Recovery* approach, referred to as BATCHISOCUT. Here, TERMCUT serves as the internal baseline, as it applies the isolating-cut computation directly without exploiting the structural refinement of the Isolating Cut Lemma. In addition, we compare against an external exact solver, denoted by *KTerminalCut* provided by Velednitsky [26], which serves as an external baseline for runtime and solution quality. To structure the experimental evaluation, we formulate the following research questions.

- **RQ1.** How do TERMCUT, ISOCUT and BATCHISOCUT scale with the number of terminals?
- **RQ2.** How much does the refinement based on the Isolating Cut Lemma improve practical performance over the direct terminal-wise baseline?
- **RQ3.** Does batching the recovery phase provide an additional practical benefit over terminal-wise recovery on reduced instances?
- **RQ4.** How well do the approaches parallelise and what algorithmic bottlenecks limit their scalability?
- **RQ5.** What memory overhead do localisation, batching and parallelisation introduce in practice?
- **RQ6.** How competitive are the proposed methods compared with an external exact solver in terms of runtime and solution quality?

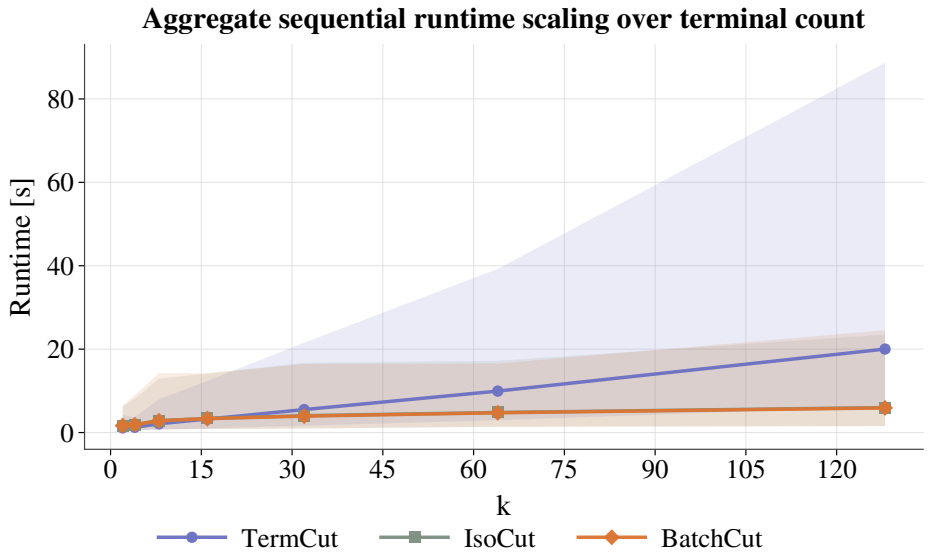


**Figure 6.1:** Aggregate sequential runtime scaling over terminal count $k$. The solid lines show the geometric mean runtime over all complete benchmark cases $(\texttt{graph}, k)$; the shaded areas indicate the interquartile range across cases.

These questions are answered in the remainder of the chapter. First we describe the experimental setup. Afterwards, we study the sequential behaviour of the approaches with respect to runtime scaling and phase-wise cost distribution, then analyse their shared-memory parallelisation, followed by memory consumption and a comparison with an external exact solver.

## 6.3 Sequential Behaviour

This section addresses **RQ1–RQ3**. We first compare how the three sequential approaches scale with increasing terminal count. We then analyse which algorithmic phases dominate the runtime and use this decomposition to assess the practical benefit of localisation and batching. Across every algorithmic setup previously defined, the recorded solution qualities of the three sequential variants were identical. Therefore, we do not discuss solution quality any further and instead focus the experimental evaluation on runtime, phase-wise behaviour, and memory consumption. To complement the instance-level plots shown later, we first report aggregate statistics over the full benchmark set. Unless stated otherwise, these aggregate results are computed over all complete benchmark cases $(\texttt{graph}, k)$, where runtime is defined as the median over the three seeds. For aggregate runtime comparisons across heterogeneous graph classes, we use geometric means, since absolute runtimes vary by several orders of magnitude.

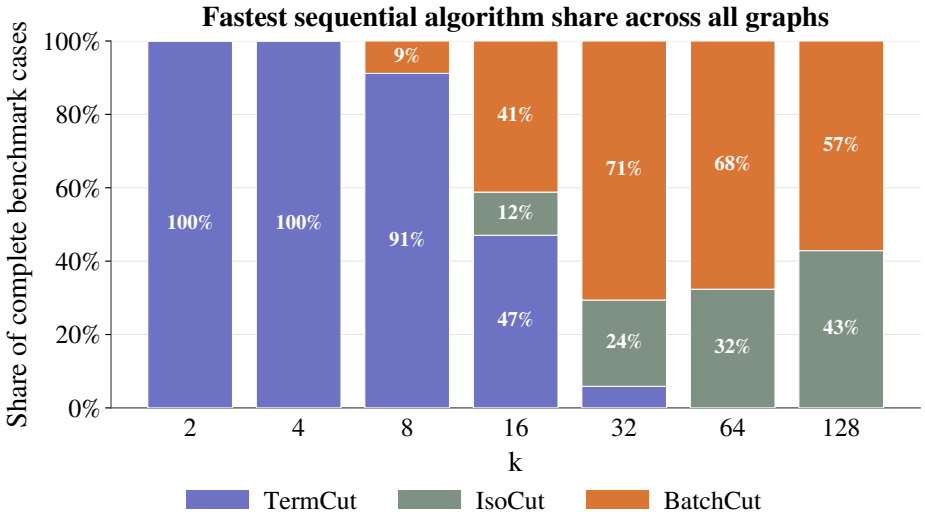


**Figure 6.2:** Share of complete benchmark cases on which each sequential method is the fastest, grouped by terminal count $k$.

**Runtime Scaling with the Number of Terminals.** We start out by comparing the overall runtime trends recorded. Figure 6.3 reports the most representative scaling trends on four chosen benchmark graphs. In particular, the selected instances cover a large geometric graph, a large social network, a large road network and a large citation graph. Thus, they capture the main runtime patterns that also appear across the broader benchmark set.

The aggregate results confirm a clear crossover in behaviour over the full benchmark set. For small terminal counts, TERMCUT is most often the fastest method, whereas for larger terminal counts the localisation-based variants dominate a growing fraction of the benchmark cases. In particular, across the low-terminal cases $k \in \{2, 4, 8\}$, TERMCUT is faster than the better structured variant on $96.9\%$ of the complete cases. Across the high-terminal cases $k \in \{32, 64, 128\}$, however, the better of ISOCUT and BATCHCUT is faster than TERMCUT on $98.1\%$ of the complete cases, and TERMCUT is slower by a factor of $2.40$ on average. At $k = 128$, TERMCUT is slower by a factor of $3.53$ on average than the better of the two localisation-based variants.

For smaller terminal counts, TERMCUT is frequently observed to be the fastest of the three sequential methods. This is not surprising as the baseline avoids the computational overhead required by the other methods. TERMCUT directly computes the terminal-wise minimum isolating cuts on the original graph while avoiding the localisation phase. For smaller and medium-sized instances, the avoided structural overhead is clearly visible. To illustrate the same trend on concrete benchmark instances. This trend is also visible on the representative instances in Figure 6.3. On `delaunay_n19`, TERMCUT is initially competitive for small terminal counts, but then grows from roughly $1.5$ seconds at $k = 2$ to about $28$ seconds at $k = 128$, whereas ISOCUT and BATCHISOCUT only rise to about $9$ seconds. A similar crossover appears on `youtube`, where TERMCUT increases to

about $56$ seconds at $k = 128$, while the two localisation-based variants remain close to $16$ seconds.

However, with an increasing amount of terminals we see the trend of fastest algorithm change.TERMCUT's runtime grows quickly, the larger the terminal set appears to be, as the method repeats the full terminal-wise construction and minimum-cut computations for every terminal on the original graph. Alternatively, ISOCUT and BATCHISOCUT first localize their solution space to then recover the family of isolating cuts on reduced instances, which improves scaling for larger $k$. The same behaviour becomes even clearer on the larger representative instances. On `roadNet-TX`, TERMCUT grows to roughly $53$ seconds at $k = 128$, whereas ISOCUT and BATCHISOCUT stay near $17.5$ and $16.5$ seconds, respectively. On `cit-Patents`, the gap is even larger. TERMCUT reaches about $455$ seconds at $k = 128$, while ISOCUT and BATCHISOCUT remain close to $90$ seconds. Hence, the crossover from the direct baseline to the localisation-based variants is not limited to one graph class, but appears consistently across geometric, social, road, and citation instances.

While there is a large difference for the runtime values of TERMCUT, the localisation-based variants do not differ significantly from one another. Across many instances, both of them show nearly identical runtime curves, with the batched variant often being slightly faster than ISOCUT for a larger number of terminals. This aligns with the algorithmic structure, as both methods share the same localisation phase and differ only in the recovery phase. Consequently the batching of BATCHISOCUT can improve the runtime of this recovery phase, but the cost of the previous computation phase in the group phase cannot be eliminated with that.

Overall, the sequential results show a clear trade-off for the implemented methods. TERMCUT remains the simplest and most effective choice for small problem instances, where the nearly nonexistent overhead dominates in contrast to the other algorithms. However, for larger $k$, the localisation-based variants quickly become favourites to use, as they profit from the localisation phase that is guaranteed by the Isolating Cut Lemma and thus scale better. Therefore, the main practical advantage of ISOCUT and BATCHISOCUT lies in substantially reducing the runtime growth as the number of terminals increases.

**Answer to RQ1.** The sequential experiments confirm a clear crossover in behaviour over the full benchmark set. On small terminal counts $k \in \{2, 4, 8\}$, TERMCUT is faster than the better localisation-based variant on $96.9\%$ of the complete benchmark cases. For larger terminal counts $k \in \{32, 64, 128\}$, however, the better localisation-based variant is faster on $98.1\%$ of the complete cases, and TERMCUT is slower by a factor of $2.40$ on average. Thus, the direct baseline is preferable for small terminal counts, whereas the localisation-based variants are the more scalable choice once $k$ becomes large.

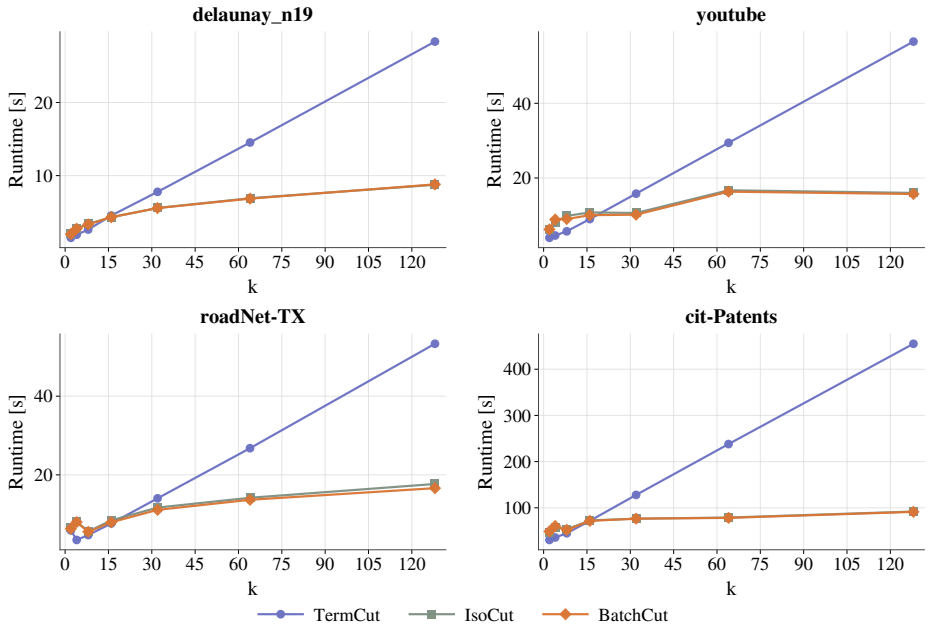


**Figure 6.3:** Median runtime of the sequential variants over increasing terminal counts on representative benchmark instances.

**Phase-wise Runtime Breakdown.** To explain the runtime trends in more detail, we decompose the measured runtimes into their main algorithmic phases. For TERMCUT we distinguish between the time spent in the repeated maximum-flow computations and the remaining terminal-wise overhead. For ISOCUT and BATCHISOCUT we distinguish between the localisation phase, the connected-component computation between the two phases and the recovery phase. For clariy, we refer to ISOCUT and BATCHISOCUT jointly as the localisation-based variants, since both methods share the same localisation phase and differ only in the structure of the recovery stage. Figure 6.4 shows the relative phase composition of TERMCUT. We distinguish between the measured maximum-flow time as well as the remaining terminal-wise overhead required, as we rebuild and process a full flow-instance for every terminal. The direct baseline scales poorly not only because it repeatedly solves maximum-flow instances, but also because the surrounding terminal-wise construction work in itself is substantial. Besides the measured flow time, each terminal requires copying the flow graph, inserting the artificial sink edges, marking the source set and translating the cut-edges back to the original graph. As all of these steps are repeated for every terminal in $R$, their contribution grows with the number of terminals and thus forms a significant part of the total runtime required. On the representative instances shown in Figure 6.4, terminal overhead accounts for about 71–74% of the runtime on `delaunay_n19`, around 67–74% on `youtube`, roughly 55–70% on `roadNet-TX`, and approximately 61–67% on `cit-Patents`. Thus, the poor scaling of TERMCUT is caused by both repeated flow computations and repeated terminal-wise graph construction.

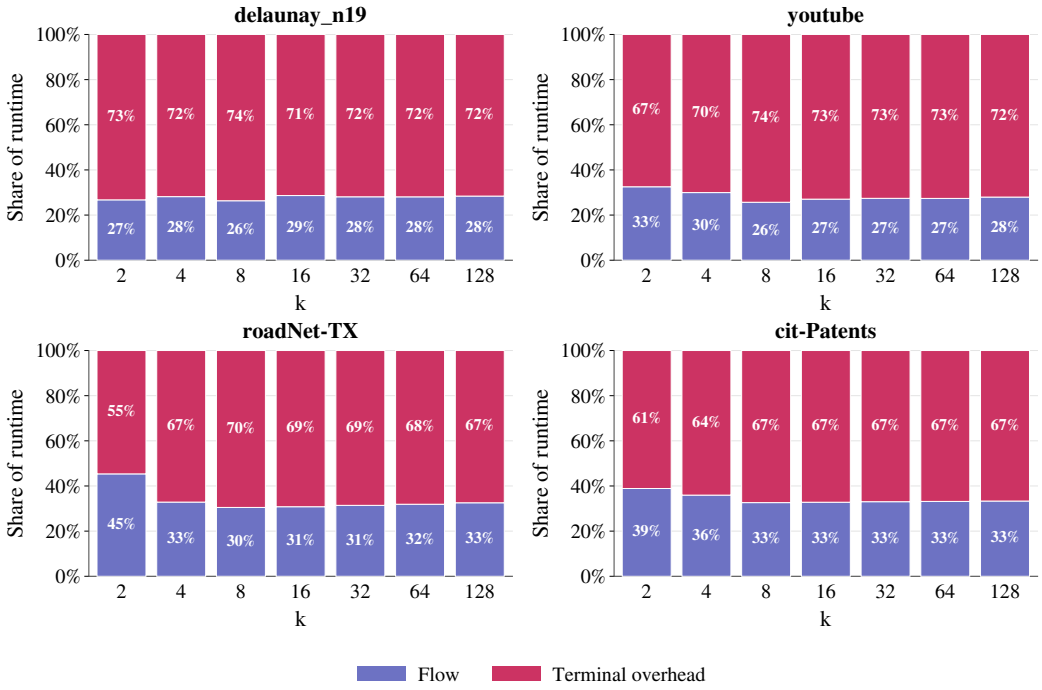


**Figure 6.4:** Relative phase composition of TERMCUT on representative benchmark instances. Terminal overhead remains a substantial fraction of the total runtime across all terminal counts.

The localisation-based variants show a different phase profile as seen in Figure 6.5. The connected-component computation between the two phases remains small throughout all tested instances and terminal counts. More precisely, the connected-component phase contributes about 8-17% at $k = 8$, decreases to roughly 4-12% at $k = 32$, and is only around 3-8% at $k = 128$ on the representative instances shown. This is consistent with the algorithmic structure, as the computation consists of one graph traversal after the union of all group cuts has been determined.

The main cost for the localisation-based variants lies in the localisation phase and its subsequent recovery phase. On the representative plots in Figure 6.5, the localisation phase contributes about 47-77% at $k = 8$, rises to roughly 67-89% at $k = 32$, and reaches approximately 70-91% at $k = 128$, depending on the graph. Since this phase is identical in ISOCUT and BATCHISOCUT, the close runtime values of the two methods can largely be explained by this shared bottleneck.

The difference between the two localisation-based variants appears only in the recovery phase. ISOCUT performs one reduced recovery computation per terminal, whereas BATCHISOCUT combines these into one disjoint-union computation. Whenever the recovery phases contributes a noticeable share of the total runtime, batching of the computations yields a visible improvement. By contrast, on instances where the localisation phase already dominated the runtime, the additional gain from batching remains modest even if the

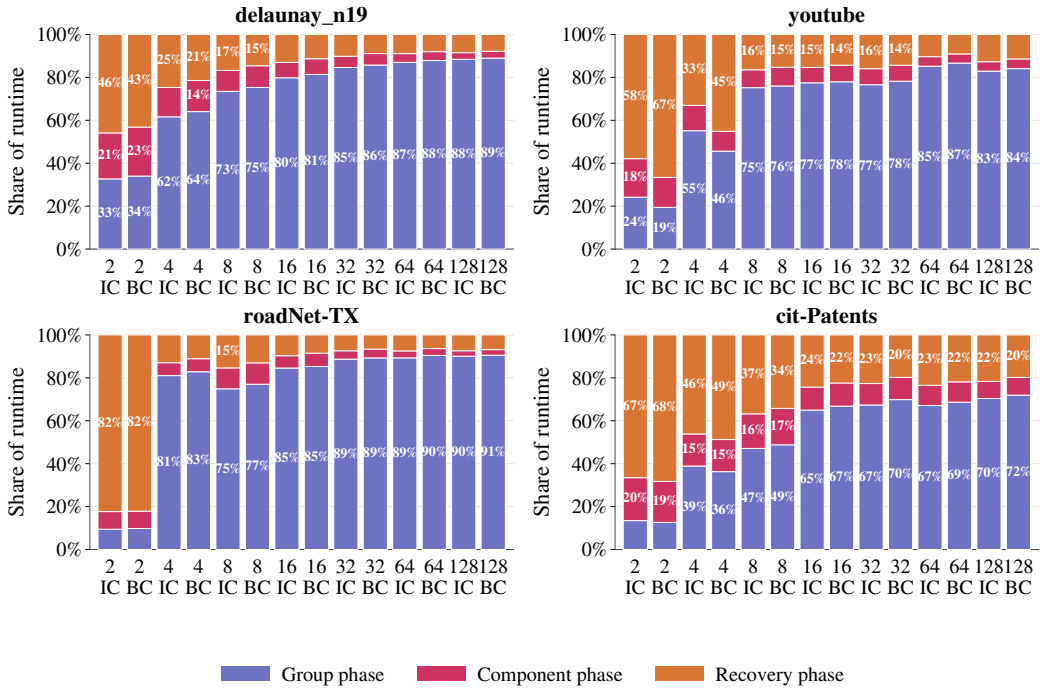


**Figure 6.5:** Relative phase composition of IsoCut and BatchIsoCut on representative benchmark instances. The localisation phase dominates, while the connected-component phase remains small.

recovery stage itself is improved.

Overall, the phase-wise measurements provide a consistent explanation for the sequential runtime behaviour. TermCut mainly suffers from repeatedly solving and rebuilding the full terminal-wise instances, which causes runtime to grow quickly with $k$ for all phases of the algorithm. On the other hand, IsoCut and BatchIsoCut invest additional time to compute a localised solution space, but are able to amortise this cost by solving the recovery stage on either smaller or fewer instances. The extra time spent computing the connected components remains relatively cheap, while the batching of the second phase yields an additional but usually negligible improvement of BatchIsoCut over IsoCut.

This interpretation is also supported by the aggregate runtime comparisons over the full benchmark set. Across the high-$k$ benchmark cases, the better localisation-based variant outperforms TermCut on $98.1\%$ of the complete cases, and at $k = 128$ the localisation-based approach improves runtime over TermCut by a factor of $3.53$ on average. By contrast, batching yields a smaller additional improvement: across all complete sequential benchmark cases, BatchCut is faster than IsoCut on $65.1\%$ of the cases, with a mean relative gain of $0.1\%$. Restricting to the high-$k$ cases, the corresponding mean gain increases to $1.8\%$.

**Answer to RQ2.** The refinement based on the Isolating Cut Lemma yields a substantial practical benefit for larger terminal counts. Across the high-$k$ cases, the better structured variant beats TERMCUT on $98.1\%$ of the complete benchmark cases, and at $k = 128$ it improves runtime by a factor of $3.53$ on average against TERMCUT. Hence, the additional localisation work is amortised effectively once the terminal count is sufficiently large.

**Answer to RQ3.** Batching the recovery phase gives an additional but smaller improvement than localisation itself. Across all complete sequential cases, BATCHCUT is faster than ISOCUT on $65.1\%$ of the cases, with a mean relative gain of $0.1\%$. On high-$k$ cases, the corresponding mean gain is $1.8\%$. Thus, batching is beneficial in practice, but its effect is usually secondary compared with the gain from localisation.

## 6.4 Parallel Scalability

This section addresses **RQ4**. We now compare the effect of the shared-memory parallelisation strategies introduced in the previous chapter for the rest of the algorithms. Particularly, we study how strong the parallel implementations benefit from additional threads, scaling behaviour and how it correlates to the algorithmically independent work exposed by each method. Unless further specified all measured speedups are relative to the corresponding single-thread execution of the same parallel implementation. We again begin with aggregate full-benchmark statistics. Figure 6.7 reports the mean speedup from one to eight threads over all complete benchmark cases for each terminal count $k$, together with the interquartile range across cases. These aggregate values complement the representative instance-level plots shown afterwards.

**Parallel Runtime Behaviour.** Across all complete benchmark cases, the mean 1-to-8-thread speedups are $2.03$ for PTERMCUT, $1.43$ for PISOCUT, and $1.46$ for PBATCHCUT. At $k = 128$, the corresponding mean speedups are $3.83$, $2.08$, and $2.06$, respectively. This confirms that all variants benefit from shared-memory parallelism, but also that TERMCUT parallelises best overall.

We first focus on the overall runtime counts of the parallel variants for threads counts of $1, 2, 4$ and $8$. Figure 6.6 shows representative runtime curves on four selected benchmark instances, namely `delaunay_n19`, `youtube`, `roadNet-TX`, and `cit-Patents`. Overall, all variants benefit from parallel execution, however, the amount of improvement differs considerably between methods and across different graph classes. This is visible on all four representative instances in Figure 6.6. On `delaunay_n19`, PTERMCUT decreases from about $28$ seconds at one thread to roughly $6$ seconds at eight threads,

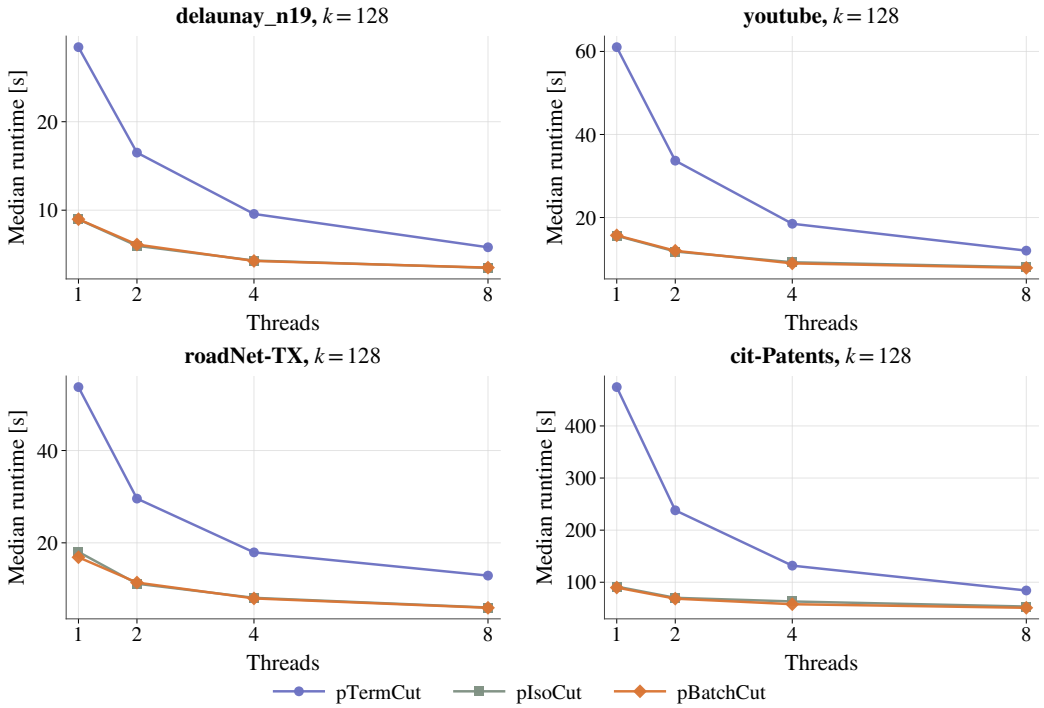


**Figure 6.6:** Median runtime of the parallel variants for thread counts $1, 2, 4$ and $8$ on representative benchmark instances for $k = 128$.

while pIsoCut and pBatchIsoCut drop from about $9$ seconds to around $3.5$ seconds. On `cit-Patents`, pTermCut falls from roughly $460$ seconds to about $85$ seconds, whereas pIsoCut and pBatchIsoCut decrease from around $90$ seconds to the low-$50$ second range. The best speedup values can unsurprisingly be reported for TermCut. The algorithm exposes one independent cut computation per terminal, thus allowing the work to naturally be distributed across the available threads. With an increasing number of terminals, the amount of available work grows proportionally, showing the increasing suitability for parallel execution. The structured variants of IsoCut and BatchIsoCut do also profit from additional threads, but their runtime curves improve more moderately. While both methods expose parallelism in the localisation phase, and for IsoCut also in the recovery phase, they additional retain necessary synchronisation points and shared preprocessing work. Particularly, the computation of the connected components between the two phases remains sequential and the group phase offers merely $\lceil \log_2 k \rceil$ many independent tasks. As a consequence, the benefit is not as large as for the baseline. Overall, the runtime curves indicate the practical benefits and their dependency on how much time is spent in the cut computations. Whenever the runtime is dominated by these tasks, additional threads do reduce wall-clock time noticeably. When other phases dominate, the gains remain more limited.

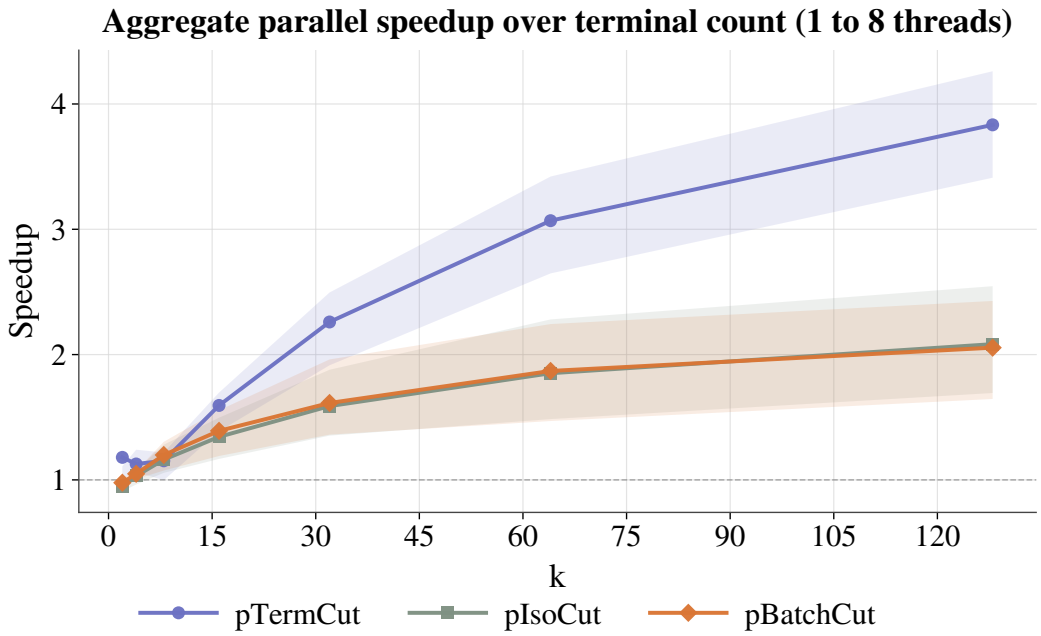


**Figure 6.7:** Aggregate parallel speedup from one to eight threads over terminal count $k$. The solid lines show the mean speedup over all complete benchmark cases $(\texttt{graph}, k)$, the shaded areas indicate the interquartile range across cases.

**Speedup and Scalability.** To further analyse parallel behaviour, Figure 6.8 reports the achieved speedup relative to the one-thread execution. TERMCUT usually achieves the strongest and most regular speedups across the benchmark set, confirming that the method benefits from its simple task structure. As every terminal induces its own independent cut problem, thus growing linearly with larger terminal counts, the available parallel work also grows directly with $k$. For example, at $k = 128$ the achieved speedup on the representative instances is about $4.9$ on `delaunay_n19`, $5.1$ on `youtube`, $4.2$ on `roadNet-TX`, and $5.7$ on `cit-Patents` for PTERMCUT. By contrast, PISOCUT and PBATCHISOCUT usually remain in the range of about $1.8$ to $3.0$, depending on the graph. For ISOCUT, the observed speedups are weaker but still visible. While the methods does benefit from parallelism in both its localisation and recovery phases, the overall gain is restricted by the sequential component-construction step separating them and by the limited amount of independent work in the first phase. Moreover, the size of the terminal-wise recovery tasks may differ significantly for their reduced instances, thus leading to unequal work distribution across threads. Therefore, even though ISOCUT does expose parallel work in two phases, it is not necessarily perfectly balanced. BATCHISOCUT also differentiates itself in its speedup behaviour. As the variant combines the recovery computations into one batched flow instance, the terminal-wise parallelism of the second phase is removed. Consequently, the speedups of BATCHISOCUT are primarily dominated by the parallelism of the group phase and the construction of the batched recovery instance, while the last flow compu-

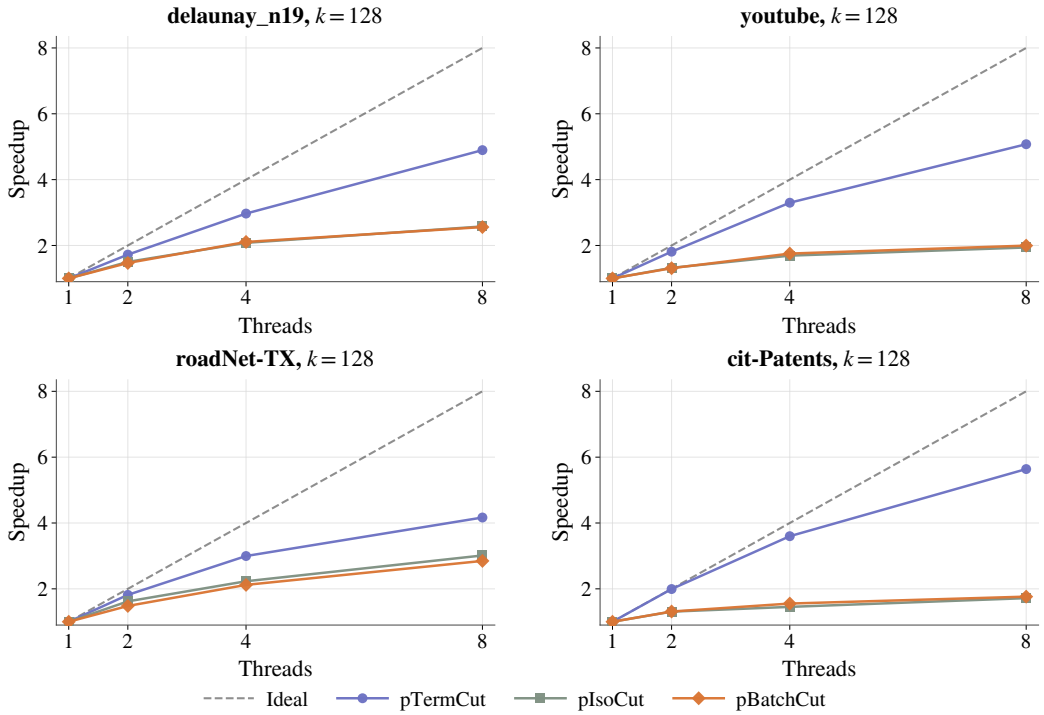


**Figure 6.8:** Speedup of the parallel variants relative to their corresponding one-thread execution on representative benchmark instances for $k = 128$.

tation remains a central but sequential run-through. The extent to which the additional threads thus may reduce the total runtime are limited, especially as the first phase has been sped up substantially with the Isolating Cut Lemma.

Hence, the speedup plots do confirm that the three parallel implementations do not merely differ in their engineering details, but also in the amount and structure of available parallel work. TERMCUT benefits from the largest pool of independent tasks, ISOCUT retains meaningful parallelism in both its phases and BATCHISOCUT trades part of its parallelism for a more compact second phase.

**Thread Imbalance.** Further analysing the sublinear speedups reported lays a deeper focus on the recorded imbalance values and structural limits imposed by the algorithms. A limitation arises from the limited number of independent tasks available in the localisation phase of the structured variants. As this phase computes merely $\lceil \log_2 k \rceil$ group cuts, the amount of parallel work available here remains inherently small for moderate values of $k$. Even for larger terminal counts considered in our experiments (i.e. $k = 128$), this allows only seven independent group-cut tasks to parallelise. Thus, the thread counts beyond this range cannot be expected to substantially improve the first phase. Another limitation arises from unequal task sizes. In TERMCUT, the terminal-wise cut problem may not necessarily be identical in difficulty, but they do tend to be similar enough for the work distribution to

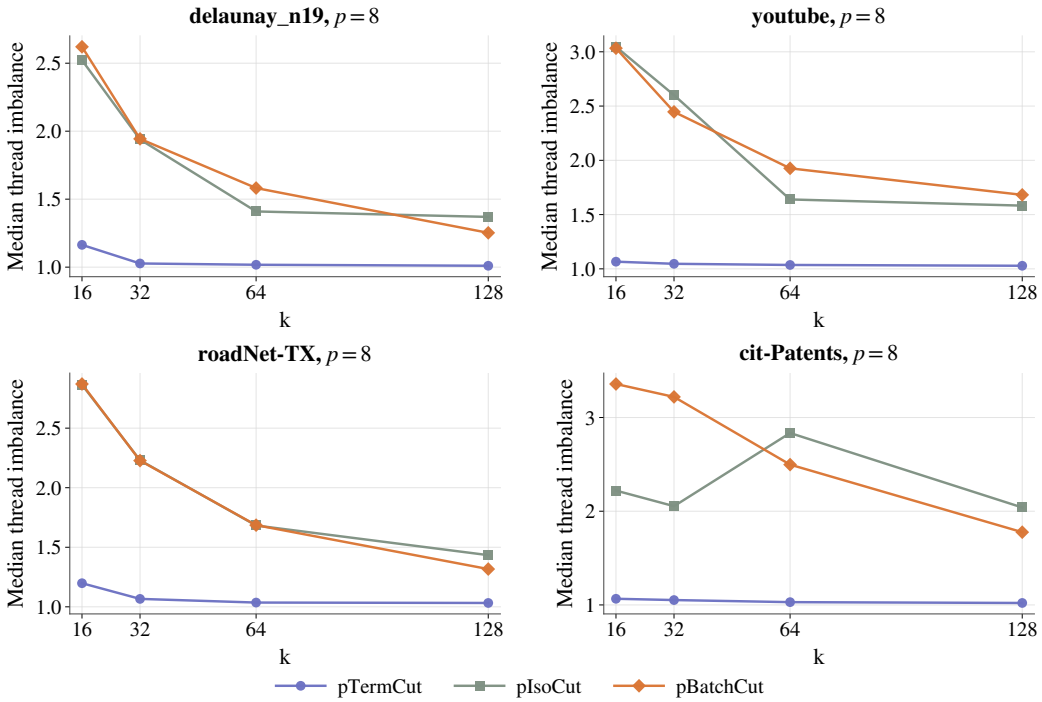


**Figure 6.9:** Measured thread imbalance of the parallel variants for thread count $p = 8$ over increasing terminal counts on representative benchmark instances.

remain reasonably effective. For IsoCut, the sizes of the reduced instances depend on the previously computed localisation group sets $U_v$, which may differ considerably in size from terminal to terminal. This can lead to more pronounced imbalance in the second phase, as some threads may finish their recovery tasks much earlier than others. This effect is reflected with the measured imbalance values and explains why speedup often slows down before all available cores are used to their full extent. The representative plots in Figure 6.9 illustrate this clearly. For pTermCut, the median imbalance stays close to ideal, remaining around 1.0-1.2 across all shown instances. In contrast, pIsoCut and pBatchIsoCut start with imbalance values above 2 on several graphs at $k = 16$ and only decrease gradually as $k$ grows, with `cit-Patents` remaining particularly imbalanced even for larger terminal counts. Finally, BatchIsoCut exhibits another type of limitation. As the final recovery stage is intentionally performed only in one global batched maximum-flow computation, this part of the runtime cannot be distributed across independent terminal-wise tasks. Hence, while batching may improve the proposed algorithmic structure of the second phase, it simultaneously causes the reduction of coarse-grained parallelism available. Therefore, the measured imbalance does not only reflect implementation overhead, but also the deliberate trade-off between fewer flow computations and less task parallelism.

Combined these measurements show that the scalability of the implemented methods is dominated primarily by their combinatorial structure. Their limiting factors are not merely

the overhead of thread management, but rather the amount of independent work exposed by each algorithm. In particular, these structural limitations can be observed for the advanced methods, where their first phase is limited by logarithmically many tasks and their recovery phase either imbalanced for IsoCut or batched into one computation BatchIsoCut.

The parallel execution is clearly beneficial for all three variants but the magnitude of the benefit depends strongly on the corresponding method. TermCut provides the most direct and scalable task parallelism, thus making it the preferable candidate for parallel speedup. The other algorithms also benefit from parallelism, but their improvements are restricted by limited task counts, synchronisation points and thread imbalance in the recovery phase.

The parallel results complement the sequential findings in an important aspect. Sequentially, we report IsoCut and BatchIsoCut to outperform TermCut for larger terminal counts by a large margin because of their superior asymptotic structure. However, in a parallel setting, TermCut recovers part of its disadvantage by exposing more coarse-grained independent work in contrast to its competitors. Thus, the preferable method in a practical setting does not only depend on the graph instance and terminal count, but also on whether sequential efficiency or exploitation of multiple cores is preferred.

**Answer to RQ4.** All three approaches benefit from shared-memory parallelisation, but the gains differ substantially by method. Across all complete cases, the mean 1-to-8-thread speedups are $2.03$ for pTermCut, $1.43$ for pIsoCut, and $1.46$ for pBatchCut. At $k = 128$, the corresponding mean speedups increase to $3.83$, $2.08$ and $2.06$ respectively. This confirms that TermCut parallelises best overall, while the structured variants remain more restricted by limited coarse-grained task parallelism and sequential bottlenecks.

## 6.5 Performance Profiles

While previous sections focused on detailed runtime behaviours for selected representative instances, performance profiles provide a complementary aggregated view across the full benchmark set. Rather than only considering absolute runtime values, they compare how often a method comes close to the best runtime achieved on a given benchmark instance. This makes performance profiles particularly useful when assessing robustness across heterogenous graph classes and terminal amounts, following the benchmarking approach of Dolan and Moré [13]. For a benchmark case $p$ and an algorithm $a$, we denote $t_{p,a}$ as the recorded runtime of $a$ on $p$. Thus, we define the performance ratio

$$r_{p,a} := \frac{t_{p,a}}{\min_b t_{p,b}},$$

where the minimum is aggregated and picked over all algorithms $b$ included in the comparison. Inferring from this for $r_{p,a} = 1$ the algorithm $a$ is the fastest method on a bench-

mark case $p$. The corresponding performance profile of an algorithm $a$ is then defined as

$$\rho_a(\tau) := \frac{1}{|P|}|\{p \in P : r_{p,a} \leq \tau\}|,$$

with $P$ being the set of graph instances under consideration. Hence, $\rho_a(\tau)$ reports the fraction of benchmakr cases on which the algorithm $a$ is at most a factor of $\tau$ slower than the fastest method. Fast rising curves that remain higher indicate methods that are more frequently among the best-performing ones.

For all performance profiles in this section, we define a benchmark case as one pair of (GRAPH, $k$), with the runtime associated with that case as the median runtime over the three seeds. This avoids over-representing individual terminal choices while preserving the overall performance trends of the methods.

**Sequential Performance Profiles.** Figure 6.10 shows the performance profiles of the sequential variants of TERMCUT, ISOCUT and BATCHISOCUT. The profiles reaffirm the trends observed in previous detailed runtime plots. Over all complete benchmark cases, TERMCUT attains the highest fastest-case share, being the fastest method on $48.1\%$ of the cases, and it lies within $10\%$ of the best runtime on $53.6\%$ of the cases. This confirms that the direct baseline is highly competitive on the easier end of the benchmark set, in particular for smaller terminal counts where its low overhead dominates.

However, as the performance ratio increases, the structured variants improve and eventually dominate a larger portion of the benchmark set. This reflects well their stronger behaviour on more complex instances, especially for more terminals where the localisation phase amortises the problem effectively.

ISOCUT and BATCHISOCUT remain close in their curves across the full range of ratios. This behaves consistently to previous phase-wise analysis, where we noted that both methods share the same dominant localisation phase and therefore differ only in the structure of the recovery stage. Thus, the performance profiles confirm the relatively small practical differences between these two algorithms, whereas the main distinction lies between them and the direct baseline TERMCUT.

**Parallel Performance Profiles.** Figure 6.11 reports the corresponding performance profiles for the parallel versions at 8 threads. These again complement earlier runtime and speedup plots. Over all complete parallel benchmark cases at $p = 8$, PTERMCUT attains the largest fastest-case share with $65.0\%$ and lies within $10\%$ of the best runtime on $71.8\%$ of the cases. Specifically, TERMCUT remains highly competitive because it exposes a large number of coarse-grained independent tasks and benefits from shared-memory parallelism. Similarly, the other algorithms remain competitive on complexer benchmark cases, especially when their improved asymptotic structure dominates.

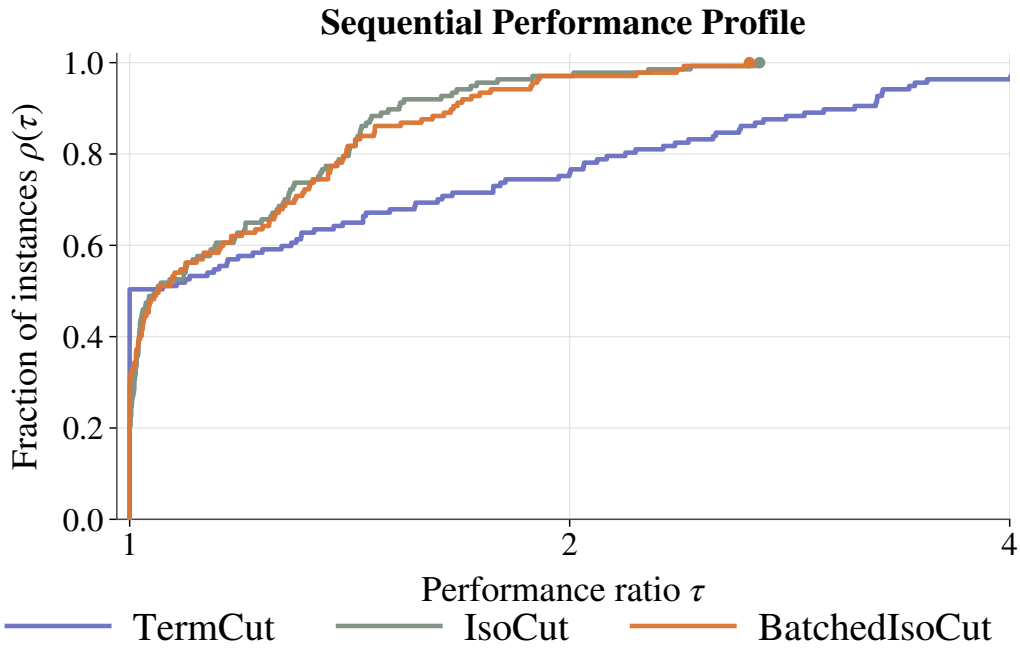


**Figure 6.10:** Performance profiles of the sequential variants over all benchmark cases $(\text{graph}, k)$, using median runtime over the three seeds.

Comparing both sequential and parallel performance profiles therefore reveals an interesting and important practical distinction. Sequentially considered, the structured variants become increasingly favourable as the number of terminal grows larger. However, TERMCUT recovers in parallel because of its simpler structure allowing more workload for parallelism. Thus, the profiles support the conclusion that no single method particularly dominate across all settings. Instead we can deduce that the choice inherently depends on the instance characteristics and whether we want to improve sequential efficiency or shared-memory scalability. The performance profiles offer a concise aggregation of the benchmark results, confirming that the baseline is the best method for the easier end of the instances while ISOCUT and BATCHCUT remain competitive on a substantial portion of the benchmark set, especially for larger terminal counts.

## 6.6 Peak Memory Usage

This section addresses **RQ5**. We compare the peak memory usage of the sequential and parallel variants and study how localisation, batching, and parallelisation affect the memory footprint in practice. On top of analysing runtime on benchmark instances, the peak memory usage can also be a relevant performance criterion. We therefore compute the peak resident set size (RSS) in MiB of both the sequential and parallel methods and compare in which way these scale for different implementations. Throughout this section, the *me-*

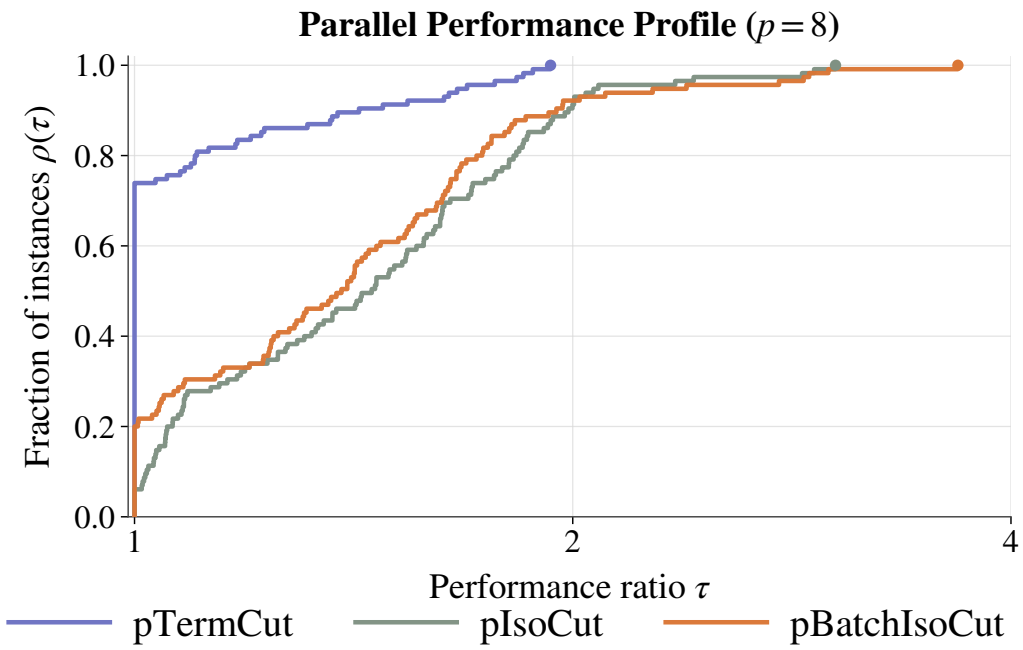


**Figure 6.11:** Performance profiles of the parallel variants at 8 threads over all benchmark cases (graph, $k$), using median runtime over the three seeds.

*dian* peak RSS is used as the primary statistic, as several configurations exhibit occasional high-memory runs.

**Sequential Peak RSS.** The sequential algorithms show comparatively stable peak-memory behaviour across all terminal counts. As shown in Figure 6.12, the sequential algorithms exhibit comparatively stable absolute peak RSS over all terminal counts, while the memory per terminal decreases strongly with growing $k$. Amongst them TERMCUT attains the lowest overall median peak RSS at 469.68 MiB, followed by BATCHISOCUT with 532.79 MiB and ISOCUT with 541.17 MiB. Therefore, the absolute differences are moderate and all sequential implementations remain in a similar memory range. Differences are far more pronounced when normalising by the number of terminals. According to Table 6.3, the median memory per terminal decreases from 231.60 MiB to 3.67 MiB for TERMCUT, from 270.71 MiB to 4.22 MiB for ISOCUT, and from 266.32 MiB to 4.17 MiB for BATCHISOCUT when moving from $k = 2$ to $k = 128$. This indicates that a substantial portion of the sequential memory footprint is fixed overhead rather than cost that grows proportionally with the terminal count.

**Parallel Peak RSS.** Figure 6.12 also shows that the parallel variants have a consistently higher absolute peak RSS than their sequential counterparts, even though the memory per terminal again decreases with increasing $k$. Over all available runs, PBATCHISOCUT at-

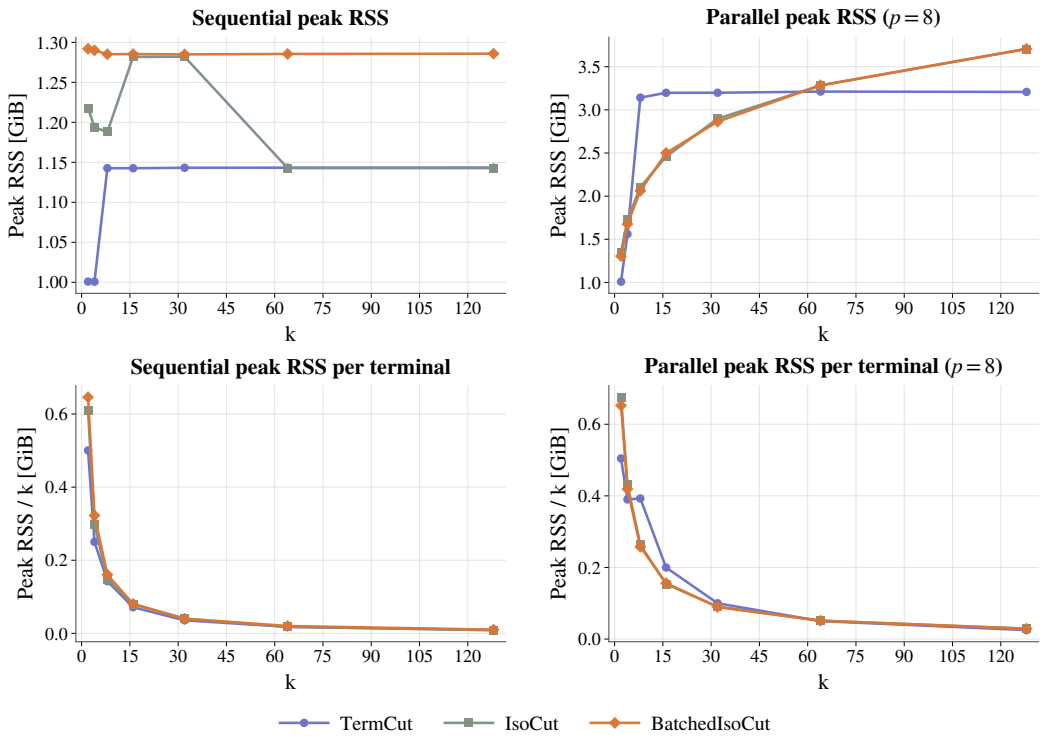


**Figure 6.12:** Median peak RSS over terminal counts $k$ for the sequential and parallel variants. The top row shows absolute peak RSS, while the bottom row normalises peak RSS by the number of terminals. The left column reports sequential runs, and the right column reports parallel runs at $p = 8$.

tains the lowest overall median peak RSS among the parallel methods at 741.65 MiB, followed by pIsoCut with 821.54 MiB and pTermCut with 902.00 MiB. Thus, introducing shared-memory parallelism shifts the memory usage upward for all algorithmic variants.

Relative to their sequential counterparts, the median peak RSS increases by factors of 1.39 for pBatchIsoCut, 1.52 for pIsoCut, and 1.92 for pTermCut. Table 6.3 furthermore shows that the median peak RSS of the parallel variants rises with increasing terminal count: pBatchIsoCut grows from 531.35 MiB at $k = 2$ to 928.39 MiB at $k = 128$, pIsoCut from 663.33 MiB to 976.74 MiB, and pTermCut from 615.00 MiB to 1137.84 MiB. Even though memory per terminal decreases for larger $k$, the total memory footprint of the parallel methods remains clearly above the sequential baseline. This indicates that parallelism introduces substantial additional memory overhead, for example through concurrent cut computations, replicated temporary state, and thread-local auxiliary data structures.

**Instance-dependent behaviour.** Even though the aggregated medians provide a robust overall picture, memory usage remains strongly instance-dependent. Table 6.4 illustrates this for $k = 32$ and 8 threads. On smaller and moderately sized graphs such

as delaunay_n15, luxembourg-sorted, or Georgetown15, all parallel methods require comparatively little memory. In contrast, larger road, web, social, and kernel instances such as road_usa, eur, and skkernel reach peak RSS values in the tens of thousands of MiB.

The same table also shows that no single parallel method dominates on every graph. In many cases, pIsoCut and pBatchIsoCut remain close to one another, while pTermCut is either competitive or substantially more memory-intensive depending on the graph structure. For example, on road_usa, the structured variants require around 32–33 GiB, whereas pTermCut reaches roughly 47 GiB. This confirms that peak memory is determined not only by the algorithmic design, but also by the graph structure and the amount of parallel work exposed by the instance.

**Answer to RQ5.** Localisation, batching, and especially parallelisation increase the memory footprint in practice. The sequential variants remain in a comparatively similar range, with overall median peak RSS values of 469.68 MiB for TermCut, 541.17 MiB for IsoCut, and 532.79 MiB for BatchCut. In contrast, the parallel variants are higher by factors of 1.92, 1.52 and 1.39 relative to their sequential counterparts. Thus, the runtime benefits of parallelism come at the cost of a noticeably increased peak memory footprint.

## 6.7 Comparison with an External Exact Solver

This section addresses RQ6. We compare the proposed methods against an external exact solver in order to assess the trade-off between runtime and solution quality. Complementing the evaluation of our own implementation, we additionally compared against the public KTerminalCut implementation provided by Velendnitsky [26]. Following comparison should be treated carefully with respect to the algorithmic objective focused on. The primary focus of our implementation did not lie with solving the minimum $k$-terminal cut problem exactly, but with computing the family of isolating cuts for a terminal set $R$ in a scalable and practically efficient manner. Arising from this computation, we automatically get a feasible multiterminal cut, which makes the $k$-terminal-cut setting a meaningful application context for evaluation of our algorithm. Accordingly, we use the external solver to assess the quality of the terminal-separating cuts induced by our methods, rather than claiming that both approaches optimise the same construction objective. As this solver follows a substantially different regime and exhibits significantly higher runtimes in preliminary tests, the comparison was restricted to three representative graph instances, namely 4elt, delaunay_n15 and Georgetown15. Each graph reuses the same terminal sets as our own experiments by exporting the identical terminal selections provided for

the seeds used and is evaluated for $k \in \{4, 8, 16\}$. For our side, we report the weighted value of the final recovered cut edge set. The exact solver reports the certified optimum whenever optimality was proven and otherwise we report the best upper bound available at termination. Thus we can compare experiments on the same terminal sets with their directly comparable cut values. Table 6.5 shows all values reported for the runs where the exact solver certified optimality for its solution. These results show the trade-off expected between runtime and solution quality. For $k = 4$ on `4elt.graph`, our method remains substantially faster in computation, but yields cut values between $1.33$ and $1.39$ times the optimum value. On `delaunay_n15` for $k = 4$, we report ratios of $1.41$ and $1.50$, while for $k = 8$ they improve roughly to $1.19$ and $1.22$. Analysing this from a practical viewpoint makes the runtime gap even more substantial. While our implementation finishes in the fraction of a second, the exact solver requires several hundred seconds on the same instance. In situations where timely solutions are paramount, quickly obtaining a feasible cut can be preferable in comparison to pursuing exact optimality at a significantly higher computational cost. Additionally, Table 6.6 shows the runs where the time limit of around 600 seconds was reached for the exact solver. In these cases, the relevant reference to consider is the best upper bound returned at termination. On `4elt` we report for $k = 8$ and $k = 16$, as well as other terminal counts on `delaunay_n15`, that our recovered cut values coincide exactly with the best feasible solution known to the exact solver at termination. Noticeable is the large runtime gap here for the differing implementations, with the exact solver not solving the optimality in the larger timeframe. The other evaluated instance `Georgetown15` behaves differently. While the runtime for our implementation remains fast, the recovered cut values are substantially larger than the reported values of the exact solver. Therefore we interpret `Georgetown15` as a weak case for the implemented isolating-cut based construction and report the detailed experimental numbers separately in the Appendix. Overall, we can see that the scalable isolating-cut computation can yield very competitive terminal-separating solutions on some instance, but can also exhibit clearly weak cases on other instances in terms of induced cut quality.

**Answer to RQ6.** The comparison with the external exact solver shows a clear trade-off between runtime and solution quality. On the tested instances, the proposed isolating-cut-based methods are substantially faster and often produce competitive terminal-separating cuts. At the same time, the exact solver can return better solutions or certified optima on the instances it is able to solve. Hence, the proposed methods should be understood as fast and scalable practical approaches, rather than as replacements for exact optimisation.

**Table 6.1:** Benchmark graphs used in our experiments. Here, $n$ denotes the number of vertices and $m$ the number of edges.

| Graph | $n$ | $m$ | Type |
|---|---|---|---|
| Mesh / Geometric / Structural Instances | | | |
| delaunay_n15 | 32 768 | 98 274 | Geometric |
| smallworld | 100 000 | 499 998 | Synthetic |
| 144-sorted | 144 649 | 1 074 393 | Structural |
| wave-sorted | 156 317 | 1 059 331 | Structural |
| turtle-sorted | 267 534 | 401 178 | Structural |
| delaunay_n19 | 524 288 | 1 572 823 | Geometric |
| ecat-sorted | 684 496 | 1 026 744 | Structural |
| Circuit / Numerical Instances | | | |
| netz4504.wd | 1 961 | 2 578 | Matrix-derived |
| memplus | 17 758 | 54 196 | Matrix-derived |
| G2_circuit | 150 102 | 288 286 | Circuit |
| shipsec5 | 179 860 | 4 966 618 | Matrix-derived |
| Bump_2911 | 2 852 430 | 62 409 240 | Circuit |
| skkernel | 3 200 806 | 90 931 906 | Kernel |
| Citation / Collaboration Instances | | | |
| coAuthorsDBLP | 299 067 | 977 676 | Collaboration |
| coPapersCiteseer | 434 102 | 16 036 720 | Citation |
| coPapersDBLP | 540 486 | 15 245 729 | Collaboration |
| cit-Patents | 3 774 768 | 16 518 947 | Citation |
| Web / Communication Instances | | | |
| email-EuAll | 16 805 | 60 260 | Communication |
| enron | 69 244 | 254 449 | Communication |
| wiki-Talk | 232 314 | 1 458 806 | Communication |
| cnr-2000 | 325 557 | 2 738 969 | Web |
| eu-2005 | 862 664 | 16 138 468 | Web |
| Road Instances | | | |
| luxembourg-sorted | 114 599 | 119 666 | Road |
| roadNet-PA | 1 088 092 | 1 541 898 | Road |
| roadNet-TX | 1 379 917 | 1 921 660 | Road |
| great-britain-sorted | 7 733 822 | 8 156 517 | Road |
| eur | 18 029 721 | 22 217 686 | Road |
| road_usa | 23 947 347 | 28 854 312 | Road |
| Social Instances | | | |
| Georgetown15 | 9 414 | 425 638 | Social |
| Virginia63 | 21 325 | 698 178 | Social |
| Indiana69 | 29 747 | 1 305 765 | Social |
| UIllinois20 | 30 809 | 1 264 428 | Social |
| loc-gowalla_edges | 196 591 | 950 327 | Social |
| youtube | 1 134 890 | 2 987 623 | Social |
| soc-lastfm | 1 191 805 | 4 519 330 | Social |

**Table 6.2:** Overall peak-memory summary grouped by algorithm family. Reported values are medians over all available runs and seeds.

| Algorithm | Median peak memory [MB] | Max peak memory [MB] | Median memory / terminal [MB] | Median memory / thread [MB] |
|---|---|---|---|---|
| K-Cut Variants | | | | |
| TC | 469.68 | 32283.60 | 42.97 | 469.68 |
| pTC | 902.00 | 393344.00 | 69.26 | 363.20 |
| Isolating-Cut Variants | | | | |
| IC | 541.17 | 32371.60 | 49.81 | 541.17 |
| pIC | 821.54 | 100557.00 | 62.80 | 314.81 |
| Batched Isolating-Cut Variants | | | | |
| BIC | 532.79 | 34008.10 | 42.65 | 532.79 |
| pBIC | 741.65 | 100554.00 | 57.05 | 294.07 |

**Table 6.3:** Peak memory grouped by algorithm family and terminal count. Values are medians over all available graphs, thread counts, and seeds.

| Algorithm | $k$ | Median peak memory [MB] | Median memory / terminal [MB] |
|---|---|---|---|
| TermCut Variants | | | |
| TC | 2 | 463.19 | 231.60 |
| TC | 4 | 463.17 | 115.79 |
| TC | 8 | 469.62 | 58.70 |
| TC | 16 | 469.65 | 29.35 |
| TC | 32 | 469.66 | 14.68 |
| TC | 64 | 469.76 | 7.34 |
| TC | 128 | 470.14 | 3.67 |
| pTC | 2 | 615.00 | 307.50 |
| pTC | 4 | 812.80 | 203.20 |
| pTC | 8 | 936.12 | 117.01 |
| pTC | 16 | 1112.76 | 69.55 |
| pTC | 32 | 1139.95 | 35.62 |
| pTC | 64 | 1113.64 | 17.40 |
| pTC | 128 | 1137.84 | 8.89 |
| Isolating-Cut Variants | | | |
| IC | 2 | 541.42 | 270.71 |
| IC | 4 | 541.68 | 135.42 |
| IC | 8 | 541.19 | 67.65 |
| IC | 16 | 541.00 | 33.81 |
| IC | 32 | 540.73 | 16.90 |
| IC | 64 | 540.32 | 8.44 |
| IC | 128 | 539.70 | 4.22 |
| pIC | 2 | 663.33 | 331.66 |
| pIC | 4 | 710.04 | 177.51 |
| pIC | 8 | 797.63 | 99.70 |
| pIC | 16 | 890.18 | 55.64 |
| pIC | 32 | 950.24 | 29.70 |
| pIC | 64 | 969.27 | 15.14 |
| pIC | 128 | 976.74 | 7.63 |
| Batched Isolating-Cut Variants | | | |
| BIC | 2 | 532.63 | 266.32 |
| BIC | 4 | 534.15 | 133.54 |
| BIC | 8 | 532.57 | 66.57 |
| BIC | 16 | 534.09 | 33.38 |
| BIC | 32 | 532.72 | 16.65 |
| BIC | 64 | 534.03 | 8.34 |
| BIC | 128 | 533.59 | 4.17 |
| pBIC | 2 | 531.35 | 265.67 |
| pBIC | 4 | 653.39 | 163.35 |
| pBIC | 8 | 745.52 | 93.19 |
| pBIC | 16 | 873.73 | 54.61 |
| pBIC | 32 | 898.18 | 28.07 |
| pBIC | 64 | 932.35 | 14.57 |
| pBIC | 128 | 928.39 | 7.25 |

**Table 6.4:** Peak memory in MB by graph for $k = 32$ and 8 threads. Values are medians over seeds.

| Graph | Type | PKIC | PIC | PBIC |
|---|---|---|---|---|
| 144-sorted | Structural | 1387.66 | 1010.60 | 1014.20 |
| Bump_2911 | Circuit | 78276.50 | 57783.70 | 57693.20 |
| G2_circuit.mtx | Circuit | 423.82 | 295.99 | 295.50 |
| Georgetown15 | Road | 526.67 | 383.97 | 391.50 |
| Indiana69 | Road | 1603.73 | 1175.91 | 1177.07 |
| UIllinois20 | Road | 1557.04 | 1153.18 | 1163.11 |
| Virginia63 | Road | 864.22 | 637.69 | 646.79 |
| cit-Patents | Citation | 22363.30 | 15911.50 | 15917.00 |
| cnr-2000 | Web | 3511.83 | 2468.44 | 2543.13 |
| coAuthorsDBLP | Collaboration | 1349.80 | 1062.39 | 945.10 |
| coPapersCiteseer | Citation | 19710.60 | 14488.50 | 14609.90 |
| coPapersDBLP | Citation | 18860.60 | 13889.30 | 14009.10 |
| delaunay_n15 | Geometric | 140.00 | 100.16 | 99.90 |
| delaunay_n19 | Geometric | 2189.76 | 1526.73 | 1525.00 |
| ecat-sorted | Structural | 1609.97 | 1121.04 | 1119.89 |
| email-EuAll | Social | 83.00 | 62.72 | 62.73 |
| enron | Communication | 345.16 | 249.43 | 244.50 |
| eu-2005 | Web | 20055.60 | – | – |
| eur | Road | 36194.25 | 25488.40 | 24976.55 |
| great-britain-sorted | Road | 13733.30 | 9429.47 | 9420.70 |
| loc-gowalla_edges | Social | 1263.98 | 1010.66 | 899.45 |
| luxembourg-sorted | Road | 202.32 | 141.00 | 140.00 |
| memplus | Circuit | 74.00 | 58.52 | 56.71 |
| netz4504.wd | Circuit | 13.00 | 13.25 | 13.25 |
| roadNet-PA | Road | 2417.50 | 1672.96 | 1671.42 |
| roadNet-TX | Road | 3028.55 | 2188.78 | 2092.19 |
| road_usa | Road | 47096.20 | 32611.90 | 32505.75 |
| shipsec5.mtx | Circuit | 6150.35 | 4314.99 | 4315.57 |
| skkernel | Kernel | 111977.00 | 79496.80 | 78206.90 |
| smallworld | Synthetic | 674.49 | 557.46 | 495.66 |
| soc-lastfm | Social | 6123.63 | 4262.57 | 4260.84 |
| turtle-sorted | Structural | 631.23 | 440.25 | 439.75 |
| wave-sorted | Structural | 1386.49 | 1022.00 | 1017.51 |
| wiki-Talk | Social | 1883.42 | 1339.18 | 1333.43 |
| youtube | Social | 4220.05 | 2932.00 | 2929.12 |

**Table 6.5:** Comparison with the external exact solver on runs where optimality was certified. The value reported for our implementation is the weighted value of the final recovered cut edge set.

| Seed | $k$ | Our runtime [s] | Exact runtime [s] | Our value | Exact optimum | Ratio |
|---|---|---|---|---|---|---|
| | | | `4elt` | | | |
| 78 | 4 | 0.09 | 16.49 | 22.00 | 16.00 | 1.38 |
| 399 | 4 | 0.04 | 69.15 | 24.00 | 18.00 | 1.33 |
| 1109 | 4 | 0.06 | 72.14 | 25.00 | 18.00 | 1.39 |
| | | | `delaunay_n15` | | | |
| 78 | 4 | 0.11 | 62.57 | 28.00 | 19.00 | 1.47 |
| 399 | 4 | 0.14 | 68.26 | 27.00 | 18.00 | 1.50 |
| 1109 | 4 | 0.13 | 67.18 | 24.00 | 17.00 | 1.41 |
| 399 | 8 | 0.16 | 644.92 | 50.00 | 41.00 | 1.22 |
| 1109 | 8 | 0.14 | 635.06 | 50.00 | 42.00 | 1.19 |

**Table 6.6:** Comparison with the external exact solver on runs where the exact solver did not certify optimality within the allotted runtime. In these cases, the reference value is the best upper bound reported at termination.

| Seed | $k$ | Our runtime [s] | Exact runtime [s] | Our value | Best upper bound | Match |
|---|---|---|---|---|---|---|
| | | | `4elt` | | | |
| 78 | 8 | 0.07 | 617.34 | 46.00 | 46.00 | yes |
| 399 | 8 | 0.05 | 618.97 | 48.00 | 48.00 | yes |
| 1109 | 8 | 0.06 | 610.39 | 49.00 | 49.00 | yes |
| 78 | 16 | 0.07 | 618.22 | 91.00 | 91.00 | yes |
| 399 | 16 | 0.09 | 620.49 | 93.00 | 93.00 | yes |
| 1109 | 16 | 0.07 | 619.24 | 97.00 | 97.00 | yes |
| | | | `delaunay_n15` | | | |
| 78 | 8 | 0.18 | 649.83 | 54.00 | 54.00 | yes |
| 78 | 16 | 0.20 | 661.47 | 104.00 | 104.00 | yes |
| 399 | 16 | 0.19 | 660.04 | 103.00 | 103.00 | yes |
| 1109 | 16 | 0.19 | 666.76 | 93.00 | 93.00 | yes |

CHAPTER 7

# Discussion

## 7.1 Conclusion

This thesis studied the practical behaviour of different algorithmic approaches for computing the family of isolating cuts for a terminal set $R$. In particular, the focus was on the engineering trade-offs between sequential and parallel implementations. The experimental evaluation showed that no single algorithm dominates across all benchmark instances and performance criteria. Instead, the most suitable method depends on the specific optimisation goal, including runtime, peak memory usage, robustness, scalability, and the amount of available hardware parallelism.

A central result of this work is that all considered methods exhibit clear trade-offs. For small terminal counts, TERMCUT is highly competitive: across the complete low-$k$ benchmark cases with $k \in \{2, 4, 8\}$, it is faster than the better localisation-based variant on $96.9\%$ of the cases. However, this picture reverses for larger terminal counts. Across the complete high-$k$ cases with $k \in \{32, 64, 128\}$, the better of ISOCUT and BATCHISOCUT is faster than TERMCUT on $98.1\%$ of the cases, and TERMCUT is slower by a factor of $2.40$ on average. At $k = 128$, this gap increases further, where TERMCUT is slower by a factor of $3.53$ on average than the better localisation-based variant. Thus, the direct baseline is attractive for small terminal sets because of its low overhead, whereas the localisation-based methods are clearly preferable once the terminal count becomes large.

The results also show that batching provides an additional, but distinctly smaller, benefit than localisation itself. Across all complete sequential benchmark cases, BATCHISOCUT is faster than ISOCUT on $64.4\%$ of the cases, but the mean relative improvement is only $0.1\%$. Restricting attention to the high-$k$ cases, the mean gain rises to $1.8\%$, and at $k = 128$ it is $1.3\%$. Hence, batching is beneficial in practice, but its effect is usually secondary compared with the gain obtained from the localisation phase itself.

The parallel variants can reduce runtime substantially in suitable scenarios, but the benefits again depend strongly on the algorithmic structure. Across all complete benchmark

cases, the mean speedup from one to eight threads is $2.03$ for pTERMCUT, compared with $1.43$ for pISOCUT and $1.46$ for pBATCHISOCUT. At $k = 128$, the corresponding mean speedups rise to $3.83$, $2.08$, and $2.06$, respectively. This confirms that TERMCUT benefits most from shared-memory parallelism because it exposes one largely independent task per terminal, whereas the structured variants are more constrained by limited coarse-grained task parallelism, synchronisation points, and load imbalance.

These runtime improvements come at a clear memory cost. On the sequential side, the overall median peak RSS values are $469.68$ MiB for TERMCUT, $541.17$ MiB for ISOCUT, and $532.79$ MiB for BATCHISOCUT. For the parallel variants, the corresponding medians increase to $902.00$ MiB, $821.54$ MiB, and $741.65$ MiB. Relative to their sequential counterparts, this corresponds to memory overhead factors of $1.92$ for pTERMCUT, $1.52$ for pISOCUT, and $1.39$ for pBATCHISOCUT. Thus, while parallelisation can improve wall-clock time, it also increases the memory footprint noticeably.

The performance-profile analysis further underlines that there is no universal winner. Over all complete sequential benchmark cases, TERMCUT attains the highest fastest-case share at $48.1\%$, while BATCHISOCUT is within $10\%$ of the best runtime on $55.7\%$ of the cases. In the parallel setting at $8$ threads, pTERMCUT becomes even more dominant in this sense, achieving the fastest runtime on $65.0\%$ of the cases and lying within $10\%$ of the best on $71.8\%$ of the cases. This again illustrates that the preferable method depends on whether one prioritises low-overhead sequential execution or stronger shared-memory scalability.

Our results further underline that algorithm performance depends strongly on instance characteristics. Graph family, instance size, and overall problem complexity have a substantial influence on both runtime and memory behaviour. Additionally, the available hardware affects the relative attractiveness of different methods in practice. While some algorithms benefit strongly from additional parallel resources, others show only limited gains or incur substantial overhead. This indicates that practical performance is shaped not only by the theoretical design of the algorithm, but also by the structure of the input instance and the target machine.

However, the conclusions of this thesis should be interpreted in the context of the experimental scope. The evaluation is based on a fixed benchmark set, a specific hardware environment, and the concrete implementations considered in this work. As a result, the observed trade-offs should be understood as empirical findings for this setting rather than as universal statements. Different graph classes, larger instances, or alternative implementation choices may shift the balance between the methods. Nevertheless, the experiments provide a consistent overall picture of the strengths and weaknesses of the investigated approaches and show that practical performance is determined by a combination of algorithmic design, instance complexity, and hardware constraints.

Overall, this thesis demonstrates that the choice of algorithm for computing isolating cuts should be guided by the target scenario rather than by a single global ranking. The main contribution is therefore not the identification of one universally best method, but a better understanding of when and why different methods are preferable in practice.

## 7.2 Future Work

A natural progression for future work is to increase the amount of exploitable parallelism inside the cut computations themselves. In particular, parallelising the underlying maximum-flow computations appears to be a promising next step. This observation is specifically relevant for BATCHISOCUT, as the amount of independent parallelise-able work is far more limited here in comparison to other methods. Additional parallelism at the level of the flow computation could potentially improve hardware utilisation and further reduce runtime. Another interesting direction is the further parallelisation of the localisation phase. Since localisation still contributes a non-negligible part of the total work, improving its parallel execution could reduce remaining sequential bottlenecks and strengthen the scalability of the overall approach. Beyond direct parallelism, the experimental results also suggest that adaptive method selection would be a possible avenue for future work. As no algorithm dominates solely across all instances, one could aim to design a prediction-based or portfolio-based approach that selects the most suitable method depending on instance features as graph size, terminal count or graph family. This could improve robustness across heterogenous benchmark sets. Another further important topic is the reduction of memory overhead in any parallel implementation. The experiments highlight that parallelisation often leads to a substantial increase in peak memory usage. Future work could therefore investigate more memory-efficient data structures, reduced duplication of state or scheduling strategies that better balance concurrency and memory consumption. Additionally, more architecture-aware optimisation, such as cache-friendly data layouts, NUMA-aware memory placements and a more careful treatment of memory locality in multicore systems, may lead to further improvements. Such optimisations are likely to be particularly relevant for the larger benchmark instances. Finally, the empirical study could be extended to a broader range of graph classes and larger benchmark sets. This characterisation would help validate the trends observed in this thesis, to better understand the dependence on instance structure and to provide a stronger basis for future adaptive approaches.

# Zusammenfassung

Das MULTITERMINAL-CUT-PROBLEM fordert eine Menge von Kanten mit minimalem Gewicht, deren Entfernung eine gegebene Menge von Endknoten paarweise voneinander trennt. Da das Problem bei drei oder mehr Endknoten NP-schwer ist, konzentriert sich diese Arbeit auf die skalierbare Berechnung von isolierenden Schnitten mit minimalem Gewicht in gewichteten, ungerichteten Graphen als praktische Grundlage für die Trennung mehrerer Endknoten. Wir untersuchen drei auf dem Maximum-Flow-Verfahren basierende Ansätze zur Berechnung der Familie isolierender Schnitte für eine Knotenmenge $R$ der Größe $k$. Als direkte Basis wird für jeden Knoten ein Minimalschnitt berechnet, was zu $k$ Maximum-Flow-Berechnungen auf dem vollständigen Graphen führt. Aufbauend auf den strukturellen Ideen hinter dem Isolating-Cut-Lemma von Li und Panigrahi berechnet ein zweiter Ansatz zunächst $\lceil \log_2 k \rceil$ Gruppenschnitte, nutzt diese, um jeden Knoten in einer reduzierten Instanz zu lokalisieren, und ermittelt anschließend die isolierenden Schnitte. Ein dritter Ansatz erweitert diese Idee, indem er die abschließende Ermittlungsphase in eine einzige Maximalfluss-Berechnung integriert.

Alle Ansätze werden in sequenzieller und paralleler Form implementiert und für die Anzahl der Knoten von 2 bis 128 anhand von Benchmark-Instanzen bewertet, die von kleinen Graphen mit nur wenigen tausend Knoten bis hin zu sehr großen Instanzen aus der Praxis reichen. Die Ergebnisse zeigen einen deutlichen Kompromiss zwischen einfachen direkten Methoden und stärker strukturierten Lokalisierungsmethoden. Bei kleinen Knotenmengen ist der direkte Ansatz oft am schnellsten, da er zusätzlichen Overhead vermeidet. Bei größeren Terminalanzahlen skalieren die lokalisierungsbasierten Varianten jedoch wesentlich besser. über alle Fälle des vollständigen High-$k$-Benchmarks mit $k \in 32, 64, 128$ hinweg ist die bessere der beiden strukturierten Varianten in $98,1\%$ der Fälle schneller als die direkte Baseline und verbessert bei $k = 128$ die Laufzeit im Durchschnitt um den Faktor $3,53$. Das Batching der Wiederherstellungsphase bietet einen zusätzlichen, wenn auch geringeren Vorteil, während die parallelen Varianten die Laufzeit weiter reduzieren und bei $k = 128$ mittlere Beschleunigungen von einem bis acht Threads von $3,83$ für PTERMCUT, $2,08$ für PISOCUT und $2,06$ für PBATCHISOCUT erzielen. Insgesamt zeigen die Experimente, dass keine einzelne Methode in allen Konfigurationen dominiert, stattdessen hängt der am besten geeignete Algorithmus von der Anzahl der Knoten, der Grapheninstanz und davon ab, ob das Hauptziel ein geringer sequenzieller Overhead oder eine stärkere Skalierbarkeit im Shared-Memory-Modus ist.

APPENDIX A

# Further Results

**Table A.1:** Extended comparison with the external exact solver for the weak-case instance `Georgetown15`. For certified runs, the reference value is the exact optimum; otherwise it is the best upper bound reported at termination.

| Seed | $k$ | Status | Our runtime [s] | Exact runtime [s] | Our value | Reference value | Ratio | Match |
|---|---|---|---|---|---|---|---|---|
| 78 | 4.00 | certified | 0.44 | 15.85 | 420.00 | 16.00 | 26.25 | no |
| 399 | 4.00 | certified | 0.45 | 16.01 | 547.00 | 16.00 | 34.19 | no |
| 1109 | 4.00 | certified | 0.37 | 70.01 | 160.00 | 18.00 | 8.89 | no |
| 78 | 8.00 | time limit | 0.47 | 605.10 | 736.00 | 44.00 | 16.73 | no |
| 399 | 8.00 | time limit | 0.49 | 611.05 | 693.00 | 46.00 | 15.07 | no |
| 1109 | 8.00 | time limit | 0.39 | 618.15 | 637.00 | 48.00 | 13.27 | no |
| 78 | 16.00 | time limit | 0.53 | 620.20 | 1,417.00 | 90.00 | 15.74 | no |
| 399 | 16.00 | time limit | 0.57 | 616.95 | 1,664.00 | 92.00 | 18.09 | no |
| 1109 | 16.00 | time limit | 0.48 | 621.84 | 1,207.00 | 96.00 | 12.57 | no |

# Bibliography


[1] Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. Subcubic algorithms for gomory–hu tree in unweighted graphs. *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, 2020.

[2] Haris Angelidakis, Yury Makarychev, and Pasin Manurangsi. An improved integrality gap for the calinescu-karloff-rabani relaxation for multiway cut. 11 2016.

[3] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. 10th dimacs implementation challenge: Graph partitioning and graph clustering. `https://sites.cc.gatech.edu/dimacs10/`, 2012. Benchmark dataset collection; accessed 2026-03-26.

[4] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: Compression techniques. In *Proceedings of the 13th International Conference on World Wide Web*, pages 595–602. ACM, 2004.

[5] Yixin Cao, Jianer Chen, and Jia-Hao Fan. An $o^*(1.84s^k)$ parameterized algorithm for the multiterminal cut problem. *CoRR*, abs/1711.06397, 2017.

[6] Chandra Chekuri and Kent Quanrud. Isolating Cuts, (Bi-)Submodularity, and Faster Algorithms for Global Connectivity Problems, March 2021.

[7] Gruia C, Howard Karloff, and Yuval Rabani. An improved approximation algorithm for multiway cut. *Journal of Computer and System Sciences*, 60(3):564–574, 2000.

[8] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The Complexity of Multiterminal Cuts. *SIAM J. Comput.*, 23(4):864–894, August 1994.

[9] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, 2011.

[10] E. A. Dinitz. Algorithm for the solution of the max-flow problem with polynomial estimation. *Soviet Mathematics Doklady*, 11:1277–1280, 1970. English translation of Doklady Akademii Nauk SSSR 194(4), 1970.

[11] Yefim Dinitz. Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation. *Soviet Math. Dokl.*, 11:1277–1280, January 1970.

[12] Yefim Dinitz. Dinitz' algorithm: The original version and even's version. In *Theoretical Computer Science: Essays in Memory of Shimon Even*, volume 3895 of *Lecture Notes in Computer Science*, pages 218–240. Springer, 2006.

[13] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking Optimization Software with Performance Profiles, March 2004.

[14] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

[15] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. RAND Corporation, Santa Monica, CA, 1962.

[16] Andrew V. Goldberg and Robert Endre Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.

[17] `https://github.com/KaHIP/KaHIP`. KaHIP source code.

[18] Laboratory for Web Algorithmics (LAW). Law datasets. `https://law.di.unimi.it/datasets.php`, 2026. Dataset collection page; accessed 2026-03-26.

[19] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[20] Jason Li and Debmalya Panigrahi. Approximate gomory-hu tree is faster than $n - 1$ max-flows, 2021.

[21] Jason Li and Debmalya Panigrahi. Deterministic Min-cut in Poly-logarithmic Max-flows, May 2022.

[22] Ankur Moitra. Advanced algorithms: Lecture 13 – submodular functions, 2016. MIT 6.854 / 18.415 lecture notes.

[23] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[24] Dominik Schultes. *Route Planning in Road Networks*. PhD thesis, Universität Karlsruhe (TH), 2008.

[25] Amanda L. Traud, Peter J. Mucha, and Mason A. Porter. Social structure of facebook networks. *Physica A: Statistical Mechanics and its Applications*, 391(16):4165–4180, 2012.

[26] Mark Velednitsky and Dorit S. Hochbaum. Isolation Branching: A Branch and Bound Algorithm for the k-Terminal Cut Problem. In Donghyun Kim, R. N. Uma, and Alexander Zelikovsky, editors, *Combinatorial Optimization and Applications*, pages 624–639, Cham, 2018. Springer International Publishing.

[27] Mark Velednitsky and Dorit S. Hochbaum. Isolation branching: A branch and bound algorithm for the k-terminal cut problem. *J Comb Optim*, 44(3):1659–1679, October 2022.

[28] Chris Walshaw. The graph partitioning archive. `https://chriswalshaw.co.uk/partition/`, 2000. Benchmark archive for graph partitioning; accessed 2026-03-26.