Supernova: From Cycles to Stars for Efficient Connectivity Augmentation

Tarik M. A. Cavalcanti

15 September 2026

3683193

Bachelor Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

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

Co-supervisors:
Dr. Ernestine Großmann
Dr. Kenneth Langedal

Acknowledgments

I would like to express my deepest gratitude to my advisors, Dr Ernestine Großmann and Kenneth Langedal, for their invaluable guidance, support, and encouragement throughout this project. I thank them for introducing me to the field of algorithm engineering and for suggesting this interesting and challenging research topic for my thesis, and for the freedom in choosing my own research direction. Their thoughtful feedback and dedication have had a significant impact on my work.

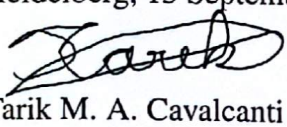
I am also very grateful to the Algorithm Engineering research group and Prof. Christian Schulz for giving me the opportunity to work on this project and for providing such a stimulating and supportive research environment.

My greatest thanks go to my family, whose love and support have accompanied me throughout my entire life journey. Without them, this would not have been possible.

Finally, I would like to thank all my friends and the wonderful people I have met during my time at Heidelberg University. Each of them, in their own way, has made this journey more enjoyable and memorable.

I hereby confirm, Tarik M. A. Cavalcanti, Matr. No. 3683193, 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than those specified, and I have identified all passages that have been taken verbatim or paraphrased from published or unpublished works, including those from digital or AI-based sources.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 thesis, 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, 15 September 2026



Tarik M. A. Cavalcanti

Abstract

Connectivity augmentation strengthens a network by adding links at minimum cost. In this thesis, we study the WEIGHTED CONNECTIVITY AUGMENTATION problem (WCA): given a graph and a set of candidate links with positive costs, the task is to find a minimum-cost subset whose addition increases the edge-connectivity of the graph by one. Even restricted variants of WCA are NP-complete. Considerable theoretical effort has led to increasingly strong approximation guarantees, culminating in a $(1.5 + \epsilon)$-approximation for the general problem. However, these algorithms are not practical and have repeatedly been shown to be outperformed by considerably simpler heuristics. We therefore study WCA from an algorithm-engineering perspective and develop efficient exact and heuristic algorithms for practical instances. Our approach formulates WCA as a WEIGHTED SET COVER problem over the minimum cuts of the input graph and systematically exploits the structure of the resulting set system. Based on this formulation, we develop compact data structures, problem-specific reduction rules, optimized greedy algorithms, and specialized exact and heuristic solvers. We combine these techniques in SUPERNOVA, a new algorithmic framework for connectivity augmentation. An extensive experimental evaluation on synthetic graphs shows that SUPERNOVA can outperform existing state-of-the-art approaches. Our exact algorithms solve substantially larger instances to optimality than the direct ILP baseline, while our heuristic algorithms offer improvements in solution quality or running time over existing practical methods. Our LAZY SUPERNOVA exact solver is approximately 1158 times faster than the unreduced direct ILP on cycles with 80 vertices, the largest instance size that the best existing algorithm can handle. The solver is also capable of solving instances with up to 2560 vertices under the same resource limits. On generated cacti, our data reductions accelerate the state-of-the-art exact solver by a factor of up to 166.6 and increase its largest fully completed size eightfold.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Many real-world systems can be modeled as graphs whose connectivity reflects their robustness against failures. For example, computer and power networks require alternative routes when individual connections fail. Adding connections, however, may incur nonuniform costs. The WEIGHTED CONNECTIVITY AUGMENTATION problem (WCA) therefore asks for a minimum-cost set of links whose addition increases the edge-connectivity of a graph by one.

The problem admits efficient algorithms in special cases, but is hard in general. If links are unweighted and every link is available, Naor, Gusfield, and Martel [13] compute an optimal augmentation by one in $\mathcal{O}(nm)$ time. In contrast, Eswaran and Tarjan [5] showed that weighted tree connectivity augmentation is NP-complete. Frederickson and JáJá [8] proved that this remains true when link costs belong to $\{1, 2\}$.

This motivates approximation algorithms and heuristics for the general weighted problem. Faraj, Großmann, Joos, Möller, and Schulz [6] engineered three heuristics and an exact integer linear program for WCA and provided the first implementations of two recent better-than-2 approximation algorithms, among other algorithms from the literature. We continue this line of research by implementing new heuristics and exact algorithms, as well as data reductions, which reduce the problem size without changing the optimal solution.

1.2 Our Contribution

This thesis builds on the preceding work by viewing WCA as a structured WEIGHTED SET COVER problem (WSC), where the minimum cuts form the universe and each link represents the set of minimum cuts it covers. Rather than materializing this potentially large WSC instance, we exploit the structure of the cactus representation of all minimum cuts. This yields specialized data structures, data reductions, and heuristic and exact solvers.

The highlights of our contributions are as follows:

- We give an explicit WSC reduction of WCA and develop explicit and WCA-specialized representations, designed for handling large cycles more efficiently.

- We introduce data reductions for diminishing the size of both the input graph and the set of links. These include three main algorithms for eliminating disposable links: PROJECTIN, PROJECTOUT and PROJECTCROSS. On real-cost cacti with 640 vertices, the complete reduction pipeline removes a median of 99.89% of candidate links and 75.47% of vertices. The reductions accelerate the direct ILP by a factor of up to 244.5 on cycles. On cacti, they make instances eight times larger optimally solvable for the ILP.

- We implement the greedy weight-coverage algorithm of Faraj, Großmann, Joos, Möller, and Schulz on our specialized WSC representations, augment their approach by introducing lazy score evaluation, and obtain its approximation guarantee from the standard analysis of GREEDY SET COVER. Our implementation is up to 35.85, 205.87, 3.13 and 14.08 times faster on stars, trees, cycles and cacti, respectively. It also lowers the median gap with respect to the optimal solution from 7.35% to 2.33% on stars and from 8.55% to 1.89% on trees, while yielding the same solution on cycles and cacti.

- We develop an algorithmic framework under the name SUPERNOVA, which is applicable to exact and heuristic solvers. This framework leverages knowledge about the structure of the instance to lazily choose a set of constraints to be iteratively satisfied.

- We use SUPERNOVA to construct exact and heuristic algorithms that outperform the previous state-of-the-art in terms of solution quality and running time on a selection of synthetic instances.

- LAZY SUPERNOVA provides exact solving without constructing every constraint in advance. It is up to 1158 times faster than the state-of-the-art ILP solver on cycles at $n = 80$ and 37.97 times faster on cacti. On cycles, it increases the largest optimally solved instance size from 80 to 2560 vertices, a factor of 32.

- GREEDY SUPERNOVA improves the speed and solution quality of the previous state-of-the-art heuristic solvers. It is up to 33.84 times faster than GWC on cycles and 24.92 times faster on cacti. On their shared reference instances, it also lowers the median gap with respect to the best-known solution from 7.66% to 2.64% on cycles and from 7.51% to 2.73% on cacti.

A detailed analysis of our experimental results and setup is given in Chapter 5.

1.3 Structure

Chapter 2 introduces the common graph-theoretic foundations and terminology used throughout the thesis. Chapter 3 reviews previous theoretical and practical work on connectivity augmentation. Chapter 4 presents the WSC formulation, data reductions, solvers, and SUPERNOVA framework. Chapter 5 details our experimental setup, results and discussion of our findings. Chapter 6 concludes the thesis with a summary of our contributions and presents potential pathways for future research.

CHAPTER 2

Fundamentals

This chapter introduces the general notation and definitions used throughout the thesis. Afterwards, we formulate the augmentation problem and introduce the minimum-cut cactus representation used by all of our algorithms.

2.1 General Definitions

An undirected simple graph $G = (V, E)$ consists of a finite vertex set V and an edge set $E \subseteq \binom{V}{2}$, where $\binom{V}{2}$ denotes the unordered pairs of distinct vertices. We write $n = |V|$ and $m = |E|$, and use $V(H)$ and $E(H)$ for the vertex and edge sets of any graph H. A *simple path* is a sequence of distinct vertices with an edge between each consecutive pair. A *cycle* closes such a path with an edge between its last and first vertices. Two paths are *edge-disjoint* if they share no edge; they are also said to be *independent*. A graph is *connected* if there exists a path joining any pair of vertices. A *tree* is a connected, acyclic graph. A *connected component* is a maximal connected subgraph. An *articulation vertex* is a vertex whose removal increases the number of connected components.

To *contract* a nonempty vertex set $X \subseteq V$ means to replace all vertices in X with a single vertex X. Any edge endpoint in X is replaced by X, and any edge with both endpoints in X is removed. We retain resulting parallel edges with their original weights. To *contract an edge* $e = \{u, v\}$ means to contract its endpoint set $\{u, v\}$.

A *cut* $K = \{A, B\}$ is an unordered bipartition of V into two nonempty sets, called its *shores*: $A \cap B = \emptyset$ and $A \cup B = V$. Its cut edges are defined as $\delta(K) = \{\{u, v\} \in E : u \in A,\ v \in B\}$, and an edge set $F \subseteq E$ *induces* K if $F = \delta(K)$. The cut has size $|\delta(K)|$. When edges have positive *weights* or *capacities* $w : E \to \mathbb{R}_{>0}$, then K has *weight* $w(K) \equiv \sum_{e \in \delta(K)} w(e)$. A *minimum cut* is a cut of minimum weight. We denote the family of all minimum cuts by $\mathcal{K}(G)$ and their common value by $\lambda(G)$, omitting the graph argument when it is clear. For disconnected graphs, we extend this notation by setting $\lambda = 0$. An unweighted graph is *k-edge-connected* if $\lambda(G) \geq k$. Equivalently, every pair of

vertices is joined by at least k edge-disjoint paths. For weighted graphs, we use the same term to mean that every cut has weight at least k, and for the path-based definition, we interpret the weights as parallel edges.

The graph families used in the thesis include paths, cycles, stars, trees, and cacti. A *path graph* on n vertices has $V = \{v_1, \ldots, v_n\}$ and $E = \{\{v_i, v_{i+1}\} : 1 \leq i < n\}$. A *cycle graph* additionally contains $\{v_n, v_1\}$, with $n \geq 3$. Cycles are crucial to our discussion, so we define $Cyc(G)$ as the set of all cycle graphs in some graph G. A *star graph* has a central vertex connected by an edge to each of the remaining $n - 1$ vertices. A *cactus graph* is a connected graph in which any two distinct cycles share no edges.

2.2 Weighted Connectivity Augmentation

An instance (G, L, c) of Weighted Connectivity Augmentation (WCA) consists of an unweighted graph $G = (V, E)$ with $k = \lambda(G)$, a set of available links $L \subseteq \binom{V}{2} \setminus E$, and positive link costs $c : L \to \mathbb{R}_{>0}$. For $L' \subseteq L$, let $c(L') = \sum_{l \in L'} c(l)$ and $G + L' = (V, E \cup L')$. The objective is to minimize $c(L')$ subject to $\lambda(G + L') \geq k + 1$. The instance is infeasible if no such link set exists. One can also consider the generalization to $L \subseteq \binom{V}{2}$, and interpret the links $L \cap E$ as additional parallel edges.

The following equivalence is central to our approach. Only the minimum cuts contribute to $\lambda(G)$, so it suffices to increase the weight of each minimum cut by at least one. We say a link l *covers* a cut K if its endpoints lie on opposite shores, and we define the *coverage* of l as $cov(l) \equiv \{K \in \mathcal{K}(G) : l \text{ covers } K\}$. Therefore, link set L' is a feasible solution if and only if $\bigcup_{l \in L'} cov(l) = \mathcal{K}(G)$.

Disconnected graphs are trivial to augment, so we focus on connected input graphs. To augment such a graph, contract every connected component and retain, for each pair of components, only the cheapest link between them. Links within one component cannot help increase the connectivity, while parallel component links are interchangeable. If the resulting component link graph is connected, a minimum spanning tree gives an optimal augmentation. Otherwise, the instance is infeasible.

Any graph admits a representation of its minimum cuts as a cactus with $\mathcal{O}(n)$ vertices and edges [12, 9]. We use a normalized cactus representation, in which the edges are unweighted, but parallel edges are allowed. In particular, a cycle of length 2 may occur as two parallel edges. Occasionally we shall omit the parallel edges in the figures in the interest of space and simplicity. Every minimum cut of a nontrivial normalized cactus has weight 2. They are induced by pairs of edges on the same cycle.

The *minimum-cut cactus representation* of G can be formalized as follows. It consists of a cactus $C = C(G) = (V_C, E_C)$ and a map $\pi : V \to V_C$. Each cactus vertex x represents the bag $\pi^{-1}(\{x\}) \equiv \{v \in V : \pi(v) = x\}$ of original vertices, which no minimum cut separates. For $S \subseteq V_C$, let $\pi^{-1}(S) = \{v \in V : \pi(v) \in S\}$. Every minimum cut $\{S, V_C \setminus S\}$ of C corresponds to the minimum cut $\{\pi^{-1}(S), \pi^{-1}(V_C \setminus S)\}$ of G. Furthermore, every minimum cut of G is represented by some minimum cut of C. Thus the representation

preserves the minimum-cut bipartitions.

Figure 2.1 gives an example of a minimum-cut cactus representation. The four vertices a, b, c, d form a complete subgraph and cannot be separated by a minimum cut, so they map to one cactus vertex x.

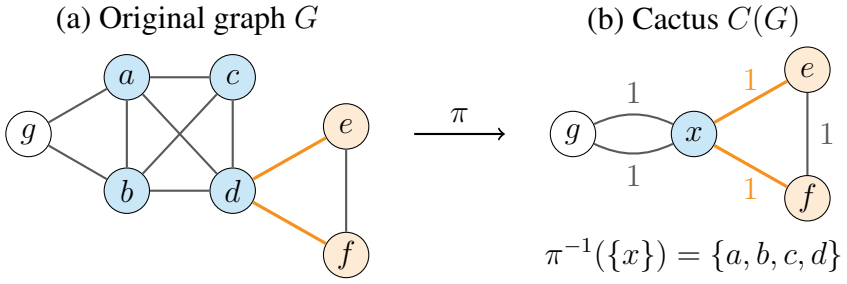


Figure 2.1: A graph G with $\lambda(G) = 2$ and its normalized minimum-cut cactus $C(G)$. The blue vertices map to x, with $\pi^{-1}(\{x\}) = \{a, b, c, d\}$; the remaining vertices keep their labels. All original edges have unit capacity, and cactus-edge capacities are shown explicitly. The orange edges induce the cut with shore $\{e, f\}$ in both graphs. The two original edges incident to g become one tree edge of capacity 2, interpreted as an implicit 2-cycle.

A link $l = \{a, b\}$ in the original graph projects to $\pi(l) = \{\pi(a), \pi(b)\}$. If several links have the same projected endpoints, they cover the same minimum cuts, so we retain a cheapest one and remember its original representative. Self-loops in the cactus are ignored, as they cover no minimum cuts. The cut correspondence then implies that a link set covers all minimum cuts of G exactly when its projection covers all minimum cuts of $C(G)$. Consequently, the cactus representation is a valid reduction of the original problem. Formally, the reduced instance is (C, L_c, c'), where $L_c = \{\pi(l) : l \in L\}$ and $c' : L_c \to \mathbb{R}_{>0}$ is defined by $c'(l) = \min_{l' \in \pi^{-1}(l)} c(l')$. Henceforth, (C, L, c) denotes this reduced instance, and we write $C = (V, E)$ when the original graph is no longer needed.

We delegate construction of the cactus representation to the VieCut library [9], so the augmentation algorithms start from the reduced instance. The construction of Nagamochi, Nakao, and Ibaraki [12] runs in $\mathcal{O}(nm + n^2 \log n + \gamma m \log n)$ time, where here $n = |V(G)|$ and $m = |E(G)|$ refer to the original graph and γ is the number of cycles in the resulting cactus. For the implementation and its reductions, we refer to Henzinger, Noe, Schulz, and Strash [9].

2.3 Block-Tree and Link Projections

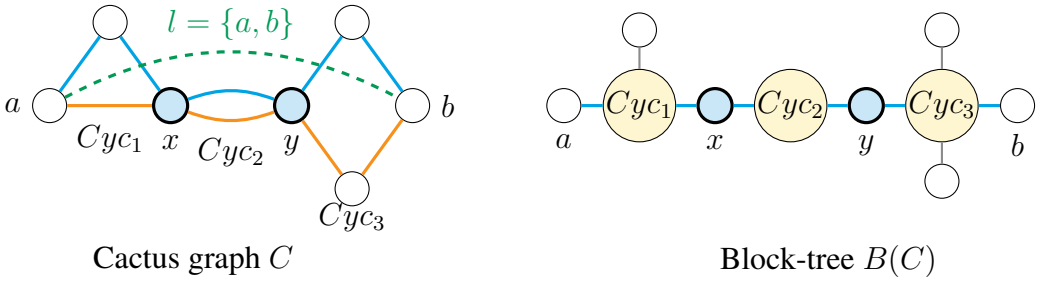
In this section we present some definitions that allow us to describe the coverage of a link. For any link l, two edge-disjoint paths between its endpoints choose complementary arcs in each traversed cycle and meet at articulation vertices. Every cut in $cov(l)$ is induced by

two edges, one edge from each path and in the same cycle. The projection path records this common structure and is most conveniently defined through the block-tree. The block-tree, in a sense, removes the two paths and collapses them into a single one, so that a path in the block-tree contains exactly the articulation vertices and the cycles traversed.

Definition 1
Let $C = (V, E)$ be a cactus graph. We define its block-tree *as the graph $B(C) = (V', E')$ where*

- $V' = V \cup Cyc(C)$,
- $E' = \{\{v, Cyc_i\} : Cyc_i \in Cyc(C),\ v \in V(Cyc_i)\}$.

Thus every cycle is replaced by a star. This preserves connectivity and removes every cycle, so $B(C)$ is a tree. In particular, there is a unique path in the block-tree between any two original vertices. This path tells us exactly which cycles a link traverses and at what vertices it intersects them. Figure 2.2 illustrates the construction and the block-tree path of a link. Now we can proceed to define the projection path of a link.



Cactus graph C Block-tree $B(C)$

Figure 2.2: A cactus graph and its block-tree. All cactus edges have unit weight; the two parallel edges between x and y form the 2-cycle Cyc_2. Each cycle Cyc_i becomes a yellow block vertex in $B(C)$. The blue and orange a-b-paths choose different arcs in each cycle, but both contain x and y. More generally, every a-b-path projects onto the highlighted path in $B(C)$. Removing the yellow block vertices from this unique path yields $V(P_l) = \{a, x, y, b\}$, whose internal vertices are therefore traversed by every a-b-path in C.

Definition 2
Let $C = (V, E)$ be a cactus graph and let $l = \{a, b\} \in \binom{V}{2}$ be a link. Let $(q_0, \ldots, q_k)$ be the unique a-b-path in $B(C)$ and let $(p_0, \ldots, p_r)$, with $p_0 = a$ and $p_r = b$, be its subsequence of vertices in V. The projection path *of l is the path graph*

$$P_l \equiv \big(\{p_0, \ldots, p_r\}, \{\{p_{i-1}, p_i\} : 1 \le i \le r\}\big).$$

Its edges are projection edges *and need not belong to E.*

The projection path is named as such because it determines how a link projects onto each cycle. Additionally, the vertices of the projection path are precisely those vertices at which the two edge-disjoint paths between the link endpoints meet. Therefore, those vertices are common to all simple paths between the link endpoints. Intuitively, the block-tree removes the choice between the two arcs of a cycle. Its unique path therefore retains exactly the articulation vertices that no path between the link endpoints can avoid.

This definition also allows us to define the projection of a link onto a cycle: namely, the vertices of that cycle that the projection path goes through. The links induced by these projections onto cycles will jointly cover exactly the minimum cuts that the original link covers, so they act, in a sense, as building blocks of the link.

Definition 3
Let $C = (V, E)$ be a cactus graph and let $l = \{a, b\} \in \binom{V}{2}$ be a link. The projection of l onto a cycle $Cyc_i \in Cyc(C)$ *is*

$$\pi_l(Cyc_i) := \begin{cases} V(P_l) \cap V(Cyc_i), & \text{if } Cyc_i \text{ lies on the } a\text{-}b \text{ path in } B(C), \\ \emptyset, & \text{otherwise.} \end{cases}$$

CHAPTER 3

Related Work

Connectivity augmentation is a classical problem in combinatorial optimization, with a rich history of theoretical results. The simplest case arises when the links have uniform cost and the link set is complete, i.e. $L = \binom{V}{2}$. In this setting, the problem can be solved in polynomial time. Let $(l_1, \ldots, l_m)$ denote the cactus leaves in the order in which they occur along some Hamiltonian cycle. If m is odd, we append l_1 to this sequence, forming an even-length sequence $(l_1, \ldots, l_{2k})$. An optimal solution is found by pairing opposite leaves in this ordering $S = \{\{l_i, l_{i+k}\} : 1 \leq i \leq k\}$.

However, WCA is a challenging problem in general. Eswaran and Tarjan [5] showed that weighted bridge-connectivity augmentation, that is increasing $\lambda(G)$ from 1 to 2, which can be reduced to the WEIGHTED TREE AUGMENTATION (WTA) problem by contracting all cycles, is NP-complete. Frederickson and JáJá [8] established hardness even when link costs belong to $\{1, 2\}$.

Despite that, an approximation factor of 2 can be obtained through several classical techniques. Frederickson and JáJá [8] gave a 2-approximation for WTA by reducing it to a minimum-cost arborescence problem. Khuller and Thurimella [10] subsequently simplified and accelerated this approach.

Stronger guarantees can be obtained for the unweighted version. Byrka, Grandoni, and Jabal Ameli [2] broke the factor-2 barrier for general connectivity augmentation with a $(2 \ln 4 - 967/1120 + \epsilon)$-approximation, approximately 1.91, using a reduction to STEINER TREE and a specialized analysis of the resulting instances. Cecchetto, Traub, and Zenklusen [3] subsequently obtained a 1.393-approximation for both tree and general connectivity augmentation. A recent preprint by Kortsarz [11] claims a $4/3$-approximation for unweighted tree augmentation; this claim concerns TAP, rather than general WCA with arbitrary link costs.

The factor-2 barrier can also be crossed for weighted trees, first under restrictions on the cost range and then for arbitrary costs. Adjiashvili [1] obtained the first such improvement for bounded link costs, and Fiorini, Groß, Könemann, and Sanità [7] strengthened the guarantee to $(1.5 + \epsilon)$. Traub and Zenklusen [16] removed the bounded-cost assumption

with a relative-greedy $(1+\ln 2+\epsilon)$-approximation: a structured initial solution is improved by replacing parts of it with cheaper components found by dynamic programming. Their local-search approach [16] achieved $(1.5+\epsilon)$ for WTA through non-oblivious local search, which measures progress with a potential function.

These improvement strategies can be extended from trees to general connectivity augmentation by exploiting the structure of the solutions of a simplified version of the problem. Traub and Zenklusen [15] reduce WCA to ring augmentation and use planar solutions of a directed version to find replacements by undirected links. Their framework yields both a relative-greedy $(1 + \ln 2 + \epsilon)$-approximation and a stronger local-search $(1.5 + \epsilon)$-approximation. These guarantees apply to arbitrary initial edge-connectivity and arbitrary positive link costs, with polynomial running time for each fixed $\epsilon > 0$. These are the strongest approximation guarantees for general WCA to date.

The practical value of these guarantees must also be assessed through implementations and experiments. Early experimental work by Watanabe, Mashima, and Taoka [17] introduced FSA, FSM, SMC, and HBD and evaluated their theoretical and empirical behavior. Faraj, Großmann, Joos, Möller, and Schulz [6] provided the first implementations of the two better-than-2 approximation algorithms for general WCA and compared them with a 2-approximation, earlier heuristics, and their own solvers. Their contributions include GWC, MSTCONNECT, a path-exchange local search, and an exact ILP with efficient minimum-cut enumeration and a heuristic initial solution. On the small instances used to evaluate the dynamic-programming-based approximations, both were orders of magnitude slower than the exact ILP, while their heuristic methods handled substantially larger instances [6].

This thesis builds on these experimental findings. The focus is on compact coverage data structures, data reductions, and heuristic and exact solvers that avoid materializing as many minimum-cut coverage constraints as possible.

CHAPTER 4

Algorithms

This chapter presents the algorithms designed and implemented in this work. We first formulate WCA as WEIGHTED SET COVER (WSC) and describe four data structures for the resulting instance. We then exploit the cactus structure to obtain data reductions. Finally, we present the implemented solvers and the SUPERNOVA algorithmic framework.

4.1 Reduction to the WEIGHTED SET COVER Problem

It is helpful both from a theoretical and practical perspective to view the WCA problem as a WSC problem. In this section, we first formally define the WSC problem and describe its relation to WCA, then we present the concrete data structures we have implemented to represent it.

4.1.1 Overview

An instance of WSC is a tuple $(\mathcal{U}, \mathcal{S}, c)$, where $\mathcal{U}$ is a finite set, called the universe, $\mathcal{S} \subseteq 2^{\mathcal{U}}$ is a collection of subsets of $\mathcal{U}$, and $c : \mathcal{S} \to \mathbb{R}_{>0}$ assigns a cost to each set. The objective is to find $S' \subseteq \mathcal{S}$ minimizing $c(S') \equiv \sum_{s \in S'} c(s)$ subject to $\bigcup_{s \in S'} s = \mathcal{U}$.

The reduction from WCA to WSC is straightforward. We identify the set of minimum cuts with the universe and the coverage of each link with a set. More precisely, for a WCA instance (C, L, c), the resulting WSC instance is $(\mathcal{K}, \{\mathit{cov}(l) : l \in L\}, c)$, where the costs are transferred from l to $\mathit{cov}(l)$ in the natural way. In this way, WCA and this WSC instance have the same feasible solutions and optimum value. We therefore use links and sets, as well as minimum cuts and elements, interchangeably, depending on whether the general set cover or the connectivity augmentation problem is more relevant.

A WSC instance is naturally represented by its bipartite graph between sets and elements, where we include an edge between a set and an element if the set contains it. Based on this idea, we implement four representations of WSC: Compressed Sparse Row (CSR),

Adjacency Matrix (AM), Partial Adjacency Matrix (PAM), and the WCA-specific Cycle Set Cover (CycSC).

As described in Section 2.3, the minimum cuts covered by a link are completely determined by its projections onto the cycles traversed by its projection path. For each such cycle, the covered cuts are precisely those induced by choosing one edge from each of the two arcs between the projected endpoints. Consequently, the cycles traversed by a link, and hence its covered cuts, can be enumerated by traversing its projection path in the block-tree and processing its projection onto each encountered cycle. Specifically, we iterate over all the edge pairs on opposite arcs for every encountered cycle. We use this structure to construct the WSC representations described below.

4.1.2 Data Structures and Implementation

The simplest data structures we tested are the CSR and AM representations. They assign indices to links and minimum cuts and directly describe the coverage relationship between them. CSR construction takes $\Theta(\sum_{s\in\mathcal{S}} |s|)$ time and space, whereas an AM has time and space complexity of $\Theta(|\mathcal{U}| \cdot |\mathcal{S}|)$ because it initializes a matrix with this many entries. Therefore, CSR is preferable when the resulting WSC instance is sparse.

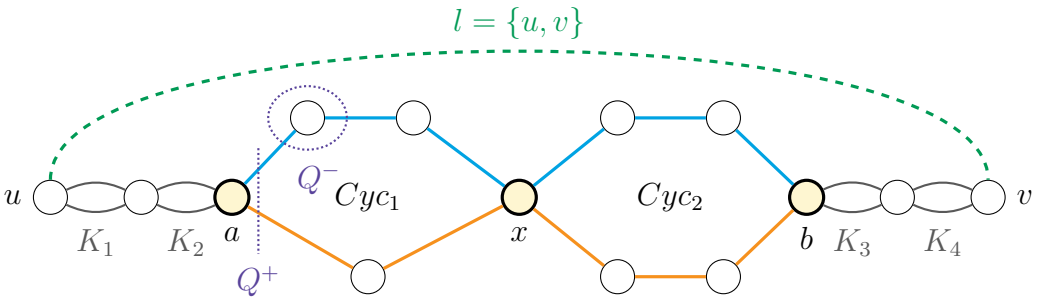
A cycle with all vertex pairs available as links shows that the explicit WSC representation can have $\Theta(n^4)$ entries. There are $\binom{n}{2}$ links and $\binom{n}{2}$ minimum cuts. Each link covers a number of minimum cuts equal to the product of the lengths of the two arcs separating its endpoints. Consequently,

$$\begin{aligned}\sum_{s\in\mathcal{S}} |s| &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (j-i)\cdot(n-(j-i)) \\ &= \sum_{d=1}^{n-1} d(n-d)^2 \\ &= \frac{n^2(n^2-1)}{12} = \Theta(n^4).\end{aligned}$$

The potentially high memory requirements of explicit WSC representations motivated the development of PAM and CycSC. The PAM representation reduces this memory demand by storing vertex-to-cut relationships instead of explicit link-to-cut incidences. It chooses a shore for every minimum cut and stores, in a bit-packed matrix, whether each vertex belongs to the selected shore. Then, for any link $\{u, v\}$, the exclusive or (XOR) operation between the rows corresponding to u and v retrieves the explicit cover relationship for that link. Its memory use is $\Theta(|V||\mathcal{U}|)$ bits: among n-vertex cacti, this is $\Theta(n^2)$ for trees and $\Theta(n^3)$ for a single cycle. It is independent of the number of links, at the cost of one row-wise XOR per queried link.

Finally, we designed CycSC, a representation tailored to WCA. It stores the cuts belonging to cycles of length 2 in either the CSR or PAM representation. For the remaining

cycles, it stores only the link projections onto them, and enumerates the covered cuts on demand. The cycle cuts covered by a projection can be enumerated with a nested loop over the two arcs separating the projected endpoints. Thus, many coverage relationships are not explicitly materialized. A conceptual view of all data structures is illustrated in Figure 4.1.



CSR: cut indices per link

$l \mapsto [K_1, K_2, K_3, K_4, Q^+, \ldots]$

Store 19 covered cut indices.

AM: one coverage row per link

	K_1	K_2	K_3	K_4	Q^+	Q^-	$\cdots$
l	1	1	1	1	1	0	$\cdots$

29 entries: 19 ones and 10 zeros.

PAM: vertex–cut matrix

	K_1	K_2	K_3	K_4	Q^+	Q^-	$\cdots$
u	0	1	0	1	0	1	$\cdots$
v	1	0	1	0	1	1	$\cdots$

XOR the two endpoint rows.

CycSC: cuts and projections

$K_1, \ldots, K_4$: use CSR or PAM.

$\pi_l(Cyc_1) = \{a, x\}$,
$\pi_l(Cyc_2) = \{x, b\}$.

Enumerate 15 cuts on demand.

Figure 4.1: The same link coverage in four WSC representations. All cactus edges have unit weight; $K_1, \ldots, K_4$ are the cuts of the four 2-cycles. Gold vertices mark the projected endpoints on Cyc_1 and Cyc_2. Each covered cut on these cycles pairs a blue edge with an orange edge. The purple dotted boundaries mark the edge pairs inducing Q^+ and Q^-. Q^+ separates u from v and is covered by l; Q^- isolates the upper-left vertex of Cyc_1 and is not covered. PAM stores membership in one chosen shore per cut; XORing the endpoint rows gives the AM row regardless of this choice.

4.2 Data Reductions

In this section, we present our data reduction algorithms for the WCA problem. There are three types of data reductions:

- **Link Domination.** If a link l_- satisfies $cov(l_-) \subseteq \bigcup_{l' \in L'} cov(l')$ and $c(l_-) \geq \sum_{l' \in L'} c(l')$, for some $L' \subseteq L \setminus \{l_-\}$, then l_- is said to be dominated by the links in L'. It can then be removed from the link set L, as any solution Sol with $l_- \in Sol$ can be replaced without increasing its cost with the substitution $Sol \leftarrow (Sol \setminus \{l_-\}) \cup L'$.
- **Minimum-Cut Domination.** If for two minimum cuts K_+, K_- every link covering K_+ also covers K_-, then K_- can safely be removed. Indeed, every feasible solution must cover K_+ and therefore also covers K_-.
- **Forced-Link Reduction.** If a minimum cut is covered by a unique link, then that link belongs to every feasible solution. We add it to the fixed solution, remove all cuts it covers by contracting edges along its projection path, and remove it from the link set.

These data reductions are direct analogues to the data reductions found in the WSC problem. However, in the set cover case, finding set dominations is a set cover instance in itself. On the other hand, we can exploit the cactus structure of WCA to find link dominations more efficiently. The rules and algorithms discussed below are safe but not exhaustive. That is, there may be other dominations that are not detected by our algorithms.

Within link dominations, we distinguish three structural patterns: direct dominations, path dominations, and cycle dominations. To detect these patterns, we develop the algorithms PROJECTIN, PROJECTOUT, and PROJECTCROSS, respectively. We first introduce each domination pattern together with its corresponding algorithm. We then briefly describe the minimum-cut domination and forced-link reductions. Since these two reductions are straightforward adaptations of standard set cover reductions, we omit their implementation details.

To facilitate our discussion, we introduce the following definition, which describes the links a set of links dominates, up to their costs.

Definition 4
Let $C = (V, E)$ be a cactus graph and $L \subset \binom{V}{2}$ a collection of links. Define $\mathrm{Drop}_{C,L}(L')$ as the set of links $\{l \in L : cov(l) \subseteq \bigcup_{l' \in L'} cov(l')\}$. We say a link $l \in \mathrm{Drop}_{C,L}(L')$ is dropped *by L' or, alternatively, that L'* drops *l.*

From now on, whenever C and L are clear from the context, we will omit them and simply write $\mathrm{Drop}(L')$. In this sense, $\mathrm{Drop}(L')$ describes the set of links that add no benefit to our solution, given that we have already added the links in L' to it. In particular, if $l \in \mathrm{Drop}(L')$ and $c(l) \geq \sum_{l' \in L'} c(l')$, then l is dominated by L'. The following statement formalizes what was already established:

Lemma 1
Let C be a cactus graph, $L \subseteq \binom{V}{2}$ and $L' \subseteq L$. If $l \in \mathrm{Drop}(L')$, then $\mathrm{Drop}(L' \cup \{l\}) = \mathrm{Drop}(L')$.

Proof. Clearly, $\mathrm{Drop}(L') \subseteq \mathrm{Drop}(L' \cup \{l\})$. Now, let $l_1 \in \mathrm{Drop}(L' \cup \{l\})$. Then $cov(l_1) \subseteq cov(l) \cup \bigcup_{l' \in L'} cov(l')$. Since $l \in \mathrm{Drop}(L')$, we have that $cov(l) \subseteq \bigcup_{l' \in L'} cov(l')$. Therefore, $cov(l_1) \subseteq \bigcup_{l' \in L'} cov(l')$, which implies that $l_1 \in \mathrm{Drop}(L')$. □

4.2.1 PROJECTIN: Direct Link Dominations

We first consider *direct* dominations: ones where a single link dominates another.

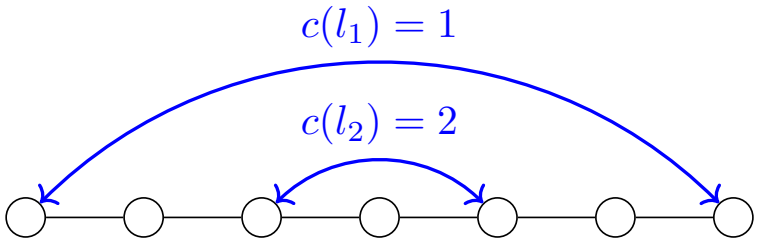


Figure 4.2: Direct domination: the endpoints of l_2 lie on the projection path of l_1, and $c(l_1) < c(l_2)$. Thus l_2 can be replaced by l_1.

The projection path characterizes one-to-one domination. For a link l_+, every link l_- with endpoints in its projection path, i.e. $l_- \subseteq V(P_{l_+})$, satisfies $cov(l_-) \subseteq cov(l_+)$. In fact, these are also precisely all links satisfying this property. Figure 4.2 illustrates this containment, with costs for which l_1 dominates l_2. To understand this, recall that the vertices of P_{l_+} are traversed by every path between the endpoints of l_+. So, any minimum cut separating two such vertices must separate the endpoints of l_+. We formally assert this statement in the following lemma.

Lemma 2
Let $C = (V, E)$ be a cactus graph and $l_+, l_- \in \binom{V}{2}$ be two links. Then $cov(l_-) \subseteq cov(l_+)$ if and only if $l_- \subseteq V(P_{l_+})$.

Proof. Let $l_+ = \{a_+, b_+\}$ and $l_- = \{a_-, b_-\}$ be two links such that $l_- \subseteq V(P_{l_+})$. Now let $K = \{A, B\} \in cov(l_-)$ be some minimum cut covered by l_-. Without loss of generality, $a_- \in A$ and $b_- \in B$. Suppose, for contradiction, that l_+ does not cover K. Then a_+ and b_+ lie on the same side of the cut, say in A. But since $a_-, b_- \in V(P_{l_+})$, all paths from a_+ to b_+ go through a_- and b_-. Hence, a_- and b_- must also lie in A, which is a contradiction.

For the converse, assume that $cov(l_-) \subseteq cov(l_+)$. We show that the endpoints of l_- belong to $V(P_{l_+})$. For the sake of contradiction, assume $a_- \notin V(P_{l_+})$. Let P_1 and P_2 be two independent paths from a_- to b_-. If a_- lies in neither path, then any two edges incident to a_- induce a minimum cut covered by l_-, but not by l_+, therefore we may assume that

a_- lies in exactly one of the paths. If it lay on both, then $a_- \in V(P_{l_+})$. Without loss of generality, assume that $a_- \in V(P_1)$. Let K be the minimum cut induced by choosing the edges from P_1 that are incident to a_-. That is a minimum cut because two consecutive edges on such a path lie on the same cycle, except if $a_- \in V(P_{l_+})$. In that case, P_2 connects a_+ to b_+ and does not contain a_-. Clearly, all vertices in this path, in particular a_+ and b_+, belong to the same side of K. So K is not covered by l_+, which is a contradiction. A visualization of this case is shown in Figure 4.3. □

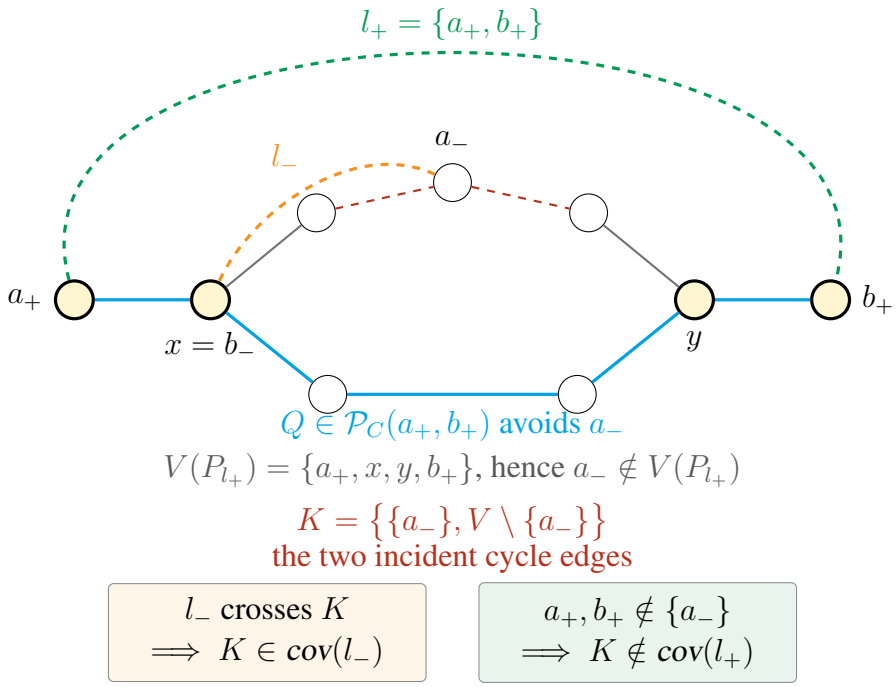


Figure 4.3: The case used in the converse direction of Lemma 2. If an endpoint a_- of l_- does not belong to $V(P_{l_+})$, an a_+-b_+-path Q avoids it. The two cycle edges incident to a_- induce a minimum cut K that is crossed by l_-. Both endpoints of l_+ remain on the other side, so l_+ does not cover K, contradicting $cov(l_-) \subseteq cov(l_+)$.

Hence, a link drops exactly the links whose endpoints lie on its projection path. We exploit this special property to find all direct dominations with our algorithm ProjectIn. Throughout the three projection routines, we define for every $a, b \in V$ a value $D(a, b)$ as a *cover bound*. For a link l with endpoints a and b, we also write $D(l)$ meaning $D(a, b)$. It records the currently known minimum cost of covering all minimum cuts of $cov(\{a, b\})$. By default, $D(a, b) \leftarrow c(\{a, b\})$ if $\{a, b\} \in L$ and $D(a, b) \leftarrow \infty$ otherwise. The updated values found by a link domination algorithm are propagated to the following ones, so that they jointly refine the cover bounds.

All link domination algorithms operate on projection rules. Let $D^*(u, v)$ denote the optimum cost of covering $cov(\{u, v\})$. For a link set $L' \subseteq L$ and a link $l \in$

$\mathrm{Drop}(L')$, $D^*(l) \leq \min\{D(l), \sum_{l' \in L'} D(l')\}$. This induces the projection rule $D(l) \leftarrow \min\{D(l), \sum_{l' \in L'} D(l')\}$. This operation is referred to as *projecting the cost of L' onto l*. Our goal is to get as close as possible to D^* by applying as few projection rules as possible. Additionally, we maintain a boolean flag $R(l)$ called the *replacement flag*. It is initially false and is set to true if during a projection operation the cost of l is equal to that of L'. That is necessary so that ultimately links do not trivially dominate themselves.

PROJECTIN operates on the principle that links with longer projection paths can project their costs onto links with immediately smaller projection paths. These, in turn, project their costs further. Therefore, PROJECTIN iterates over all pairs of vertices in descending order with respect to the size of their projection path. For every pair $\{u, v\}$, with ordered projection path $P_{\{u,v\}} = (u, x, \ldots, y, v)$, $\{u, v\}$ projects its cost onto the two links $\{u, y\}$ and $\{x, v\}$. Algorithm 1 gives more details for clarity. The name PROJECTIN reflects that a link's cost is projected into the links inside its projection path.

Algorithm 1: PROJECTIN

Input: Cactus graph $C = (V, E)$, link set L, link costs c
Output: Replacement costs D, replacement flags R
foreach $\{a, b\} \in \binom{V}{2}$ **do**
  $D(a, b) \leftarrow c(\{a, b\})$ if $\{a, b\} \in L$, and ∞ otherwise
  $R(a, b) \leftarrow$ false
$h \leftarrow \max_{\{a,b\} \in \binom{V}{2}} |P_{\{a,b\}}|$
for $k \leftarrow h$ **downto** 3 **do**
  foreach $\{u, v\} \in \binom{V}{2}$ *with* $|P_{\{u,v\}}| = k$ **do**
    $(p_0 = u, p_1, \ldots, p_{k-1} = v) \leftarrow P_{\{u,v\}}$
    foreach $\{r, s\} \in \big\{\{p_0, p_{k-2}\}, \{p_1, p_{k-1}\}\big\}$ **do**
      if $D(u, v) \leq D(r, s)$ **then**
        $D(r, s) \leftarrow D(u, v)$
        $R(r, s) \leftarrow$ true
return D, R

Correctness. Whenever the link $\{u, v\}$ is processed, the algorithm maintains the invariant that $D(u, v)$ is the minimum cost of covering $\mathit{cov}(\{u, v\})$ by a single link. The invariant holds for a maximal projection path because its only initial candidate is its corresponding link, if that link exists. For the induction step, consider $P_{\{u,v\}} = (p_0 = u, p_1, \ldots, p_{k-1} = v)$. Every strict superpath has already been processed because pairs are considered in descending order of projection-path size. In a tree, every contained path is reached by repeatedly deleting the first or last vertex of its containing path. The same holds after removing the cycle vertices from block-tree paths. A projection path may have many immediate

superpaths, since it may be extended at either endpoint in several ways. However, every one of these superpaths was already processed. Consequently, all candidates from all strict superpaths have reached $D(u, v)$ before $\{u, v\}$ is processed. The algorithm then forwards this minimum to the two immediate subpaths of $P_{\{u,v\}}$, preserving the invariant for the next layer.

Running time. Bucketing all vertex pairs by projection-path size and traversing their paths takes $\mathcal{O}(n^3)$ time in the current implementation. For each of the $\mathcal{O}(n^2)$ pairs, we traverse the corresponding projection path to find its length and immediate subpaths. Since projection paths have size $\mathcal{O}(n)$, this takes $\mathcal{O}(n^3)$ time in total. A path graph attains this bound. Once the buckets and paths are available, the two propagation updates per pair take $\mathcal{O}(n^2)$ time. The table and buckets use $\mathcal{O}(n^2)$ space.

4.2.2 PROJECTOUT: Path Dominations

In the previous section, we have described how to find one-to-one link dominations. However, in the case where the links are weighted, it is also possible that a combination of cheaper links can dominate a more expensive one.

In this section, we handle dominations arising from links belonging to P_l, where l is the link to be dominated, and defer interactions that occur inside cycles to the following section. More precisely, we strengthen the domination requirement for a link set L' from $\mathit{cov}(l) \subseteq \bigcup_{l' \in L'} \mathit{cov}(l')$ to *path-based dominations* $E(P_l) \subseteq \bigcup_{l' \in L'} E(P_{l'})$. We would like to find the cheapest set of links whose projection edges cover all the projection edges of l. This is clearly a sufficient, yet not exhaustive, requirement for the domination of l. We present a brief proof of this statement in the following lemma, and an illustration in Figure 4.4.

Lemma 3
Let $C = (V, E)$ be a cactus graph and $l \in \binom{V}{2}$ be a link, and $L' \subseteq \binom{V}{2}$ be a set of links such that $E(P_l) \subseteq \bigcup_{l' \in L'} E(P_{l'})$. Then $l \in \mathrm{Drop}(L')$.

Proof. Let the ordered vertices in $V(P_l)$ be $(p_0, p_1, \ldots, p_k)$, where p_0 and p_k are the endpoints of l. By assumption, for every $i \in \{1, \ldots, k\}$, there exists a link $l_i \in L'$ such that $\{p_{i-1}, p_i \in E(P_{l_i})\}$. Both endpoints therefore belong to $V(P_{l_i})$, so by Lemma 2 $\{p_{i-1}, p_i\} \in \mathrm{Drop}(L')$. Now consider a minimum cut covered by l. Since it separates p_0 from p_k, it must separate some pair p_{i-1}, p_i. So it is covered by $\{p_{i-1}, p_i\}$, and by extension by l_i. Thus, $l \in \mathit{Drop}(L')$. □

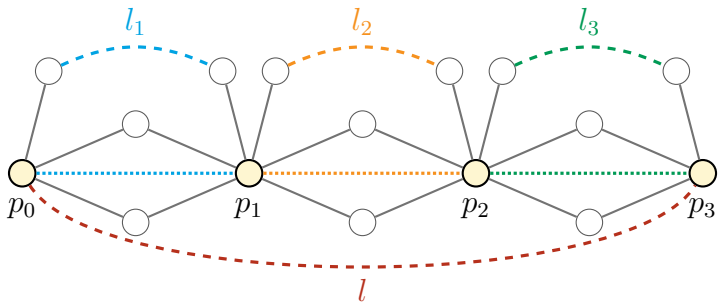
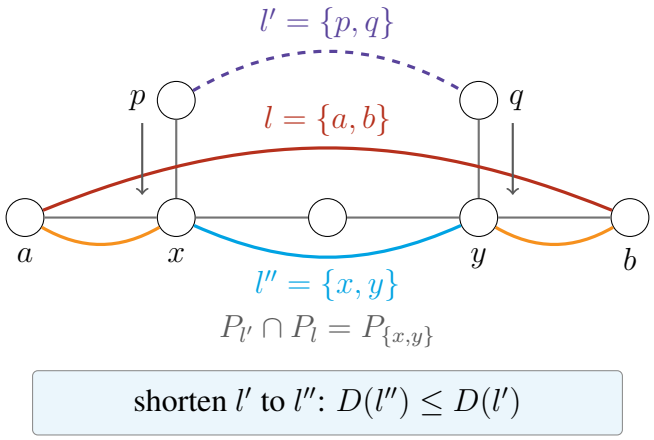
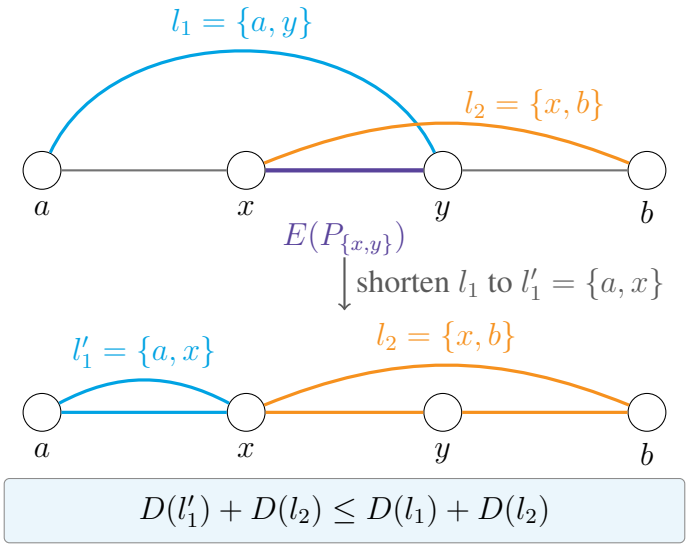


Figure 4.4: Path dropping with $L' = \{l_1, l_2, l_3\}$. Each P_{l_i} contains the dotted projection edge $\{p_{i-1}, p_i\}$ of l, so together they cover $E(P_l)$ and $l \in \mathrm{Drop}(L')$.

Our task is to find all path dominations, and update D correspondingly. Assuming that PROJECTIN has already been executed, we can simplify this task using the following observations. Let l be a link and L' a set of links that path-dominates l. After PROJECTIN, every pair of links l_1 and l_2 such that $V(P_{l_1}) \subseteq V(P_{l_2})$ satisfies $D(l_1) \leq D(l_2)$. Therefore, any link in L' can be replaced by a link whose endpoints lie in its projection path without increasing the total cover bound of L', provided that the resulting set still path-dominates l. When we apply such a substitution to a link $l' \in L'$, we say that we *shorten* l'. Repeated shortening allows us to restrict our search space to link sets in which every link has both endpoints in $V(P_l)$ and every pair of distinct links $l_1, l_2 \in L'$ satisfies $E(P_{l_1}) \cap E(P_{l_2}) = \emptyset$. Figure 4.5 illustrates these two simplifications.



(a) Shorten l' to l'' so both endpoints lie on P_l. The orange links remain unchanged, preserving coverage of $E(P_l)$.



(b) Shorten l_1 to l_1' so the projection paths are edge-disjoint. The unchanged link l_2 covers the removed overlap.

Figure 4.5: Shortening links after PROJECTIN, with $l = \{a, b\}$. In each example, replacing a link by the indicated shorter link preserves coverage of $E(P_l)$ without increasing the total cover bound of L'. The shortened links are evaluated using their cover bounds D.

Our final step is to derive a recurrence relation that leads to a dynamic program. Let $D^*(a, b)$ be the minimum total cover bound of a set of links covering $E(P_{\{a,b\}})$, using the bounds D obtained after ProjectIn. By our preceding observation, we may assume that their projection paths partition $P_{\{a,b\}}$ into consecutive subpaths. Consider an optimal such partition. If it consists of a single link, its cover bound is $D(a, b)$. Otherwise, its last link is $\{x, b\}$ for some internal vertex x of $P_{\{a,b\}}$. The remaining links form an optimal partition of the subpath from a to x because if a cheaper partition existed, we could use it together with $\{x, b\}$ to improve the entire partition. Hence, the total cover bound is $D^*(a, x) + D(x, b)$. Conversely, for any internal vertex x, an optimal partition from a to x can be extended by $\{x, b\}$ to cover the entire path. Since we do not know where the last link begins, we consider every internal vertex and obtain

$$D^*(a, b) = \min \left\{ D(a, b), \min_{x \in V(P_{\{a,b\}}) \setminus \{a,b\}} \big(D^*(a, x) + D(x, b)\big) \right\}.$$

The values for D^* on the right-hand side of the recurrence pertain to projection paths that are strictly shorter than $P_{\{a,b\}}$. Since paths with one edge are trivially solvable, we can compute D^* for all pairs of vertices in increasing order of projection path size. We call this dynamic program ProjectOut, since it projects the cost of covering smaller paths into larger ones. A description of ProjectOut is given in Algorithm 2 and an image depicting the recurrence is given in Figure 4.6.

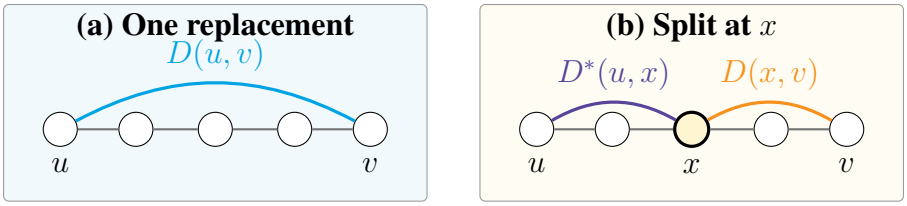


Figure 4.6: The alternatives in the ProjectOut recurrence. A projection path is covered either by one replacement with cover bound $D(u, v)$ or by an optimal partition from u to an internal vertex x, followed by a final link with cover bound $D(x, v)$.

Running time. There are $\binom{n}{2} = \mathcal{O}(n^2)$ vertex pairs. For each pair $\{u, v\}$, ProjectOut tests every internal vertex of $P_{\{u,v\}}$, of which there are at most $n - 2$. The running time is therefore $\mathcal{O}(n^3)$, and this bound is tight for a path, where the sum of all projection-path lengths is $\Theta(n^3)$. The tables D, D^*, and R require $\Theta(n^2)$ space.

Algorithm 2: PROJECTOUT

Input: Cactus graph C, cover bounds D and replacement flags R after PROJECTIN
Output: Refined cover bounds D, updated flags R

```
h ← max_{{a,b}∈C(V,2)} |V(P_{a,b})|
for k ← 3 to h do
    foreach {a, b} ∈ C(V,2) with |V(P_{a,b})| = k do
        foreach x ∈ V(P_{a,b}) \ {a, b} do
            w ← D(a, x) + D(x, b)
            if w ≤ D(a, b) then
                D(a, b) ← w
                R(a, b) ← true
return D, R
```

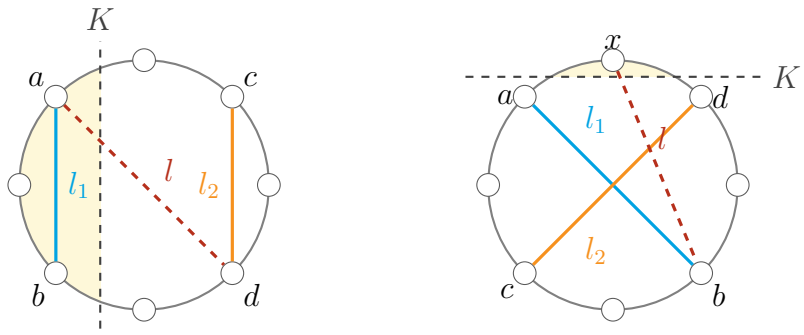
4.2.3 PROJECTCROSS: Cycle Dominations

The approach we have described so far is sufficient to find all link dominations if all cycles are of length 2. Otherwise, there is an additional way links can dominate each other. For the remainder of this section, we focus on a single such cycle. We then look for link dominations arising purely from links inside it, which we call *cycle dominations*. For that purpose, we derive a characterization of $\mathrm{Drop}(L')$, that shall act as a guiding principle for finding such dominations.

We begin by studying the set $\mathrm{Drop}(L')$ when L' contains exactly two links l_1 and l_2. Consider a cycle graph Cyc drawn as a circle in the plane. Imagine the links l_1 and l_2 drawn as chords of the circle. If the chords intersect, then the links are said to *cross*. If they instead share an endpoint, they are said to *touch*. Otherwise, they are said to be *disjoint*. All interesting interactions between l_1 and l_2 occur when they cross or touch.

These interactions are more clearly demonstrated with a visualization. Figure 4.7 shows this intuition. Assume that l_1 and l_2 are disjoint and consider a candidate link l to the set $\mathrm{Drop}(L')$. Then we can visualize a cut that separates the endpoints of l, but keeps the endpoints of l_1 and l_2 in the same partition. For that not to be possible, the links l_1 and l_2 must clearly cross over. Simultaneously, they must also touch both endpoints of l, otherwise one may isolate that endpoint by a cut that separates it from the rest of the cycle, which will keep all endpoints of both l_1 and l_2 in the same partition. This is more easily expressed with a visualization. The following lemma formalizes this idea in a more technical way.

Lemma 4
Let $l_1 = \{v_a, v_b\}$ and $l_2 = \{v_c, v_d\}$ be two links on a cycle Cyc. If l_1 and l_2 intersect or touch, then $\mathrm{Drop}(\{l_1, l_2\})$ contains exactly the links whose endpoints are in $\{v_a, v_b, v_c, v_d\}$. In case they are disjoint, then the drop set is trivial, i.e., $\mathrm{Drop}(\{l_1, l_2\}) = \{l_1, l_2\}$.



(a) Disjoint links (b) An untouched endpoint

$$K \in \mathit{cov}(l), \qquad K \notin \mathit{cov}(l_1) \cup \mathit{cov}(l_2) \qquad \Longrightarrow \qquad l \notin \mathrm{Drop}(\{l_1, l_2\})$$

Figure 4.7: Cuts demonstrating $l \notin \mathrm{Drop}(\{l_1, l_2\})$ on a cycle. The dashed black line crosses the two cycle edges defining the minimum cut K. (a) The links l_1 and l_2 neither cross nor share an endpoint. The cut then separates l_1 from l_2 and is covered by the candidate $l = \{a, d\}$, but by neither l_1 nor l_2. (b) The links l_1 and l_2 cross, but do not touch an endpoint of l. So the cut isolating x is covered only by l.

Proof. For the sake of formality, note that $L' \subseteq \mathrm{Drop}(L')$. Assume some cyclic order of the vertices and edges of Cyc. Let $v_1, ..., v_n$ be the vertices of Cyc in this order, and let $e_i = \{v_i, v_{i+1}\}$ for $i = 1, ..., n-1$ and $e_n = \{v_n, v_1\}$. First, assume l_1 and l_2 are disjoint. Without loss of generality, $v_a < v_b < v_c < v_d$. Consider a link $l = \{v_x, v_y\}$ such that $v_x \notin l_1 \cup l_2$. Then the minimum cut induced by e_x, e_{x-1} is covered by l, but not by l_1 or l_2, so $l \notin \mathrm{Drop}(\{l_1, l_2\})$. If both endpoints are shared with either l_1 or l_2, then l covers the minimum cut induced by e_b, e_d. Hence, $l \notin \mathrm{Drop}(\{l_1, l_2\})$.

Now let l_1 and l_2 touch or cross. Without loss of generality, $v_a < v_c \leq v_b < v_d$, up to some cyclic permutation of Cyc. If l touches neither l_1 nor l_2, then again the cut induced by e_x, e_{x-1} is covered by l but not by l_1 or l_2. Now suppose l touches both with l_1 and l_2. Assume for contradiction that there is a cut $K = \{A, B\}$ that is covered by l but covered by neither l_1 nor l_2. Then without loss of generality, $v_a, v_b \in A$ and $v_c, v_d \in B$. If $v_a, v_b \in A$, this means that one of the arcs separating them is contained in A. That means that either v_c or v_d is also in A, which is a contradiction. Therefore, $l \in \mathrm{Drop}(\{l_1, l_2\})$. □

We can use this lemma together with the following general observation to construct $\mathrm{Drop}(L')$ for an arbitrary set L' of links. A set of links $L' = \{l_1, \ldots, l_k\}$ forming a path such that $l_i = \{a_i, b_i\}$ and $b_i = a_{i+1}$ for $i \in \{1, \ldots, k-1\}$ acts as a large link connecting a_1 and b_k. So clearly $\{a_0, b_k\} \in \mathrm{Drop}(L')$. We state this result in the following two lemmas.

Lemma 5
Let $G = (V, E)$ be a graph. For any three vertices $x, y, z \in V$, $\mathit{cov}(\{x, z\}) \subseteq \mathit{cov}(\{x, y\}) \cup \mathit{cov}(\{y, z\})$.

Proof. Let $K = \{A, B\} \in \mathit{cov}(\{x, z\})$. Without loss of generality, $x \in A$ and $z \in B$. If $y \in A$, then $K \in \mathit{cov}(\{y, z\})$, while if $y \in B$, then $K \in \mathit{cov}(\{x, y\})$. □

Lemma 6
Let $G = (V, E)$ be a graph. For any set of vertices $x_1, \ldots, x_n \in V$, $\mathit{cov}(\{x_1, x_n\}) \subseteq \bigcup_{i=1,\ldots,n-1} \mathit{cov}(\{x_i, x_{i+1}\})$.

Proof. We prove the statement by induction on n. By Lemma 5 it is true for $n = 3$. Assume it is true for some k. $\mathit{cov}(\{x_1, x_{k+1}\}) \subseteq \mathit{cov}(\{x_1, x_k\}) \cup \mathit{cov}(\{x_k, x_{k+1}\})$. By the induction step, $\mathit{cov}(\{x_1, x_k\}) \subseteq \bigcup_{i=1,\ldots,k-1} \mathit{cov}(\{x_i, x_{i+1}\})$. Combining both statements finishes the proof. □

Together, Lemma 4 and Lemma 5 give us a set of simple domination rules. We design our PROJECTCROSS algorithm based on them. Denote $D^*(l)$ as the optimal cover of a link l up to cycle dominations. Then the following relations hold:

1. Crossing Rule: Let v_0, v_1, v_2, v_3 be vertices in some cycle in cyclic order. Then $D^*(v_i, v_{i+1 \mod 4}) \leq D(v_0, v_2) + D(v_1, v_3)$.

2. Touching or Triangle Rule: Let v_0, v_1, v_2 be vertices in some cycle. Then $D^*(v_0, v_2) \leq D(v_0, v_1) + D(v_1, v_2)$.

We apply these rules until no further improvements to D can be made. We shall later see that this approach is sufficient to find $D*$.

To achieve this, we make a simple observation. The cheapest link $l = \arg\min_{l'} D(l')$ already satisfies $D(l) = D^*(l)$, since both rules only improve D by combining two links. Furthermore, any link can only be improved by combinations of strictly cheaper links. Thus, a Dijkstra-like approach comes to mind. We initialize a priority queue with all links keyed by their initial value of D. Then, at every iteration, we pop a link l and hold the invariant that all rules combining links cheaper than l have been exhaustively applied. Hence, $D(l) = D^*(l)$. To maintain this invariant for the next link, we keep the set of all links which have already been processed S, and combine l with every link in S, applying the appropriate rule. Algorithm 3 describes this procedure.

For any single cycle with n vertices, we need to process $\binom{n}{2}$ links. At iteration i, there are $i - 1$ links in the set of processed links. If implemented naively, this procedure would then require $\sum_{i=1}^{\binom{n}{2}}(i-1) = \Theta(n^4)$ time. However, when processing l, we do not need to consider every link in S. We only need those which share an endpoint or cross with l and whose cover bound is less than or equal to the threshold $\max_{l'} D(l') - D(l)$. At the start, S has few links, but as more links are processed $D(l)$ increases, so that the threshold will sink. Hence, if S is able to efficiently iterate over only the relevant links, we may expect a better running time.

Exhaustively applying the triangle and cross rules is sufficient to compute D^* for every link in a cycle. To prove this, we proceed to characterize $\mathrm{Drop}(L')$ for an arbitrary set of

Algorithm 3: PROJECTCROSS

Input: Cycle graph $Cyc = (V, E)$, replacement costs D, replacement flags R
Output: Updated costs D and flags R

```
Q ← a min-priority queue containing every l ∈ (V choose 2) with key D(l)
S ← ∅
while Q ≠ ∅ do
    l ← Q.popMin()
    foreach l′ ∈ S do
        if l and l′ share an endpoint then
            T ← {(l ∪ l′) \ (l ∩ l′)}
        else
            if l and l′ cross in Cyc then
                (v0, v1, v2, v3) ← the endpoints of l ∪ l′ in cyclic order
                T ← {{v_i, v_{i+1 mod 4}} : i = 0, ..., 3}
            else
                T ← ∅
        foreach t ∈ T do
            w ← D(l) + D(l′)
            if w ≤ D(t) then
                R(t) ← true
                if w < D(t) then
                    D(t) ← w
                    Q.updateKey(t, w)
    S ← S ∪ {l}
return D, R
```

links. The idea is that when two links cross or touch, they not only add all the links made from their endpoints to $\mathrm{Drop}(L')$, but this also chains through to all other links connected by touching or crossing. In the following definition, we introduce the *link intersection graph*, which serves to formalize this concept.

Definition 5
Let $Cyc = (V, E)$ be a cycle graph and $L \subset \binom{V}{2}$ a collection of links. Then the link intersection graph *$H_{Cyc}(L)$ of L on Cyc is a graph where $V(H(L)) = L$ and $\{l_1, l_2\} \in E(H)$ if and only if either l_1 and l_2 intersect when drawn as straight lines on the planar embedding of C or they share an endpoint. To reduce clutter, we shall simply write $H(L)$ if the underlying cycle is clear from context. An example visualization of this graph can be seen in Figure 4.8.*

(a) Links on the cycle **(b) Link intersection graph**

Figure 4.8: A cycle with seven links and its link intersection graph. Two vertices in the right panel are adjacent exactly when the corresponding links in the left panel cross or share an endpoint. Matching colors identify the same link in both panels.

Now, based on this definition, L' will induce a set of connected components in $H(L)$. The links dropped by L' are exactly those whose endpoints are touched by the same induced connected component. The intuition behind this is simple. Assume that we have a set of links L'. For every pair of links in L' that touch or cross, by Lemma 4 and Lemma 5, the clique formed by their endpoints is in $\mathrm{Drop}(L')$. By Lemma 1 we may add the links in this clique to L' and $Drop(L')$ will remain unchanged. But now, using these cliques, we can find a path from every endpoint of some link to the endpoint of every other link, as long as there is a path of touching or crossing links connecting them. Therefore, by Lemma 6, all these vertices are merged into a larger clique in $\mathrm{Drop}(L')$. We formalize this intuition in the following theorem. A depiction of this result can be seen in Figure 4.9.

Theorem 7
Let $Cyc = (V, E)$ be a cycle graph, $L \subseteq \binom{V}{2}$ a collection of links, and $L' \subseteq L$. Let Q be a connected component of the subgraph of the link intersection graph $H(L)[L']$, and let $V_Q := \bigcup_{l \in Q} l$ be the set of endpoints of its links. Then, for every $x, y \in V_Q$, $\{x, y\} \in \mathrm{Drop}(L')$.

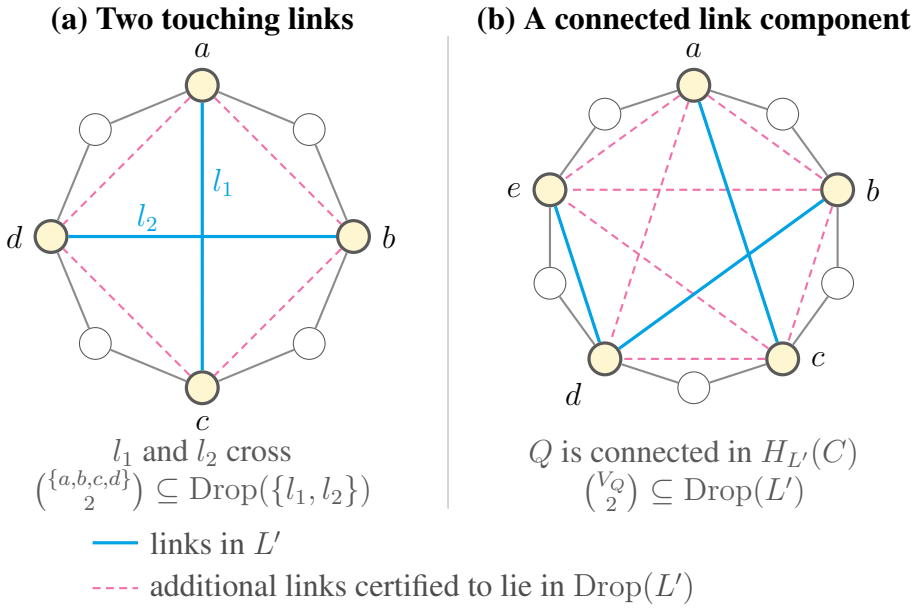


Figure 4.9: The two-link lemma and its connected-component generalization. The solid blue chords are the links in L'. Dashed magenta chords show additional available links in $\mathrm{Drop}(L')$ on the same cycle. In the left panel, the nontrivial touching case is represented by two crossing links, which generate every pair of their four endpoints. In the right panel, the three blue links form a connected component Q in the link intersection graph and generate every pair of vertices in V_Q.

Proof. We prove the statement by induction on the number of links in Q.

For $|Q| = 1$, the statement is trivial.

Assume the statement holds for all connected components of size k, for some $k \geq 1$. Let Q be a connected component of size $k+1$. We can choose two links $l_1, l_2 \in Q$ such that Q is connected if they are removed. By the induction hypothesis, for every $x, y \in V_{Q \setminus \{l_1\}}$ and $z, w \in V_{Q \setminus \{l_2\}}$, $\{x, y\}, \{z, w\} \in \mathrm{Drop}(L')$. Adding all such edges to L' does not change $\mathrm{Drop}(L')$ by Lemma 1, so we may consider that they belong to L'. Now, clearly, all vertices in V_Q are connected by links in L'. Therefore, by Lemma 6, for all $x, y \in V_Q$, $\{x, y\} \in \mathrm{Drop}(L')$. □

Using this result, we can now prove that PROJECTCROSS finds all dominations inside a cycle. To prove that, we need to show that for every link set L', every link $l \in \mathrm{Drop}(L')$ satisfies $D(l) \leq \sum_{l' \in L'} D(l')$. This is simple to prove.

Theorem 8
Let $Cyc = (V, E)$ be a cycle graph and $L = \binom{V}{2}$ the complete set of links. Let $D : \binom{V}{2} \to \mathbb{R}$ be a function satisfying $D(l) \leq D(l_1) + D(l_2)$ for every pair of links l_1, l_2 with $\{l_1, l_2\} \in E(H(L))$. Then for any $L' \subseteq L$, for every $l \in \mathrm{Drop}(L')$, $D(l) \leq \sum_{l' \in L'} D(l')$.

Proof. For every l, choose L' such that $l \in \mathrm{Drop}(L')$ that minimizes $\sum_{l' \in L'} D(l')$. If this L' satisfies our condition, then every other set of links containing l in its drop set will also satisfy it. Now L' is obviously a path in $H(L)$, as we may remove all links outside the path connecting two links incident to the endpoints of l, which will only reduce the sum and keep l in $\mathrm{Drop}(L')$. Let the path be $l_1, l_2, \ldots, l_k$. Every link $l' \subset l_1 \cup l_2$ satisfies $D(l') \leq D(l_1) + D(l_2)$. Every link $l'' \subset l_2 \cup l_3$ satisfies $D(l'') \leq D(l_2) + D(l_3)$. So every link $l''' \subset l_1 \cup l_2 \cup l_3$ satisfies $D(l''') \leq D(l_1) + D(l_2) + D(l_3)$. Continuing this way ends the proof. □

4.2.4 Forced-Link Reductions and Minimum-Cut Dominations

As explained in the introduction to the data reductions, there are two important reduction rules besides link dominations: forced-link reductions and minimum-cut dominations. If a minimum cut is covered by only a single link, every feasible solution must contain that link, regardless of its cost. We therefore add it to the fixed solution and simplify the cactus by contracting the parts whose minimum cuts are now covered. Minimum-cut dominations remove redundant coverage requirements. If every remaining link covering a minimum cut K_+ also covers another minimum cut K_-, then covering K_+ automatically covers K_-. We can therefore discard the requirement to cover K_- separately while retaining K_+. Figure 4.10 illustrates both reductions and the resulting cactus contractions.

These two reductions are implemented directly in the set cover setting. Therefore, unlike link dominations, they do not exploit the cactus structure. They are simple implementations of standard set cover reductions. To find minimum-cut dominations, for every element in

the WSC reduced instance, we iterate over all sets covering it and then calculate their intersection. To find forced-link reductions we simply verify if any element is covered by a single set. To make this more explicit we provide the pseudocode for both reductions in Algorithm 4. The algorithm records fixed sets in F and maintains the remaining requirements in $\mathcal{U}'$. The residual instance restricts each set outside F to $\mathcal{U}'$, retaining its identity and cost.

Algorithm 4: Forced-Link Reductions and Minimum-Cut Dominations

Input: WSC instance $(\mathcal{U}, \mathcal{S}, c)$
Output: Remaining requirements $\mathcal{U}'$, fixed sets F, or infeasible
$\mathcal{U}' \leftarrow \mathcal{U}$, $F \leftarrow \emptyset$
foreach $e \in \mathcal{U}$ **do**
  if $e \in \mathcal{U}'$ **then**
    $\mathcal{S}_e \leftarrow \{s \in \mathcal{S} : e \in s\}$
    if $\mathcal{S}_e = \emptyset$ **then**
      return infeasible
    else if $|\mathcal{S}_e| = 1$ **then**
      Let s be the unique set in $\mathcal{S}_e$
      $F \leftarrow F \cup \{s\}$
      $\mathcal{U}' \leftarrow \mathcal{U}' \setminus s$
    else
      $I \leftarrow \mathcal{U}'$
      foreach $s \in \mathcal{S}_e$ **do**
        $I \leftarrow I \cap s$
      `// Retain e; covering it also covers every other element of I.`
      $\mathcal{U}' \leftarrow \mathcal{U}' \setminus (I \setminus \{e\})$
return $\mathcal{U}', F$

Reductions of one type can trigger further reductions of another. For example, removing a dominated link may leave a minimum cut with only one covering link. We therefore apply the reductions in successive rounds until no further change occurs.

4.3 Solvers

In this section, we describe and analyze our algorithms for solving the WCA problem.

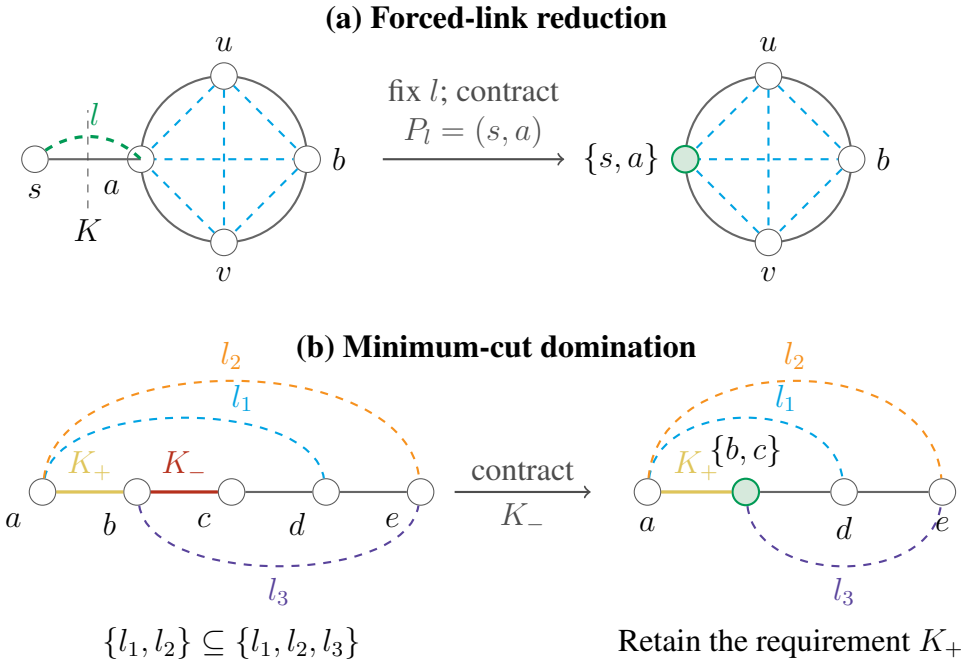


Figure 4.10: Forced-link reduction and minimum-cut domination. Solid edges belong to the cactus; dashed edges are candidate links. Tree edges have weight 2 and represent implicit 2-cycles. (a) All vertex pairs on the cycle are available as links, but only $l = \{s, a\}$ covers the cut K. Fixing l contracts its projection path, merging s and a. (b) The first and second path edges represent K_+ and K_-, respectively. Links l_1, l_2 cover both cuts, while l_3 covers only K_-. Thus K_- is redundant, and its edge can be contracted by merging b and c.

4.3.1 GREEDY SET COVER

Our first algorithm is an adaptation of the greedy weight-coverage algorithm (GWC) provided by Faraj, Großmann, Joos, Möller, and Schulz [6], but framed in the set cover context.

Adapting GWC to the WSC setting is straightforward. Indeed, the set cover reduction is a conceptual framework that allows us to view the augmentation problem in a more general framework. The fact that sets in $\mathcal{S}$ represent links and elements in $\mathcal{U}$ represent minimum cuts is abstracted away.

The algorithm is a simple greedy approach that acts as follows. Let A denote the elements covered by the current solution. Then we define the score of a set s as the number of uncovered elements it contains: $\text{score}(s) = |s \setminus A|$. At every step, a set is chosen that maximizes the score-cost ratio $\frac{\text{score}(s)}{c(s)}$. Naively implemented, GREEDY SET COVER must compute the score for every set at every iteration, then add the best set to the solution. Since there can be up to $\mathcal{O}(n^2)$ sets, the score computation can be expensive. Algorithm 5 gives this basic procedure.

Algorithm 5: GREEDY SET COVER

Input: $\mathcal{U}, \mathcal{S}, c$
Output: $Sol \subseteq \mathcal{S}$
procedure GREEDY SET COVER($\mathcal{U}, \mathcal{S}, c$)
$Sol \leftarrow \emptyset$
$A \leftarrow \emptyset$
while $A \neq \mathcal{U}$ **do**
  $s \leftarrow \arg\max_{s \in \mathcal{S}} \left\{ \frac{|s \setminus A|}{c(s)} \right\}$
  $Sol \leftarrow Sol \cup \{s\}$
  $A \leftarrow A \cup s$
return Sol

To avoid as many score computations as possible, we refine the algorithm with lazy score evaluation. It is based on the principle that the score of a set can only decrease at every iteration. Therefore, we store the sets in a priority queue keyed by their score-cost ratios. At every iteration, we pop the set with the best ratio and recompute it. In case it has not changed, it is guaranteed to still be the best set, so we do not need to compute further scores. Otherwise, we decrease its key in the priority queue and proceed to recomputing the score of the next top-ranked set.

Runtime. For a feasible instance, every selected set covers at least one new element and no set is selected twice. Thus, the number q of iterations is at most $|\mathcal{U}|$. With an explicit set representation, recomputing all scores takes $\mathcal{O}(\sum_{s \in \mathcal{S}} |s|)$ time per iteration, as we must iterate over every element in every set. Lazy evaluation may still require recomputing every

remaining score in an iteration, and each priority-queue update takes logarithmic time. This gives the worst-case bound

$$\mathcal{O}\left(q\left(\sum_{s\in\mathcal{S}}|s|+|\mathcal{S}|\log(1+|\mathcal{S}|)\right)\right).$$

The benefit of lazy evaluation is that it avoids many of these recomputations in practice.

Solution quality. The GREEDY SET COVER algorithm has already been described in the literature [4] and tight theoretical bounds have been found [14]. Noticing the connection between WCA and WSC allows us to apply these results directly to our problem. The simplest and most well-known bound is that the solution is $\leq H(\Delta)\mathrm{OPT}$ [4], where Δ is the size of the largest set in $\mathcal{S}$ and H is the harmonic function $H(n)=\sum_{i=1}^{n}\frac{1}{i}$. In our case, Δ is maximized by a link bisecting a cycle. If the cycle has n vertices, then that link covers $\Theta(n^2)$ minimum cuts.

Implementation

Clearly, there are two crucial operations that must be supported by our WSC representation to implement GREEDY SET COVER efficiently: adding a new set to the solution and computing the number of newly covered elements for a given set. The latter is especially important, as it may be called many times per iteration. In particular, we have identified two issues that may arise in the presence of large cycles:

1. The number of elements in $\mathcal{U}$ and the number of elements covered by each set are large.
2. For links crossing some cycle, there will be a large overlap in the elements they cover. This means that GWC will have to recalculate the scores of many sets and lazy score evaluation will be ineffective.

To address the first issue, we have developed a new approach to calculate the scores of each set. It is the inspiration for our hybrid WSC representation introduced in Section 4.1. We now describe how it operates in the context of GREEDY SET COVER.

For each cycle Cyc in the cactus, we partition its edges into classes A_i. Two distinct edges belong to the same class precisely when the minimum cut they induce is not covered by any link in the current solution. For this algorithm, provide each cycle with an orientation and enumerate its vertices and edges accordingly, i.e. $V(Cyc)=\{v_1,v_2,\ldots,v_{|V(Cyc)|}\}$ and $E(Cyc)=\{e_1,e_2,\ldots,e_{|E(Cyc)|}\}$. That allows us to interpret the links as intervals on a line. For a link l with projection $\pi_l(Cyc)=\{v_i,v_{i+k}\}$, let $I(l)=\{e_i,\ldots,e_{i+k-1}\}$ be the edge set connecting v_i and v_{i+k} along the chosen orientation. If $\pi_l(Cyc)=\emptyset$, let $I(l)=\emptyset$. Consequently, two edges belong to the same class when they have identical interval memberships for every selected link. In other words, two edges e_i and e_j belong to the same

class if and only if $e_i \in I(l') \Leftrightarrow e_j \in I(l')$ for every $l' \in Sol$, where Sol is the current solution. These classes therefore encode precisely the uncovered cuts. Each pair of edges in the same class will keep both endpoints of every selected link on the same side of the cut they induce.

This approach facilitates the computation of the score of a candidate link l. Each newly covered cut within a class A_i pairs an edge in $A_i \cap I(l)$ with an edge in $A_i \setminus I(l)$. The contribution of this cycle to the score is therefore

$$\text{score}_{Cyc}(l) = \sum_i |A_i \cap I(l)| \cdot |A_i \setminus I(l)|.$$

Classes of size one contribute no cuts and can be ignored. We therefore keep track of both the intervals and their class identifiers, so that their contributions can be combined without enumerating the individual cuts.

This data structure also permits efficient updates when a new set is added to the solution. Initially, all edges are in the same class A_0, as any pair of edges is a minimum cut. When a link l is added to the solution, each class properly intersecting $I(l)$ is split, resulting in a new class $A'_i = I(l) \cap A_i$, and is updated so that $A_i \leftarrow A_i \setminus I(l)$. This separates exactly the edge pairs whose cuts are newly covered by l and preserves the partition property. Figure 4.11 visualizes the score calculation in panel (a) and the class splitting procedure in panel (b).

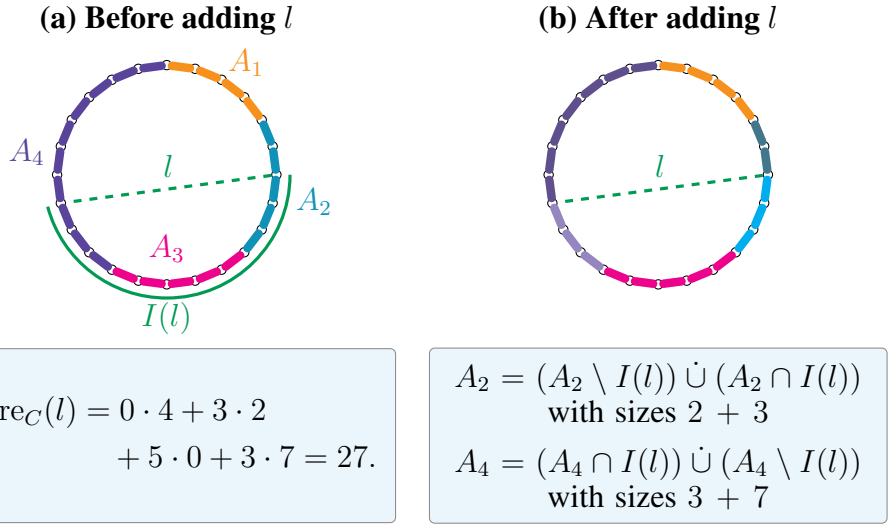


Figure 4.11: Score calculation and class splitting for a candidate link l on one cycle C. (a) The edge classes have sizes $|A_1| = 4$, $|A_2| = 5$, $|A_3| = 5$, and $|A_4| = 10$. The interval $I(l)$ contains respectively 0, 3, 5, and 3 edges from these classes, so this cycle contributes $0 \cdot 4 + 3 \cdot 2 + 5 \cdot 0 + 3 \cdot 7 = 27$ newly covered cuts to the score of l. (b) Adding l splits A_2 into classes of sizes 2 and 3, and A_4 into classes of sizes 3 and 7. Classes wholly outside or inside $I(l)$ remain unchanged.

In summary, our GREEDY SET COVER algorithm implementation is analogous to GWC by Faraj, Großmann, Joos, Möller, and Schulz [6], with some key implementation differences. Our implementation allows us to compute scores while avoiding contractions on the cactus graph whenever a link is added to the solution. Furthermore, lazy score evaluation potentially avoids many score computations.

4.3.2 GREEDY CHEAPEST

In their MSTCONNECT algorithm, Faraj, Großmann, Joos, Möller, and Schulz [6] seek to calculate a feasible solution quickly and then remove the excess. It starts from a minimum spanning forest of the link graph (V, L), which is a spanning tree when this graph is connected. Such a forest can be computed with Kruskal's algorithm.For a feasible WCA instance, every minimum cut is crossed by a candidate link. The forest path joining that link's endpoints must also cross the cut, so the forest is a feasible augmentation.

Kruskal's algorithm chooses edges in nondecreasing order of cost while avoiding the creation of cycles. We therefore construct an analogous set cover heuristic that considers sets in nondecreasing cost order. A set is only added to the solution if it covers a new element. Algorithm 6 gives the procedure.

Our requirement that added sets cover new elements is stronger than Kruskal's requirement that a link should not create a cycle. A set that covers no new minimum cuts may not create a cycle in the link graph. On the other hand, if a link covers a new minimum cut, it is guaranteed to not create a cycle in the link graph. Therefore our algorithm has less excess to remove at the end than MSTCONNECT.

Algorithm 6: GREEDY CHEAPEST

Input: $\mathcal{U}, \mathcal{S}, c$
Output: $Sol \subseteq \mathcal{S}$
procedure GREEDY CHEAPEST$(\mathcal{U}, \mathcal{S}, c)$
$Sol \leftarrow \emptyset$, $C \leftarrow \emptyset$
while $C \neq \mathcal{U}$ **do**
  $s \leftarrow \arg\min_{s \in \mathcal{S}: s \cap (\mathcal{U} \setminus C) \neq \emptyset} c(s)$
  $Sol \leftarrow Sol \cup \{s\}$
  $C \leftarrow C \cup s$
return Sol

Runtime. The algorithm sorts the sets by cost, which takes $\mathcal{O}(|\mathcal{S}| \log |\mathcal{S}|)$ time. It then iterates over the sorted sets, and verifies whether they cover a new element. Each added set covers at least one new element, therefore the initial solution has $\mathcal{O}(|\mathcal{U}|)$ sets. Both operations of adding a new set and verifying if a set covers a new element require iterating over every element covered by the selected set. Together, the runtime is therefore

$\mathcal{O}(|\mathcal{S}| \log |\mathcal{S}| + |\mathcal{S}||\mathcal{U}| + |\mathcal{U}|^2)$. Since $|\mathcal{S}|$ and $|\mathcal{U}|$ are both $\mathcal{O}(n^2)$, the overall runtime is $\mathcal{O}(n^4)$. However, in case the sets are small or there are few sets, the runtime is expected to be significantly better.

Solution quality. Unfortunately, GREEDY CHEAPEST can produce arbitrarily bad solutions. Consider a set cover instance with $n > 2$ elements and $n + 1$ sets. The first n sets are singletons, each with cost 1. Each covers a single, pairwise distinct element. The last set covers all n elements and has cost $1 + \epsilon$ for some $\epsilon > 0$. GREEDY CHEAPEST will select the n singleton sets, for a total cost of n, while the optimal solution is to select the last set, for a total cost of $1 + \epsilon$.

4.3.3 SC-ILP

We translate the EILP algorithm into the set cover context as SC-ILP (Set Cover ILP), formulated as follows:

$$\begin{aligned} \min \quad & \sum_{s \in \mathcal{S}} c(s) x_s, & \\ \text{subject to} \quad & \sum_{s \in \mathcal{S}: e \in s} x_s \geq 1 & (e \in \mathcal{U}), \\ & x_s \in \{0, 1\} & (s \in \mathcal{S}). \end{aligned}$$

Here, x_s indicates whether set s belongs to the solution, and each covering constraint requires an element of $\mathcal{U}$ to belong to at least one selected set. The set cover formulation of the EILP is identical to the original one, up to renaming. Therefore, the set cover approach can only improve the performance with a faster construction. Since the EILP is dominated by the solving stage, a significant improvement is not expected.

4.4 Solving the Cycle Problem: SUPERNOVA

From the previous sections, we have learned that a considerable impediment to the performance of the algorithms is the presence of big cycles in the graph to be augmented. Cycles make the resulting set cover instance large: each cycle $Cyc_i \in Cyc(C)$ contributes $\binom{|Cyc_i|}{2}$ elements. Moreover, the number of elements covered by each set is generally also large. Further exacerbating the situation, links crossing the same cycle can have substantial overlap in their coverage, forcing the greedy algorithm to recompute many scores.

On the other hand, for stars, the resulting set cover instance has a small $|\mathcal{U}|$, as they are trees. Links can cover at most two elements per star, and the coverage overlap within a star is at most two elements.

Hence, we seek to solve the problems cycles introduce with a simple ansatz: *pretend that cycles are stars*.

We do this by solving the connectivity augmentation problem with whichever method we choose on the block-tree of the input graph. The idea is that a solution to the block-tree may already be very close to a solution to the original instance. As we shall see in the following analysis, the block-tree is a cactus graph that represents a subset of the minimum cuts of the original instance. Therefore, this method works under the premise that the block-tree represents a strong subset of the total minimum cuts present in the original instance, in the sense that if those minimum cuts are covered, a large portion of the remaining minimum cuts are likely to be collaterally covered.

After solving the block-tree, we conceptually contract the chosen links, obtaining a *residual instance*. The residual instance is a cactus graph representing the minimum cuts not covered by the links in the previous iteration. There are two ways to proceed from here. One can either lock the choice of links and continue to solve the residual instance, or add the violated constraints, i.e. the uncovered minimum cuts, to the original instance and solve the resulting instance from scratch. This choice results in a trade-off between the quality of the solution and the time taken to compute it. The more you attempt before locking, the better the solution is likely to be, but the longer it will take to compute. However, regardless of choice, this method can reach a feasible solution by potentially considering a small number of minimum cuts.

In other words, this approach can be understood as a lazy constraint generation method, where the constraints are heuristically chosen. Throughout this thesis, we have exclusively chosen to restrict the instance to the block-tree due to the properties specified previously. Namely, the coverages of each link and their intersections with one another are small, and we can apply strong and fast data reductions since the block-tree is a tree. Additionally, the approximation guarantee for the GREEDY SET COVER algorithm is generally better on trees, especially if they have small diameter.

This serves as an initial investigation into the potential of lazy constraint generation methods for solving the WCA problem. In the following we clarify this by describing the general framework of the SUPERNOVA algorithm and providing a theoretical analysis of its correctness. Then, we present the concrete implementation choices we have made that were used in our experimental evaluation.

4.4.1 General Framework

In a sense, SUPERNOVA is an algorithmic framework. That is, at every iteration, a few decisions must be made: which minimum cuts to restrict the current problem to, what specific solver and configuration to use and whether to lock the resulting solution or not. For instance, depending on the current residual, either a stronger or weaker set of constraints may be more effective. Additionally, a different approximation or heuristic algorithm may yield a better solution or quality guarantee for that specific residual.

We provide detailed pseudocode for this framework in Algorithm 7. The operation $\mathrm{ChooseCuts}(R)$ returns a nonempty family of minimum cuts of a nontrivial residual R. The default choice is the family represented by $B(R)$. The operation $\mathrm{Residual}(C, L, c, S)$

returns a residual cactus, its candidate links and costs, and a map ρ from each residual link to a cheapest representative in L. The operation Lift maps residual cuts back to cuts of the current committed instance C. The map $\mathrm{original}$ tracks links in the residual back to the input instance.

Algorithm 7: SUPERNOVA

Input: A feasible cactus instance (C, L, c)
Output: An augmentation Sol of the input instance
$Sol \leftarrow \emptyset$, original $\leftarrow$ identity
while $|V(C)| > 1$ **do**
  $\mathcal{K} \leftarrow \emptyset$, $R \leftarrow C$, lock $\leftarrow$ false
  repeat
    $\mathcal{K} \leftarrow \mathcal{K} \cup \mathrm{Lift}(\mathrm{ChooseCuts}(R))$
    $\mathcal{A} \leftarrow$ choose a solver
    $S \leftarrow \mathcal{A}(\mathcal{K}, L, c)$
    $(R, L_R, c_R, \rho) \leftarrow \mathrm{Residual}(C, L, c, S)$
    lock $\leftarrow (|V(R)| = 1)$ or choose whether to commit S
  until lock
  $Sol \leftarrow Sol \cup \mathrm{original}(S)$
  original $\leftarrow$ original $\circ \rho$
  $(C, L, c) \leftarrow (R, L_R, c_R)$
return Sol

For clarity, only committed solutions contribute to the final augmentation. Discarded trial solutions do not. Before commitment, the reduced link set returned by $\mathrm{Residual}$ is used only to describe possible future residual instances, not to restrict the next solve on $(\mathcal{K}, L, c)$. On a feasible instance, each refinement adds an uncovered cut and each commitment covers a nonempty set of cuts. Since the original cut family is finite, these choices cannot continue indefinitely without covering all cuts.

4.4.2 Theoretical Analysis

First, we must understand why SUPERNOVA is correct. For that, we must show that the block-tree represents a subset of the minimum cuts of the original instance. Specifically, it represents the minimum cuts consisting of consecutive edges in a cycle. If we restrict the definition of the cactus representation of the minimum cuts of a graph, presented in Chapter 2, to only represent a subset $\mathcal{K}' \subseteq \mathcal{K}(C)$ of the minimum cuts, and we choose $\mathcal{K}'$ to be the consecutive-edge minimum cuts, then the resulting representation is the block-tree. To see that, consider a cycle $Cyc' \in Cyc(C)$. Let the vertices and edges in the cycle be $v_1, \ldots, v_n$ and $e_i = \{v_i, v_{i+1}\}$, respectively, with indices taken cyclically. Then, a consecutive-edge minimum cut induced by e_i, e_{i+1} will partition the cactus into the part

attached to v_{i+1} and its complement. If we take b to be the vertex added by the block-tree construction to the cycle Cyc', then the minimum cut induced by $\{b, v_{i+1}\}$ will partition the block-tree in the same way. So the block-tree represents consecutive-edge minimum cuts. Therefore, solving the WCA problem on the block-tree is equivalent to covering a subset of $\mathcal{U}$ from the WSC perspective.

Alternative Constraints. In fact, there is an interesting geometric view of the construction of the tree representation of a subset of minimum cuts in a cycle. If we draw the cycle in the plane, then for every minimum cut corresponding to the edge pair $\{e_i, e_j\}$, we can draw a line segment connecting the midpoints of e_i and e_j. As long as no two segments intersect in their interiors, we can construct a tree representing the minimum cuts. We do so by placing a vertex in every region separated by the segments and joining vertices whose regions are adjacent. Each original cactus vertex belongs to the region containing its position between consecutive cycle edges. Figure 4.12 illustrates this construction for consecutive-edge cuts and for a larger family of non-intersecting cuts.

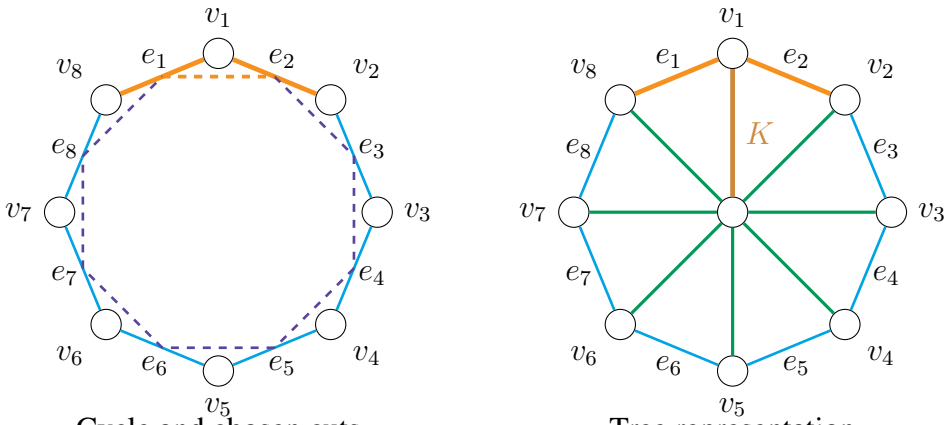
That means at every stage, we have additional potential options for selecting minimum cuts. Stars are a particularly simple option, because every link covers at most two cuts per star. Other choices may yield stronger or weaker restrictions. In general terms, weaker restrictions can be solved faster, admit more reductions and stronger solution quality guarantees for the GREEDY SET COVER algorithm. On the other hand, stronger constraints are more likely to produce a smaller residual instance.

To guarantee the correctness of our approach, we must show that contracting the links of a partial solution produces a cactus representation of the uncovered minimum cuts, so that SUPERNOVA exposes every minimum cut. To *contract a link* means to contract its projection edges. The following lemma states that this remaining cactus represents exactly the minimum cuts we still need to cover.

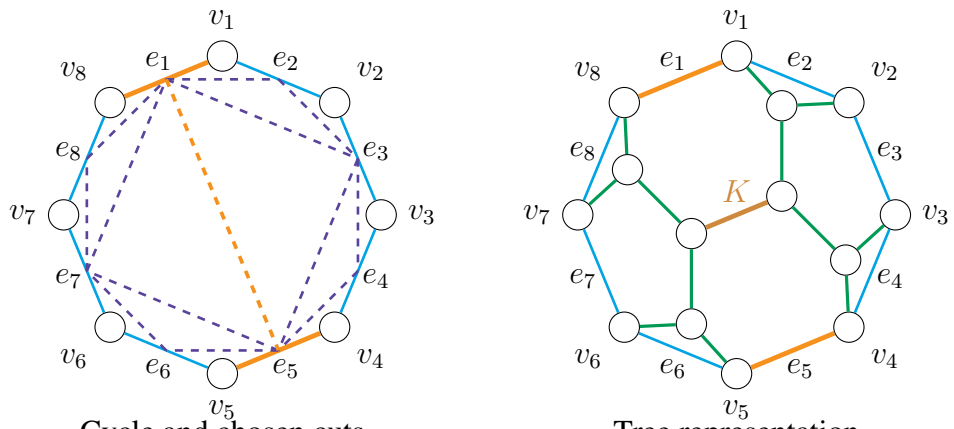
Lemma 9 (Remaining minimum cuts)
Let $S \subseteq L$ be a set of chosen links, and let R be the cactus obtained by contracting the projection edges of all links in S. The minimum cuts of R correspond exactly to the minimum cuts of C not covered by S.

Proof. Consider a minimum cut K not covered by S. First, contracting any link in S preserves the shores of K. If a contraction edge of some link in S covered K, then so would the link itself, which is a contradiction. Therefore, all uncovered minimum cuts are preserved in R. Conversely, assume there exists a minimum cut K' in R that does not correspond to an uncovered minimum cut in C. Contractions cannot create new minimum cuts, so K' must correspond to a minimum cut K in C. If it was covered by $l \in S$, then K' would be covered by one of its contraction edges. But every minimum cut separating the endpoints of this contraction edge is no longer a cut, because the endpoints have been joined together. Therefore, K' must correspond to an uncovered minimum cut in C, concluding our proof. □

Consequently, solving the remaining cactus completes the chosen links to a solution



(a) Consecutive-edge cuts yield a star; the highlighted cut K is induced by e_1, e_2.



(b) A larger family of non-intersecting cuts; the highlighted cut K is induced by e_1, e_5.

Figure 4.12: Two families of minimum cuts and their tree representations. Cycle edges are blue, and dashed segments on the left indicate the chosen minimum cuts. Orange highlights the edges inducing the cut K and its dashed segment; the corresponding tree edge on the right is dark orange. Other tree edges are green. The cycle is also shown on the right for orientation and is not part of the tree.

of the original instance. If only one vertex remains, then all minimum cuts are already covered.

Solution Quality

This view helps us understand the theoretical implications of our approach. If we do not lock intermediate solutions, then we keep solving restrictions of the same original instance. Any solution to the full instance also satisfies the restricted one, because there are fewer minimum cuts to cover. Let OPT and OPT' denote the optimum values of the full and restricted instances, respectively. Then clearly $OPT' \leq OPT$. If Sol' is an α-approximate solution to the restricted instance, then $c(Sol') \leq \alpha OPT' \leq \alpha OPT$. Therefore, if Sol' also covers every minimum cut of the full instance, it is an α-approximation for it. In particular, an exact solution to the restricted instance that is feasible for the full instance is also exact for the full instance.

Locking a solution has a different effect on the bounds. Once the chosen links are fixed, their cost remains part of the final solution, even if a later iteration would prefer different links. Thus, the approximation bounds of the locked solutions add up.

Theorem 10
Suppose SUPERNOVA *finishes after locking k solutions. For each i, let the solver be an α_i-approximation for the restricted instance considered at the i-th locking step, and let Sol_i denote the chosen links in the original instance. Then $Sol = \bigcup_{i=1}^{k} Sol_i$ is a $\sum_{i=1}^{k} \alpha_i$-approximation for the original instance.*

Proof. Let Sol^* and Sol_i^* be an optimal solution to the original instance and the i-th restricted instance, respectively. Since every iteration corresponds to covering a subset of the original minimum cuts, $c(Sol_i^*) \leq c(Sol^*)$. Since the i-th solver is an α_i-approximation, we obtain $c(Sol_i) \leq \alpha_i c(Sol_i^*) \leq \alpha_i c(Sol^*)$. Summing over i gives $c(Sol) = \sum_i c(Sol_i) \leq \sum_i \alpha_i c(Sol^*)$. □

Theorem 10 implies that even if we use an exact solver at every iteration, every locked solution adds 1 to the final approximation bound. So if we lock solutions twice, we obtain a 2-approximation. Therefore, at every iteration, we must consider whether we accept the current bound and residual and lock the solution, or whether we try again with a different restriction and solver combination.

Repeated Block-Tree Restrictions

In the following, we analyze the strategy of repeatedly choosing the block-tree restriction and locking the resulting solution. We show deliberately pessimistic worst-case bounds. Our goal is to show that the number of minimum cuts considered and the maximum coverage of a link with respect to these cuts are linearly bounded by the number of vertices.

Let C_i be the remaining cactus before the i-th iteration, with $C_0 = C$, and define $n_i = |V(C_i)|$. Let $\mathcal{K}_B(C_i)$ denote the minimum cuts of C_i represented by its block-tree. For a cycle of length 2, both block-tree edges represent the same cut, which we count only once.

Lemma 11
If we solve and lock the block-tree restriction of a nontrivial C_i, then $n_{i+1} \leq n_i/2$.

Proof. Since every block-tree edge must be covered, every vertex of C_i lies on the block-tree path of some chosen link. Its projection path contains this vertex and at least one other vertex of C_i. Contracting the chosen links therefore joins every vertex with at least one other vertex. Our bound immediately follows. □

Corollary 12
The maximum number of iterations is $\lceil \log_2 n_0 \rceil$.

We next bound the number of minimum cuts considered in one iteration. For a cactus with n vertices, let t be the number of cycles of length 2 and r the number of longer cycles. Each 2-cycle contributes one cut, while every longer cycle contributes one consecutive-edge cut per vertex. Thus,

$$|\mathcal{K}_B(C)| = t + \sum_{\substack{Cyc' \in Cyc(C) \\ |V(Cyc')| \geq 3}} |V(Cyc')|.$$

Removing one edge from every cycle of length at least 3 gives a tree, so

$$n - 1 = t + \sum_{\substack{Cyc' \in Cyc(C) \\ |V(Cyc')| \geq 3}} (|V(Cyc')| - 1).$$

The two expressions differ by r, giving $|\mathcal{K}_B(C)| = n + r - 1$. Every longer cycle contributes at least 2 to the second sum, so $r \leq (n-1)/2$. Therefore, $|\mathcal{K}_B(C)| \leq \frac{3}{2}(n-1)$.

Theorem 13
The repeated block-tree strategy considers at most $3n_0$ minimum cuts in total, counting cuts considered in different iterations separately.

Proof. In the i-th iteration, we consider at most $\frac{3}{2}(n_i - 1)$ cuts. Since $n_i \leq n_0/2^i$, summing over all iterations gives

$$\sum_i |\mathcal{K}_B(C_i)| \leq \frac{3}{2} \sum_i (n_i - 1) \leq \frac{3}{2} \sum_{i \geq 0} \frac{n_0}{2^i} = 3n_0.$$

□

Theorem 14
An original link covers at most $2n_0$ of the considered minimum cuts in total, again counting contributions from different iterations separately.

Proof. Let l_i be the link between the vertices of C_i containing the endpoints of the original link l. If both endpoints have been contracted together, the link covers no remaining cuts. Otherwise, it covers one considered cut for every 2-cycle it crosses and at most two for every longer cycle. Let t_i and r_i be the numbers of 2-cycles and longer cycles in C_i, respectively. Then

$$|cov(l_i) \cap \mathcal{K}_B(C_i)| \leq t_i + 2r_i \leq t_i + \sum_{\substack{Cyc' \in Cyc(C_i) \\ |V(Cyc')| \geq 3}} (|V(Cyc')| - 1) = n_i - 1.$$

Summing over the iterations gives at most $\sum_i (n_i - 1) \leq 2n_0$ considered cuts covered by l. □

To finalize our analysis, we combine these bounds with the approximation bound for GREEDY SET COVER. In each iteration, the largest set has size $\Delta_i \leq n_i - 1 \leq n_0$. Since $H(n) \leq \ln n + 1$ and there are at most $\lceil \log_2 n_0 \rceil$ iterations, Theorem 10 gives

$$\sum_i H(\Delta_i) \leq \lceil \log_2 n_0 \rceil (\ln n_0 + 1) = \mathcal{O}(\log^2 n_0).$$

Thus, repeatedly locking and solving the block-tree restriction with the greedy algorithm is an $\mathcal{O}(\log^2 n_0)$-approximation.

This means that repeated application of the greedy approach is a form of gamble. In the first iteration, Δ_1 can be much smaller than the largest set of the full instance. A cycle Cyc' of length at least 3 contributes at most 2 considered cuts per link instead of up to $\lfloor |V(Cyc')|^2/4 \rfloor$. If the first restricted solution already solves the full instance, we keep this smaller approximation bound. However, every further locked solution adds another term to the bound. This does not necessarily mean that the actual solution becomes worse, but our guarantee becomes weaker.

4.4.3 Implemented Variants

So far, we have described the choices available in the general framework and analyzed repeated block-tree restrictions. For the purposes of experimental evaluation, we have implemented two specific variants of the SUPERNOVA framework. We now describe how they proceed and how previous bounds apply to them.

FROZEN SUPERNOVA

In the frozen variation, we first solve the block-tree restriction and lock the chosen links. We then contract them and solve the full remaining cactus. Thus, we solve at most two instances, rather than repeatedly replacing the remaining cycles by stars. For either instance, we can use one of the set cover solvers described previously. If the first solution already covers all minimum cuts, we do not need the second solve. This variant expects that the first solution already produces a small enough residual, so that it may be solved directly.

The previous results immediately give a bound for this variation. If the two solvers have approximation guarantees α_1 and α_2, then Theorem 10 gives an $(\alpha_1 + \alpha_2)$-approximation. In particular, using exact solvers gives a 2-approximation, while GREEDY SET COVER gives $H(\Delta_1) + H(\Delta_2) = \mathcal{O}(\log n)$. However, only the first instance is restricted to the block-tree. The second instance may contain quadratically many minimum cuts, so the linear bound on the total number of considered cuts does not apply to this variation.

LAZY SUPERNOVA

In the lazy variation, we do not lock intermediate solutions and always use an exact ILP as the solver. We start with one ILP containing a variable for every original link, but only the constraints for the block-tree cuts $\mathcal{K}_B(C)$. Whenever the solver finds a candidate integer solution Sol, we construct its residual cactus. If more than one vertex remains, we take the cuts represented by the remaining block-tree, and add them to the ILP model. In particular, the ILP solver can continue working on the same model rather than starting from scratch. All original link variables remain available at all times and may be reconsidered. Therefore, link reductions on the restricted instances are not applied. Hence, LAZY SUPERNOVA is an exact algorithm, provided that the ILP solver is exact.

4.4.4 Implementation Details: BULKCONTRACTIONS

Another relevant detail of our algorithm is that we perform all contractions in bulk. Throughout the project, we insist on using the CSR data structure to represent sparse graphs. CSR graphs are static data structures, so contracting edges is not an operation natively supported by their interface. Instead, we maintain a union-find data structure that describes which nodes contract together, and a map from the links on the contracted graph to the original links.

To determine which vertices belong to the same union-find class, we proceed in two phases. In the first phase, for every selected link, we join all vertices on its projection path. In a tree, this phase suffices. In the presence of cycles, however, vertices may join a class if they cross in a cycle in the following sense.

Let Cyc' be a cycle with vertices $(v_1, \ldots, v_n, v_1)$ in cyclic order. Suppose that, after the first phase, $v_a \sim v_b$ and $v_c \sim v_d$ for some $a < c < b < d$, where $\sim$ indicates that vertices

belong to the same contraction class. In that case, these two classes are said to cross, so that all vertices must be merged, i.e. $v_a \sim v_b \sim v_c \sim v_d$.

We detect these patterns by scanning each cycle while keeping track of the active classes. We first record the union-find classes of each vertex after joining the vertices on the projection paths. Then for every class, we store its last occurrence on that cycle. Next, we scan the vertices in order while keeping a stack of active local classes. A class becomes active when it is encountered and remains so until its last occurrence has been reached. The idea is that whenever we encounter an active class, any other active class above it on the stack must cross with it, and therefore must be merged. Conversely, whenever two classes cross, the second occurrence of the earlier class is encountered while the other class is active. Therefore, the algorithm detects every crossing. The pseudocode of this algorithm is shown in Algorithm 8.

Algorithm 8: BULKCONTRACTIONS

Input: Cactus graph $C = (V, E)$ and selected links L'
Output: Union-find structure U describing the contracted vertices

$U \leftarrow$ a union-find structure containing one class for each $v \in V$
`// Join the vertices on every selected projection path.`
foreach $l \in L'$ **do**
  $(p_0, p_1, \ldots, p_k) \leftarrow P_l$
  for $i \leftarrow 1$ **to** k **do**
    $U.\,\mathrm{union}(p_{i-1}, p_i)$
`// Merge classes whose occurrences cross on a cycle.`
foreach $Cyc' = (v_1, \ldots, v_n, v_1) \in Cyc(C)$ **do**
  for $i \leftarrow 1$ **to** n **do**
    $x_i \leftarrow U.\,\mathrm{find}(v_i)$
    $\mathrm{last}[x_i] \leftarrow i$
    $\mathrm{active}[x_i] \leftarrow \mathrm{false}$
  $S \leftarrow$ an empty stack
  for $i \leftarrow 1$ **to** n **do**
    $x \leftarrow x_i$
    if $\neg\,\mathrm{active}[x]$ **then**
      $\mathrm{active}[x] \leftarrow (i < \mathrm{last}[x])$
      if $\mathrm{active}[x]$ **then**
        $S.\,\mathrm{push}(x)$
    else
      while $S.\,\mathrm{top}() \neq x$ **do**
        $y \leftarrow S.\,\mathrm{pop}()$
        $j \leftarrow \max\{\mathrm{last}[x], \mathrm{last}[y]\}$
        $U.\,\mathrm{union}(x, y)$
        $x \leftarrow U.\,\mathrm{find}(x)$
        $\mathrm{last}[x] \leftarrow j$, $\mathrm{active}[x] \leftarrow \mathrm{true}$
      $S.\,\mathrm{pop}()$
      $\mathrm{active}[x] \leftarrow (i < \mathrm{last}[x])$
      if $\mathrm{active}[x]$ **then**
        $S.\,\mathrm{push}(x)$
return U

CHAPTER 5

Experimental Evaluation

The experiments examine whether the structural techniques developed in the previous chapter improve connectivity augmentation in practice. We first compare the set cover representations, then study the effect of data reductions on exact and heuristic solvers. Finally, we compare LAZY SUPERNOVA with the direct ILP formulation and evaluate the solution quality and runtime of the frozen variants against GWC and MSTCONNECT with local search.

5.1 Experimental Setup

We first describe our experimental setup, including instance generation and measurement methodology, and then summarize the configurations used in the evaluation.

We use four classes of synthetic instances: cycles, stars, trees, and cacti. Cycles expose the worst-case growth of the explicit set cover representation, since one cycle represents quadratically many minimum cuts. Stars maximize the number of leaves among trees of a fixed size, and therefore require many links to cover their incident minimum cuts. The tree instances test whether findings on stars extend to less concentrated tree structures. The cactus instances further test the generalization power of the algorithms. They are generated by successively attaching cycles of size between 2 and 16 at existing vertices.

We use two link-cost distributions: the uniform real distribution on $(0, 1]$ and the uniform integer distribution on $\{1, \ldots, 5\}$, abbreviated real and integer, respectively, in the following. The candidate link sets are complete except for links parallel to cactus edges. This extends the synthetic setup of Faraj, Großmann, Joos, Möller, and Schulz [6]. This setup gives all algorithms many alternatives. The integer distribution introduces many equal cost choices, which lead to less potential for data reductions.

We run the algorithms on the synthetic instances as follows. For every family and instance size, we use five matched seeds for the graph and link generation. Cycles and stars yield the same graph at a given size, but trees and cacti produce different graphs and link

distributions for each seed. We execute each algorithm on instances with size starting at $n = |V(C)| = 20$ and double that value until all five attempts at a size fail, independently for each family and cost distribution.

The following hardware and software configuration is used for all experiments. All benchmarks were performed on an AMD EPYC 9754 processor with 128 physical cores and 256 hardware threads. It has a base clock frequency of 2.25 GHz, max boost clock of 3.1 GHz and total available RAM of 755 GiB. We use Gurobi 13.0.1 for solving the ILP models. All code is written in C++20 and compiled with GCC 13.3.0 with optimization level `-O3` and `-march=native`. Every algorithm process was allocated an external budget of 300 seconds and 16 GiB of memory. Runtime is the recorded total time from input reading to outputting the final solution. In particular, it excludes instance generation and the construction of the cactus representation. Memory is the recorded peak resident memory in system pages of 4096 bytes (4 KiB) each. We also record the final solution cost and use stage timings where they help explain the total runtime.

5.2 Set Cover Representations

In this section we compare our set cover representation against the native connectivity augmentation approach by Faraj, Großmann, Joos, Möller, and Schulz [6]. We compare GWC with our analogous GREEDY SET COVER algorithm.

Figures 5.1–5.3 compare runtime and memory for real link costs, and solution quality for both cost distributions. We compare three configurations: the native approach GWC and two configurations of our GREEDY SET COVER algorithm. The first configuration uses the explicit set cover representation in CSR format, and the second uses the CycSC representation with PAM for cycles of length 2. The pure CSR approach additionally includes *trimming*, which is the removal of redundant sets from the solution after the greedy selection process. It is currently a topic of further research how to implement trimming efficiently on the CycSC representation.

The preferred representation depends strongly on the graph structure. On stars and trees, the explicit CSR representation is substantially faster than its competitors. At $n = 2560$, CSR with trimming is 23.9 times faster than GWC on stars and 86.0 times faster on trees, comparing median total runtimes, despite the additional overhead of trimming. Memory usage remains similar on stars, but CSR requires more memory on larger trees.

On the other hand, on cycles, explicitly storing the set cover instance becomes prohibitively expensive. At $n = 320$, CSR requires approximately 13 GiB, compared with 10 MiB for CycSC. CycSC avoids this growth and gives the lowest runtimes on larger cycles and cacti, although GWC generally uses less memory. On cacti, CycSC completes all five instances up to $n = 10240$, four times the largest completed size of either GWC or pure CSR with trimming.

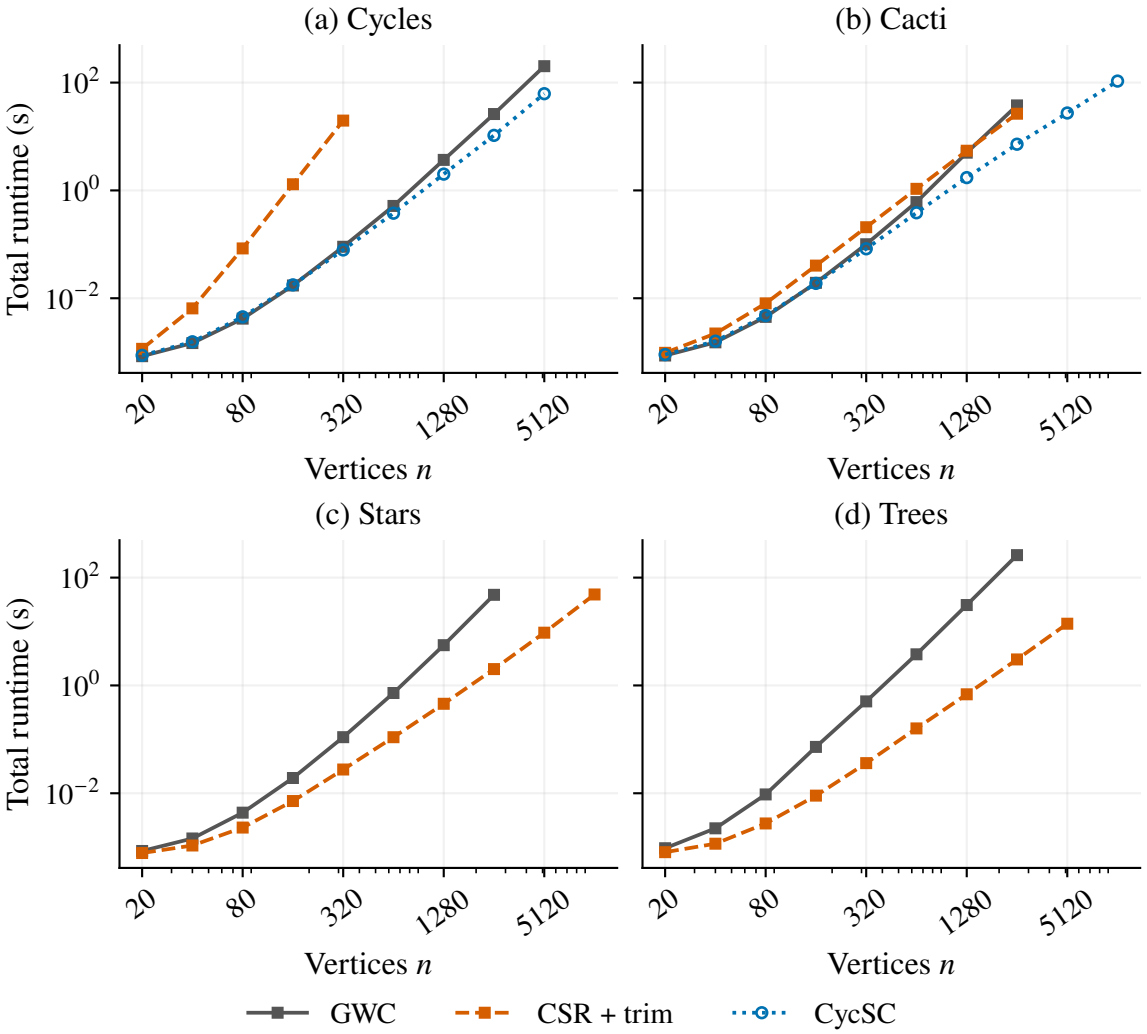


Figure 5.1: Runtime of GWC and GREEDY SET COVER for link costs drawn uniformly from $(0, 1]$. Each point shows the median total runtime over five completed samples, including construction and, for CSR + trim, trimming. Sizes with fewer than five completions are omitted. Both axes are logarithmic and use common ranges across panels.

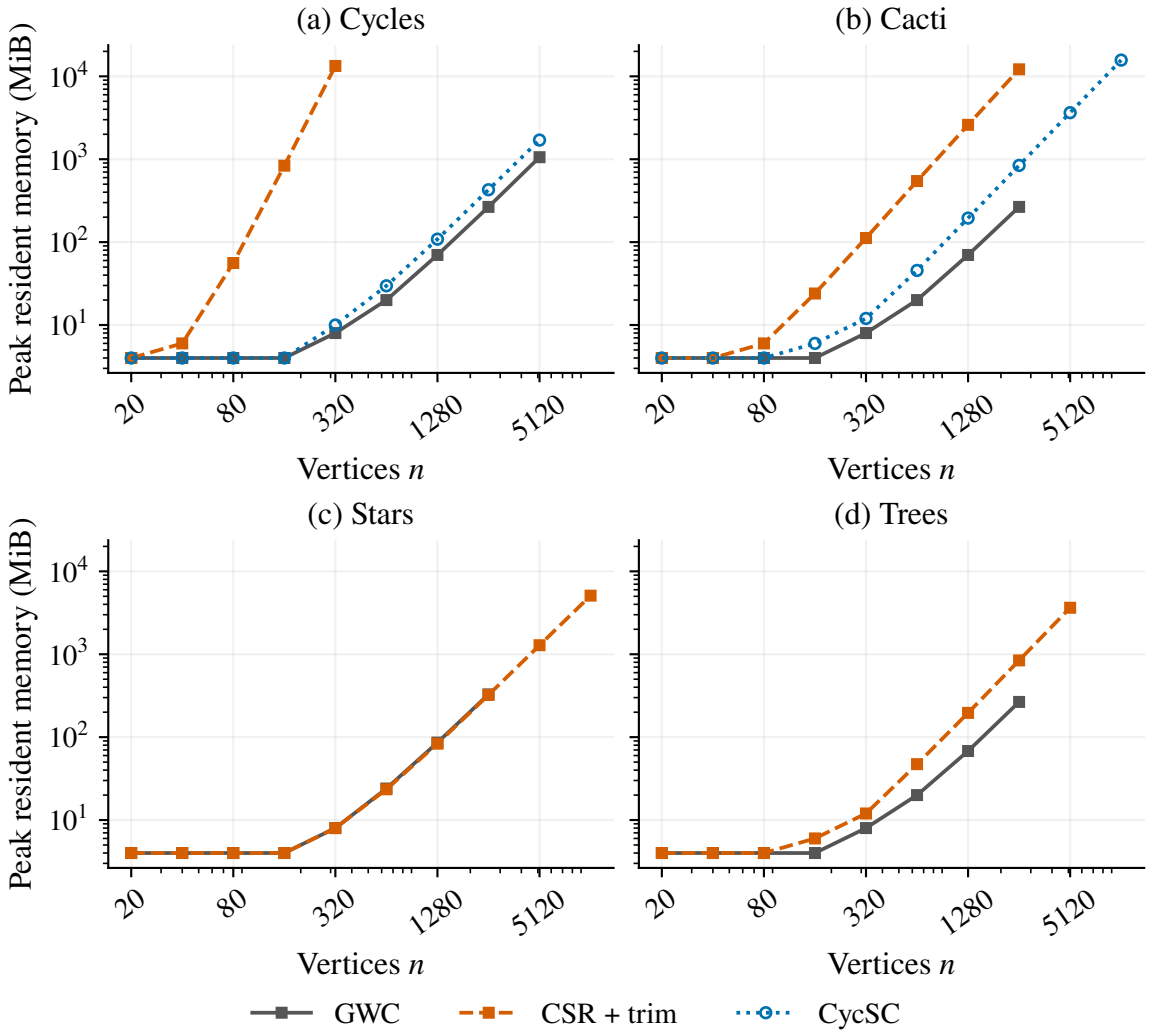


Figure 5.2: Peak resident memory of GWC and GREEDY SET COVER for real link costs. Each point shows the median over five completed samples, converted from 4096-byte pages to MiB. Sizes with fewer than five completions are omitted. Both axes are logarithmic and use common ranges across panels.

The solution quality profiles show the additional benefit of trimming. We have included only the results for trees, because the results for the other families are similar. On real-cost trees, CSR with trimming lowers the median gap to the reference from 8.55% for GWC to 1.89%. The improvement is smaller for integer costs, where both methods already produce solutions closer to the reference. CycSC and GWC return almost identical costs on their shared instances.

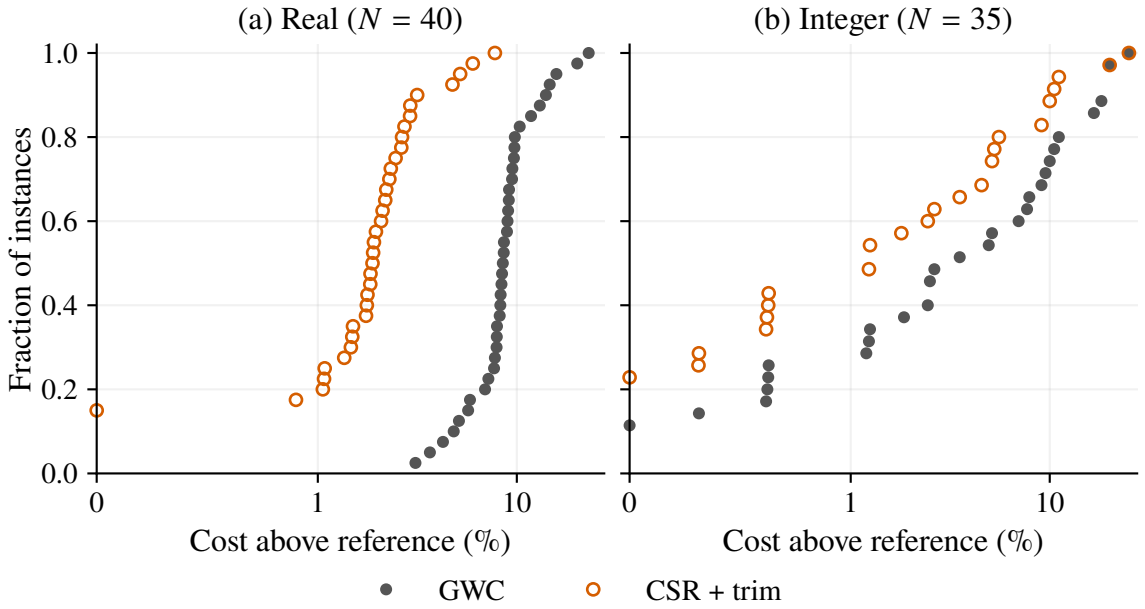


Figure 5.3: Solution quality of GWC and GREEDY SET COVER with CSR and trimming on trees. Dots show the fraction of shared completed instances within a given percentage above the reference, for (a) 40 real-cost and (b) 35 integer-cost instances; further left is better. The horizontal scale is linear from 0 to 1% and logarithmic thereafter. Hollow CSR + trim dots keep overlaps visible.

5.3 Heuristic SUPERNOVA

In this section, we investigate the use of SUPERNOVA to improve the performance of heuristic algorithms. We consider two heuristic SUPERNOVA configurations which we call GREEDY SUPERNOVA and ANXIOUS SUPERNOVA. Both are variants of FROZEN SUPERNOVA. They solve the block-tree in the first stage, commit and contract their selected links, and then solve the final full residual. They differ only in the solver they use. GREEDY SUPERNOVA applies GREEDY SET COVER in CSR format with trimming and no reductions. ANXIOUS SUPERNOVA uses the ILP solver in both stages, with reductions enabled. The former allows us to compare GREEDY SET COVER with its direct SUPERNOVA counter-

part. The latter allows us to compare FROZEN SUPERNOVA when a more powerful solver is used.

Both approaches are compared against a common baseline. For that, we merge the greedy configurations from Section 5.2 and choose the datapoint with the best performance. This simulates a hypothetical greedy algorithm that combines the advantages of all representations. We refer to it as BASELINE GREEDY. Since stars and trees offer no cycle structure to exploit, rendering SUPERNOVA ineffective, we focus on cacti and cycles. Figures 5.4–5.6 compare total runtime, peak resident memory, and solution quality for both cost distributions and all three compared approaches. Additionally, we include the best median independently for runtime and memory among all exact solvers at sizes with five completions. This helps us note the speedup trade-offs achieved with respect to optimal solving.

5.3.1 GREEDY SUPERNOVA

From Section 4.4, we know that the block-tree restriction imposed by GREEDY SUPERNOVA in the first stage will improve its approximation guarantee for the first solving stage. The second stage will, however, add to that factor. If the residual of the contraction is small, then a better approximation guarantee than that of the base greedy algorithm is to be expected. We verify if this intuition holds in practice.

For both families, GREEDY SUPERNOVA demonstrates runtime improvements. At $n = 5120$, GREEDY SUPERNOVA is 6.63 times faster on real-cost cycles and 13.33 times faster on integer-cost cycles. It also completes all five cycle instances at $n = 10240$ for both distributions, twice the largest fully completed size of BASELINE GREEDY. On cacti at $n = 5120$, the speedups are smaller: 1.62 for real costs and 1.18 for integer costs. Here, BASELINE GREEDY completes all five instances up to $n = 10240$, while GREEDY SUPERNOVA reaches only $n = 5120$.

On the other hand, GREEDY SUPERNOVA uses more memory than the baseline. On real-cost cycles at $n = 5120$, median peak memory is 1058 MiB for the baseline and 1381 MiB for GREEDY SUPERNOVA. On integer-cost cycles at $n = 2560$, it is 274 MiB and 349 MiB, respectively. The difference is larger on cacti. Peak memory usage is 266 MiB versus 1136 MiB for real costs at $n = 2560$, and 72 MiB versus 254 MiB for integer costs at $n = 1280$.

Solution quality is compared only on instances that have a reference cost and were completed by both SUPERNOVA variants and at least one baseline configuration. On these real-cost instances, GREEDY SUPERNOVA lowers the median gap from 4.17% for the selector to 2.64% on cycles. On cacti, both median gaps round to 2.73%. For integer costs, GREEDY SUPERNOVA does not improve the median gap over the selector. In fact, the median gap increases from 1.25% to 2.19% on the shared cycles, while the cactus gap increases from 1.37% to 2.74%.

In summary, we conclude that GREEDY SUPERNOVA has potential to improve runtime and solution quality of the greedy algorithm at the cost of increased memory usage. However, the improvement in solution quality is not consistent. This is likely due to the residual

instance being too large for the second stage, but more experiments are needed to confirm this hypothesis. It is also more effective when applied to larger cycles. The cacti used have only relatively small cycles, so a test with larger and fewer cycles would be of interest.

5.3.2 ANXIOUS SUPERNOVA

ANXIOUS SUPERNOVA generally requires more runtime than both previous approaches and more memory on cycles. This is expected, as ILP is much more expensive than greedy. On real-cost cycles at $n = 5120$, its median runtime is 79.15 seconds, compared with 9.39 seconds for GREEDY SUPERNOVA and 62.21 seconds for the baseline. Median peak memory is 2903, 1381, and 1058 MiB, respectively. Real-cost cacti at the same size are an exception to the memory penalty: ANXIOUS SUPERNOVA uses 2914 MiB, compared with 5158 MiB for GREEDY SUPERNOVA and 3646 MiB for the baseline. This is due to the effectiveness of the reductions, which lower the instance sizes.

It is however more effective in terms of solution quality. This is most dramatic on cycles with real costs, where it achieves an optimal solution on all 40 shared instances. That is because the first solve contracts the instance to a single vertex in all 45 completed real-cost cycle runs. On the other datasets it is also significant. On the shared real-cost cacti, ANXIOUS SUPERNOVA lowers the median gap to 0%, compared with 2.73% for both GREEDY SUPERNOVA and the baseline. In other words, it solves optimally at least 50% of the shared instances. For integer-cost cacti, the gaps for the corresponding algorithms are 0.18%, 2.74%, and 1.37%, also an improvement.

The computational savings over exact solving depend on the graph family and cost distribution. On real-cost cacti at $n = 5120$, ANXIOUS SUPERNOVA takes 90.15 seconds and 2914 MiB, compared with 81.33 seconds and 2801 MiB for the exact baseline. This could be attributed to the types of reductions applied. On integer-cost cycles at $n = 2560$, ANXIOUS SUPERNOVA is also slower: 43.19 seconds versus 20.85 seconds. While the exact approach uses reductions on the instance as a whole, ANXIOUS SUPERNOVA applies reductions to the block-tree in the first stage and to the residual in the second stage. However, on real-cost cycles at $n = 2560$, it reduces runtime from 21.69 to 11.66 seconds and peak memory from 4715 to 768 MiB, improvements by factors of 1.86 and 6.14.

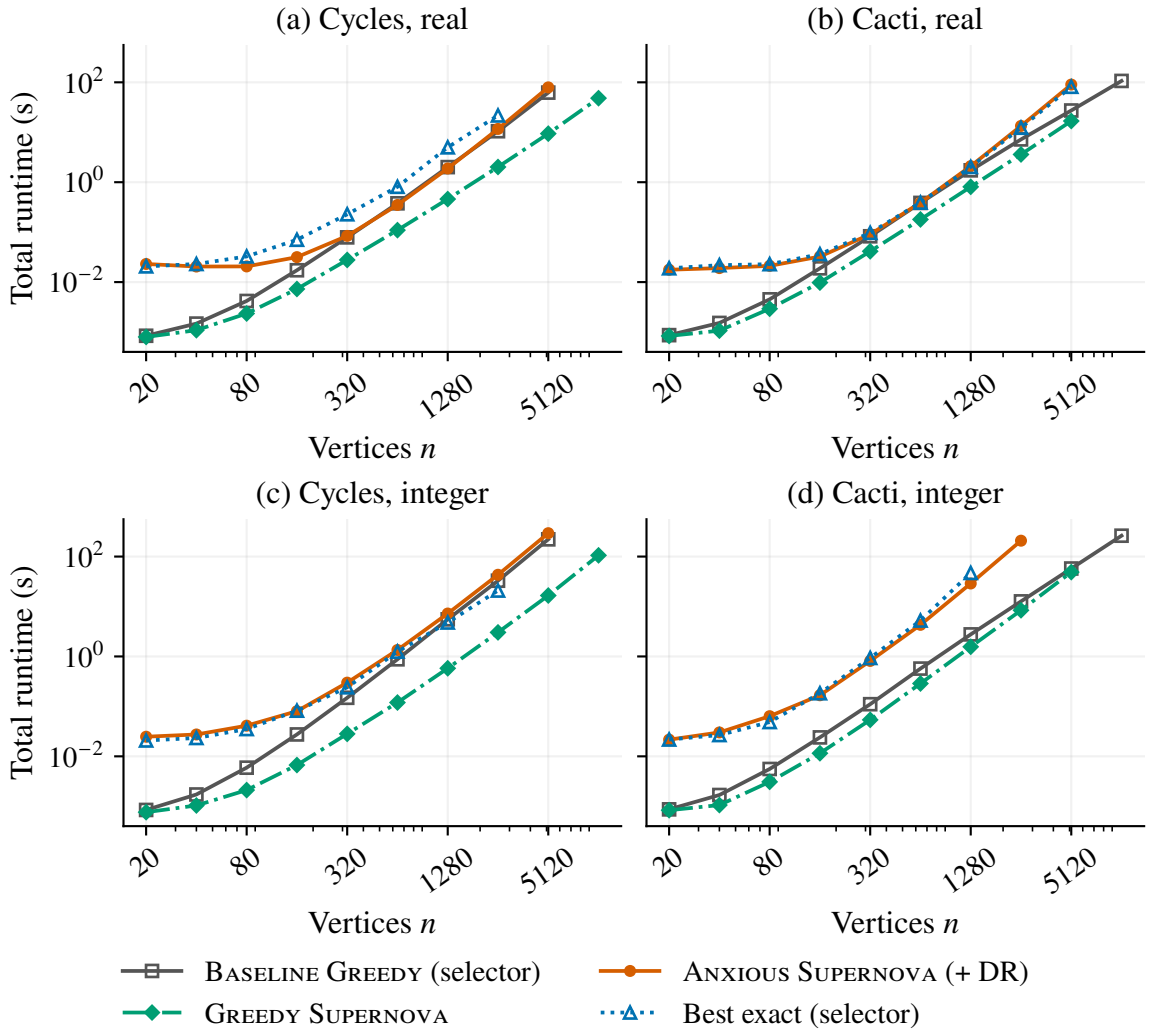


Figure 5.4: Total runtime of Greedy Supernova, Anxious Supernova, and the hypothetical selector on cycles and cacti, with real costs in the top row and integer costs in the bottom row. Each point is the median over five completed samples, including construction, contraction, reductions, and trimming where applicable. Only Anxious Supernova enables reductions. Sizes with fewer than five completions are omitted. The selector takes the lowest median among GWC, CSR with trimming, and CycSC at each size. Both axes are logarithmic, with common ranges across panels.

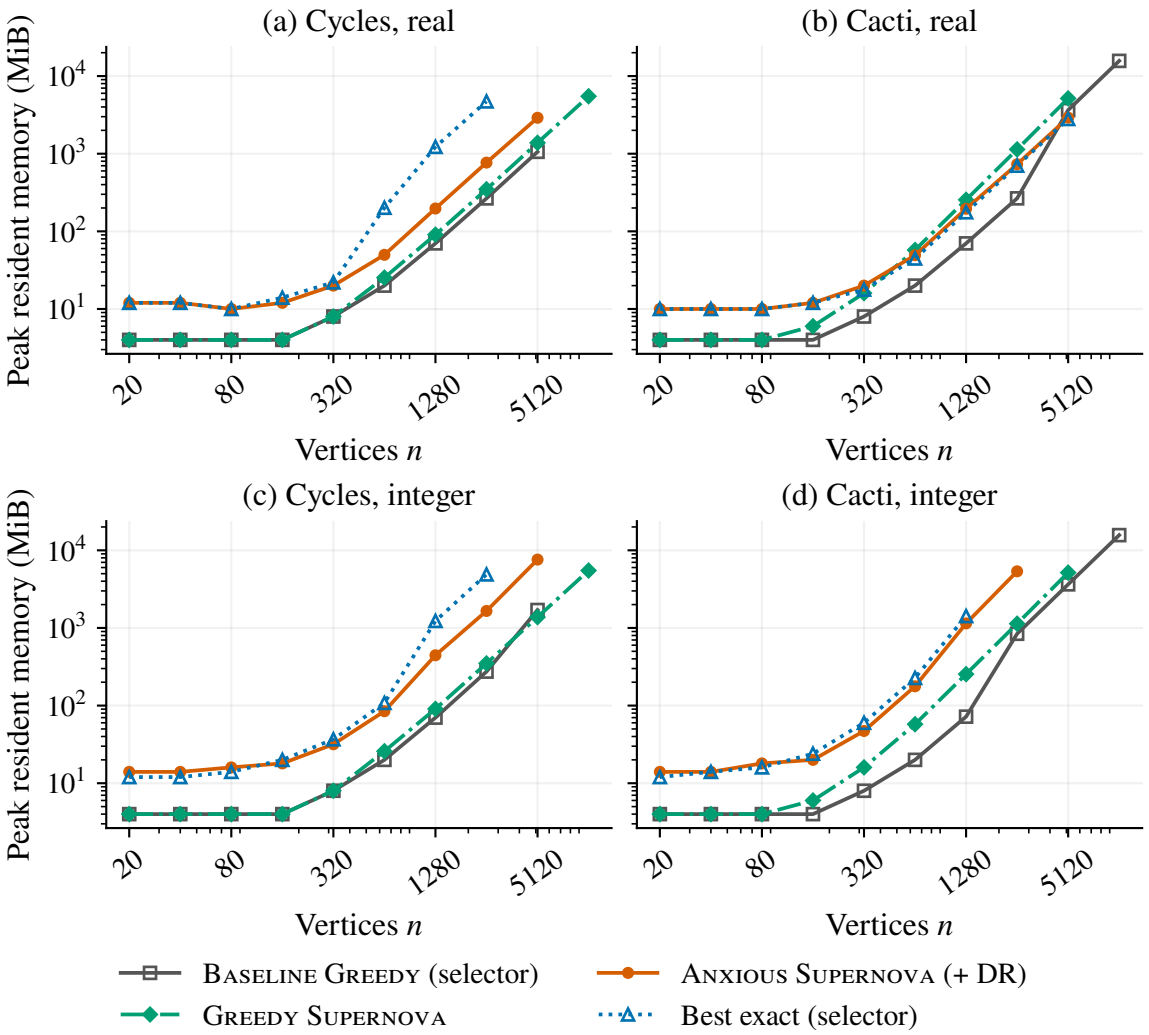


Figure 5.5: Peak resident memory of GREEDY SUPERNOVA, ANXIOUS SUPERNOVA, and the hypothetical selector on cycles and cacti, with real costs in the top row and integer costs in the bottom row. Only ANXIOUS SUPERNOVA enables reductions. Each point is the median over five completed samples, converted from 4096-byte pages to MiB. Sizes with fewer than five completions are omitted. The selector independently takes the lowest memory median among GWC, CSR with trimming, and CycSC at each size; its choice may differ from the runtime selector. Both axes are logarithmic, with common ranges across panels.

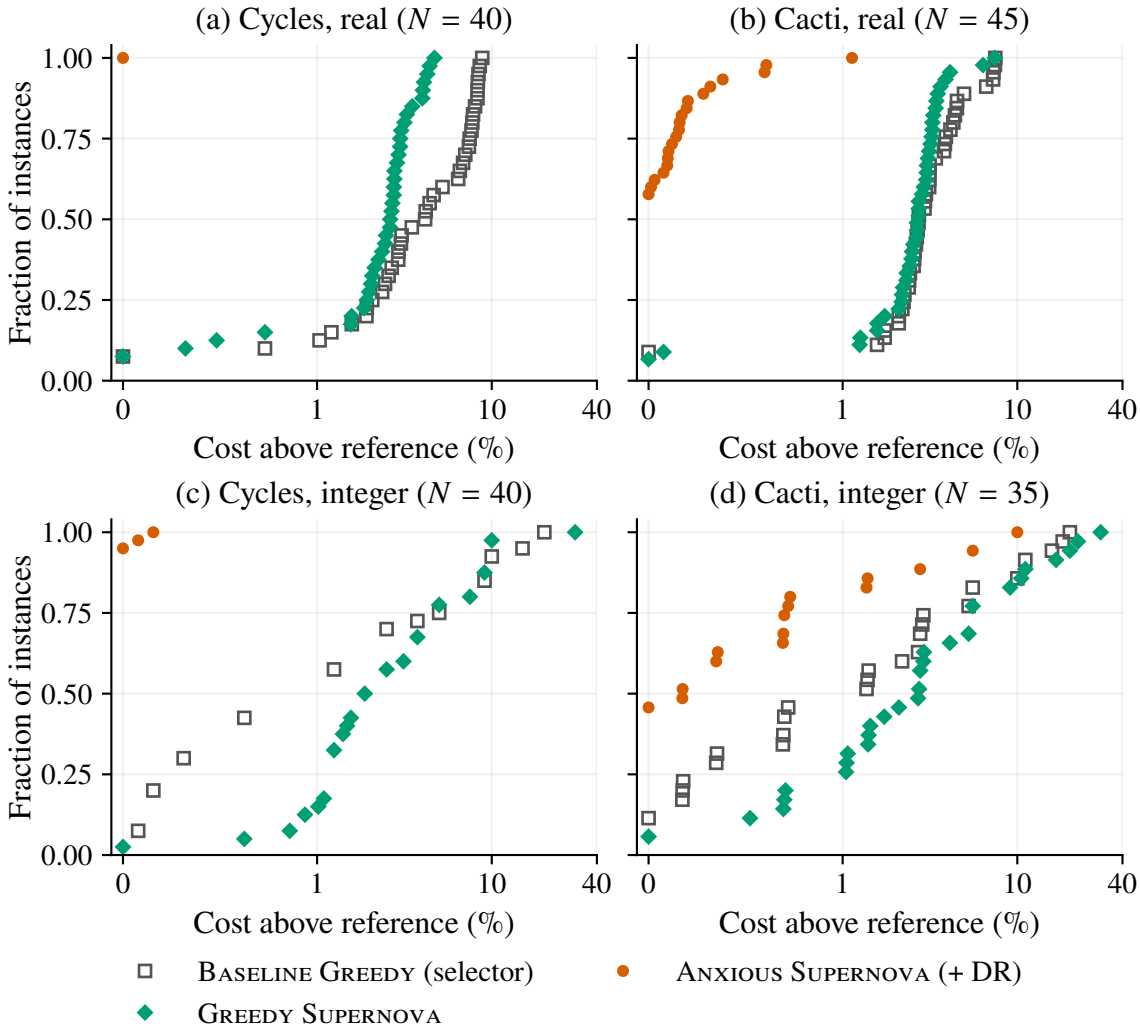


Figure 5.6: Solution quality of Greedy Supernova, Anxious Supernova, and the hypothetical selector. Only Anxious Supernova enables reductions. Each panel uses the same N completed instances for all three methods. The selector takes the lowest reported cost among completed GWC, CSR with trimming, and CycSC runs on each instance. Dots show the fraction within a given percentage above the reference; further left is better. Hollow selector markers keep overlaps visible. The horizontal scale is linear from 0 to 1% and logarithmic thereafter, with common ranges across panels.

5.4 Data Reductions for Heuristics

We investigate the effect of data reductions on heuristic solvers. We use the CycSC configuration as a case study. We test with reductions disabled and enabled. We focus on real costs and include all four graph families. Integer costs are excluded because they yield similar trends.

Figures 5.7 and 5.8 show median total runtime and peak resident memory. They show the cost of enabling reductions. Since GREEDY SET COVER is a fast heuristic, the additional time spent on reductions outweighs the time saved during solving. Even the memory usage is increased, despite the instance size being smaller after reductions.

The most dramatic effect can be observed on cycles. Enabling reductions multiplies total runtime by 29.16, 1.35, 2.00, and 1.89 on cycles, cacti, stars, and trees, respectively. Our PROJECTCROSS algorithm is the most expensive reduction, so this is expected. Peak memory also increases on cycles, stars, and trees. The geometric mean factors range from 1.34 to 1.53. These ratios are taken as averages over all instances.

The runtime and memory penalties are accompanied by an advantage in solution quality. To measure this effect, we pair the two matching runs of each instance and compute the relative cost saving $100(1 - c_{\text{on}}/c_{\text{off}})$, where c_{on} and c_{off} are the reported solution costs with and without reductions. Figure 5.9 shows the distribution of these savings within each graph family, pooling the paired instances across all observed sizes. Each instance receives equal weight, as no relation between instance size and relative saving was found. Vertical lines mark zero saving and the median.

Interestingly, while reductions generally lead to cheaper solutions, the solutions to some instances become more expensive. For real costs, the median savings over the matched instances are 3.19% on cycles, 3.59% on cacti, 3.28% on stars, and 5.89% on trees. All 45 tree instances improve, whereas three cycle instances and one cactus instance become more expensive. For integer costs, no median savings were obtained from reductions, though savings were observed in individual instances.

In conclusion, data reductions generally improve solution quality at the cost of increased runtime and memory usage. However, that is not always the case, and additional experiments are needed to determine what conditions are most likely to yield improvements. Additionally, a comparison of reductions on different heuristics would be of interest. Perhaps some heuristics are more sensitive to reductions than others, and the trade-off between runtime and solution quality may be different.

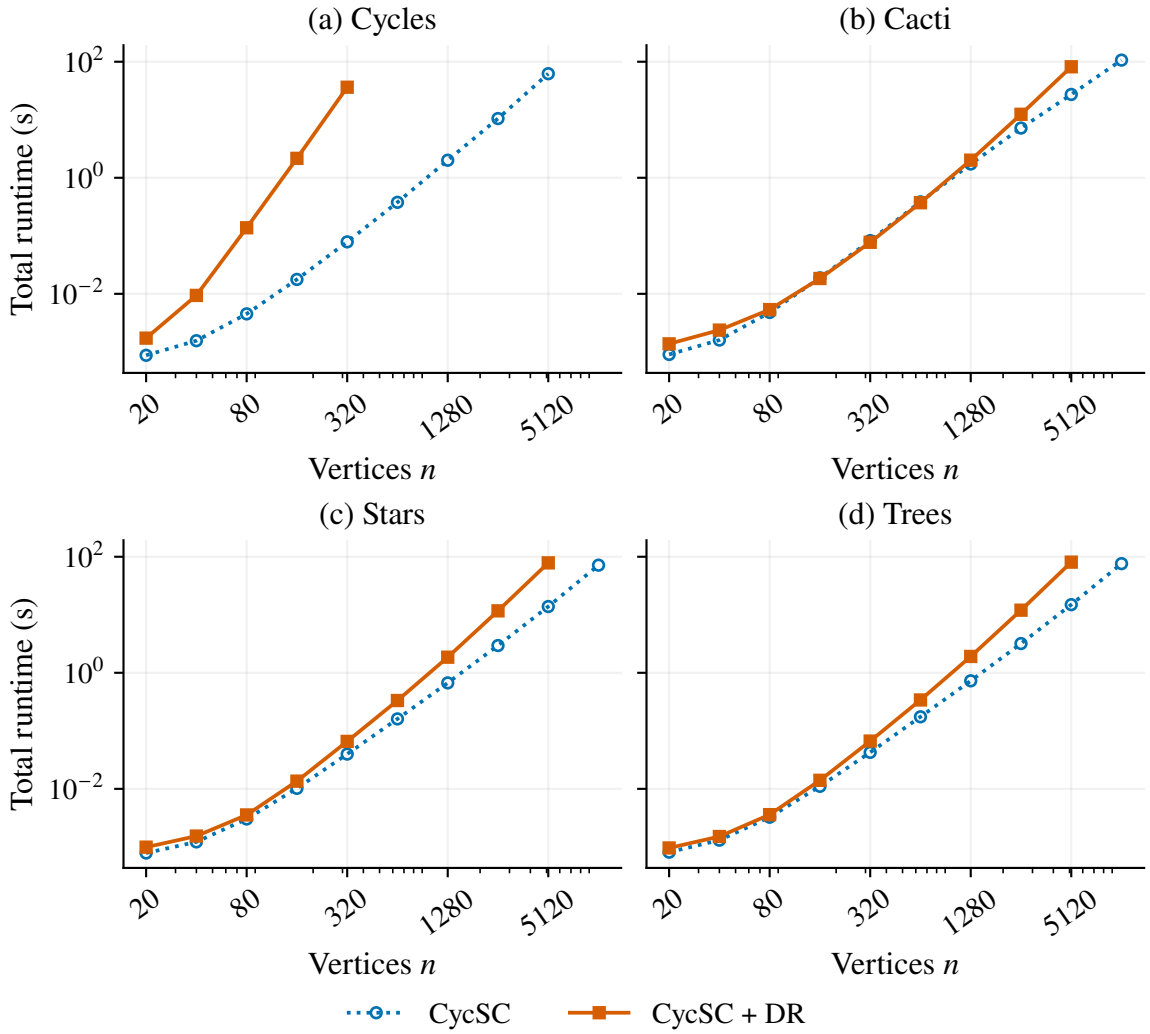


Figure 5.7: Total runtime of CycSC with and without data reductions for real costs. Each point is the median of five completed samples at that size, including all preprocessing and solving stages. Sizes with fewer than five completions are omitted. Both axes are logarithmic, with common ranges across panels.

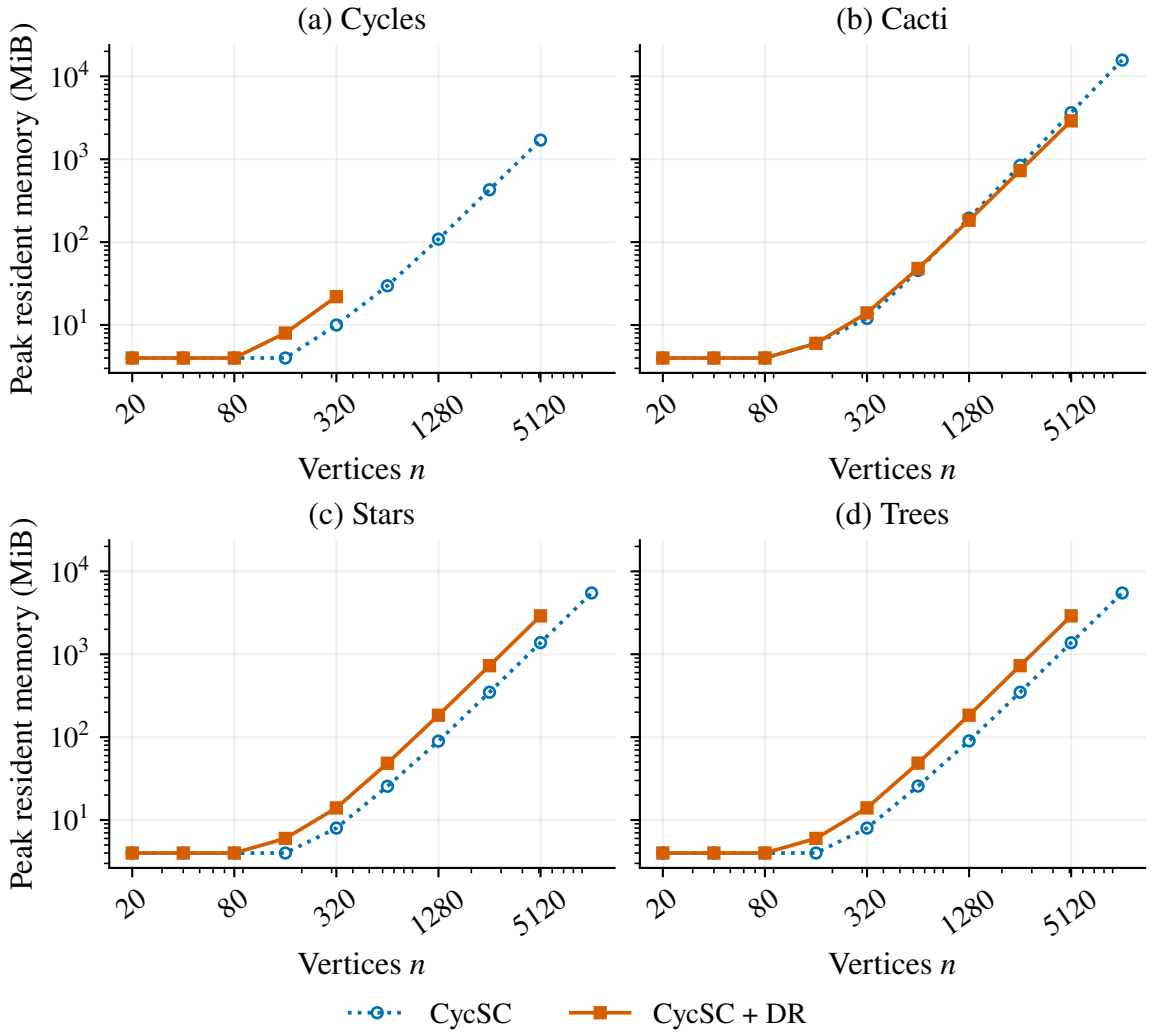


Figure 5.8: Peak resident memory of CycSC with and without data reductions for real costs. Each point is the median of five completed samples, converted from 4096-byte pages to MiB. Sizes with fewer than five completions are omitted. Both axes are logarithmic, with common ranges across panels.

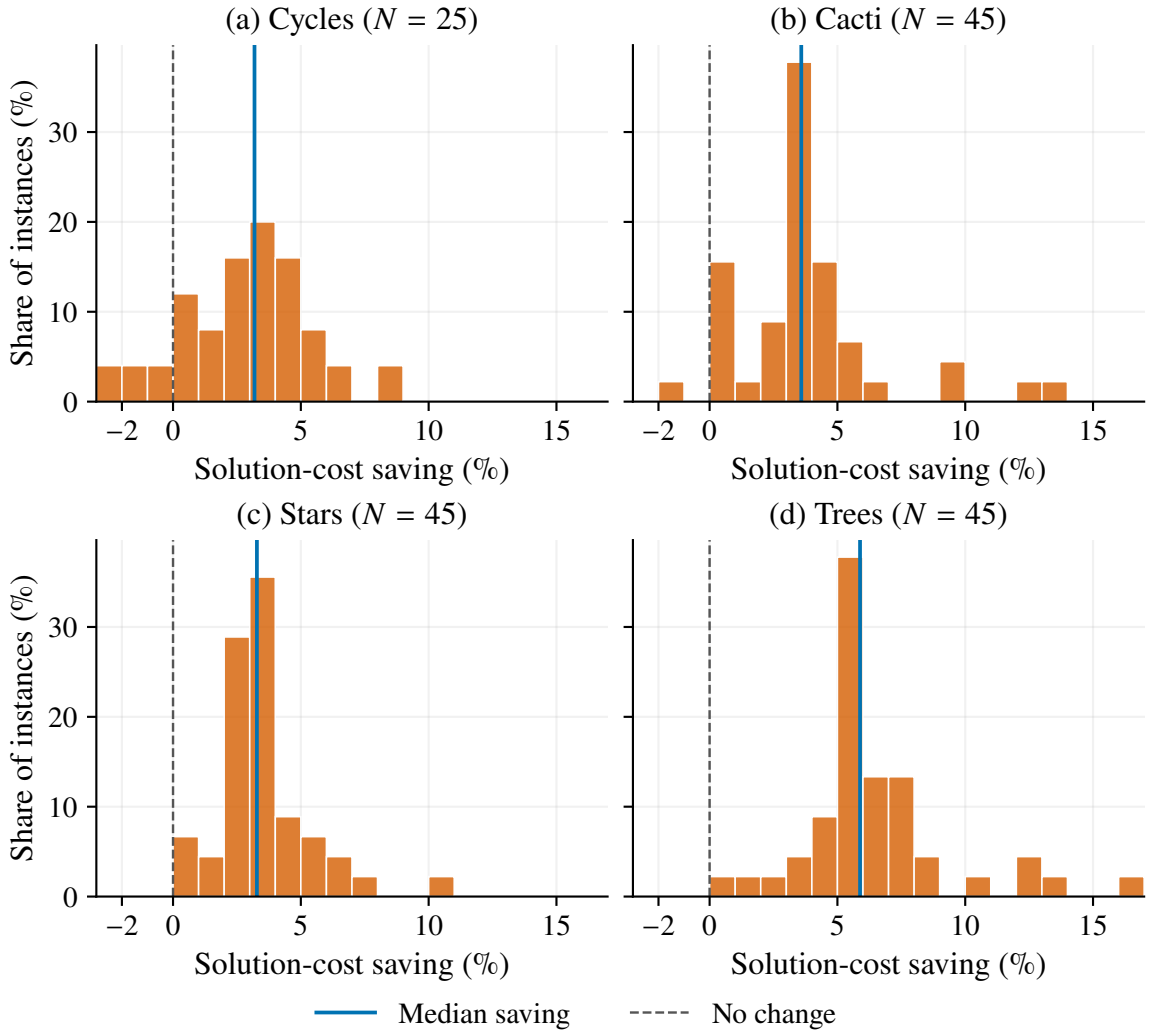


Figure 5.9: Distribution of relative solution-cost savings from enabling reductions in CycSC for real costs, pooled across instance sizes. Each histogram uses common bins of width one percentage point; bar heights give the percentage of paired instances in each bin. Positive savings indicate cheaper solutions with reductions. The solid line marks the median and the dashed line marks zero saving. Only sizes with five completions in both settings are included; N is the number of paired instances. Both axes are linear, with common ranges across panels.

5.5 Exact Solvers

In this section we compare four exact solver configurations: SC-ILP and LAZY SUPERNOVA, each with and without reductions. Since all approaches are exact, we compare them in terms of memory usage and runtime. We do not include the original ILP solver in the comparison because its results are identical to the ones obtained by our SC-ILP, which we will use as a baseline for comparison. LAZY SUPERNOVA only differs from SC-ILP on graphs containing cycles of length at least 3, so we only run it on cycles and cacti.

Data reductions improve both the runtime and the largest solvable instance size of SC-ILP on all four datasets with real costs, as shown in Figure 5.10. Here, speedups compare median total runtimes at the same size, including the reduction work, and the largest solvable size requires all five samples to complete within the resource limits. On cycles at $n = 80$, reductions decrease runtime from 37.78 to 0.154 seconds, a speedup of 244.5, and extend the largest completed size from 80 to 320 vertices, a factor of 4. On cacti at $n = 640$, runtime falls from 65.71 to 0.394 seconds, a speedup of 166.6, while the largest completed size increases eightfold, from 640 to 5120 vertices. The gains are smaller on stars: at $n = 5120$, runtime decreases from 57.45 to 14.85 seconds, a speedup of 3.87, and the largest completed size doubles to 10240 vertices. On trees at $n = 2560$, runtime falls from 66.94 to 3.84 seconds, a speedup of 17.45, and reductions allow four times as many vertices, increasing the largest completed size to 10240. Figure 5.11 shows corresponding decreases in peak resident memory: at these same sizes, reductions lower median memory usage by factors of 41.9 on cycles, 81.3 on cacti, 3.53 on stars, and 10.50 on trees.

LAZY SUPERNOVA provides a further way to avoid the cost of the explicit set cover representation on cycles and cacti. Without reductions, its median runtime on cycles at $n = 80$ is 0.0326 seconds, compared with 37.78 seconds for SC-ILP, making it approximately 1158 times faster. It completes all five samples up to $n = 2560$, or 32 times the largest size of SC-ILP without reductions, while using 29.9 times less memory at the shared size $n = 80$. On cacti at $n = 640$, it reduces runtime from 65.71 to 3.62 seconds, a speedup of 18.18, and uses 8.35 times less memory. This relatively mild improvement relative to that of the reductions can be attributed to the fact that the reduction rules on cacti are exceptionally effective.

The effect of adding reductions to LAZY SUPERNOVA differs sharply between these two datasets. For cycles, reductions pose a dramatic overhead, as PROJECTCROSS, the main reduction algorithm on cycles, is more computationally expensive than the other reductions. The dramatic speedup achieved by LAZY SUPERNOVA can therefore be attributed to the power of its reduced constraint set. It effectively reduces the instance in negligible time by adding a small set of constraints.

The stage breakdown in Figure 5.12 explains why further improvements to the ILP solver alone would have little effect on the reduced real-cost instances. Reductions take an increasingly higher share of the total runtime as instances increase in size. At the largest fully completed SC-ILP sizes, reductions account for an average of 99.9% of total runtime

on cycles ($n = 320$) and 93.5% on cacti ($n = 5120$). On stars and trees at $n = 10240$, their shares are 66.0% and 71.3%, respectively, with input and other overhead accounting for most of the remaining time. ILP solving contributes less than 0.1% in all four cases.

This balance differs for integer costs: solving still accounts for 99.4% on cycles at $n = 80$ and 93.9% on cacti at $n = 640$, while reductions account for 70.7% on stars at $n = 5120$ and 63.1% on trees at $n = 2560$. The reason for this discrepancy can be attributed to the early stopping trick applied to PROJECTCROSS, described in Section 4.2.3. It is very effective for link distributions where the ratio from the smallest to the largest cost is small. That is because in the real distribution there are always links with very small costs that could potentially yield further reductions, no matter how insignificant they might be. These results therefore suggest testing an earlier cutoff for reductions. Stopping the reduction process earlier would leave more work for solving, so the potential saving must be evaluated against the additional runtime and memory of the larger remaining instance.

We have focused on real costs because they show the effects of data reductions most clearly. With integer costs, repeated link costs leave fewer opportunities for domination, so runtime gains from reductions are smaller and largely disappear on stars. LAZY SUPERNOVA retains its advantage over SC-ILP on cycles and cacti, even with reductions enabled, where the real-cost configurations perform almost identically.

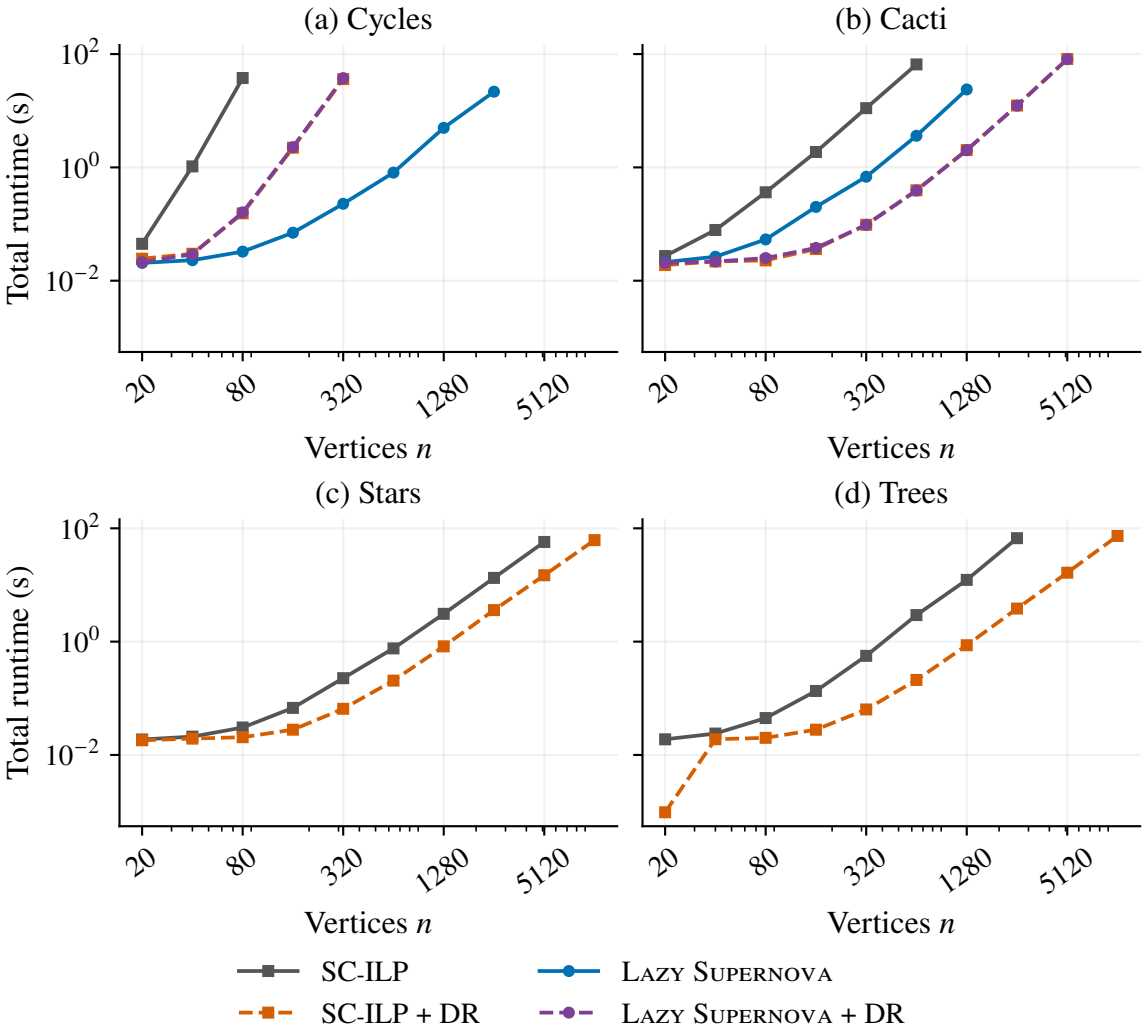


Figure 5.10: Runtime of the exact solvers for link costs drawn uniformly from $(0, 1]$. Each point gives the median total runtime over five completed samples at a tested size, including input reading, preprocessing, and solving. Sizes with fewer than five completions are omitted. Both axes are logarithmic and use common ranges across panels. + DR denotes enabled reductions; Lazy Supernova is shown only for cycles and cacti. The reduced configurations nearly overlap on cycles and cacti.

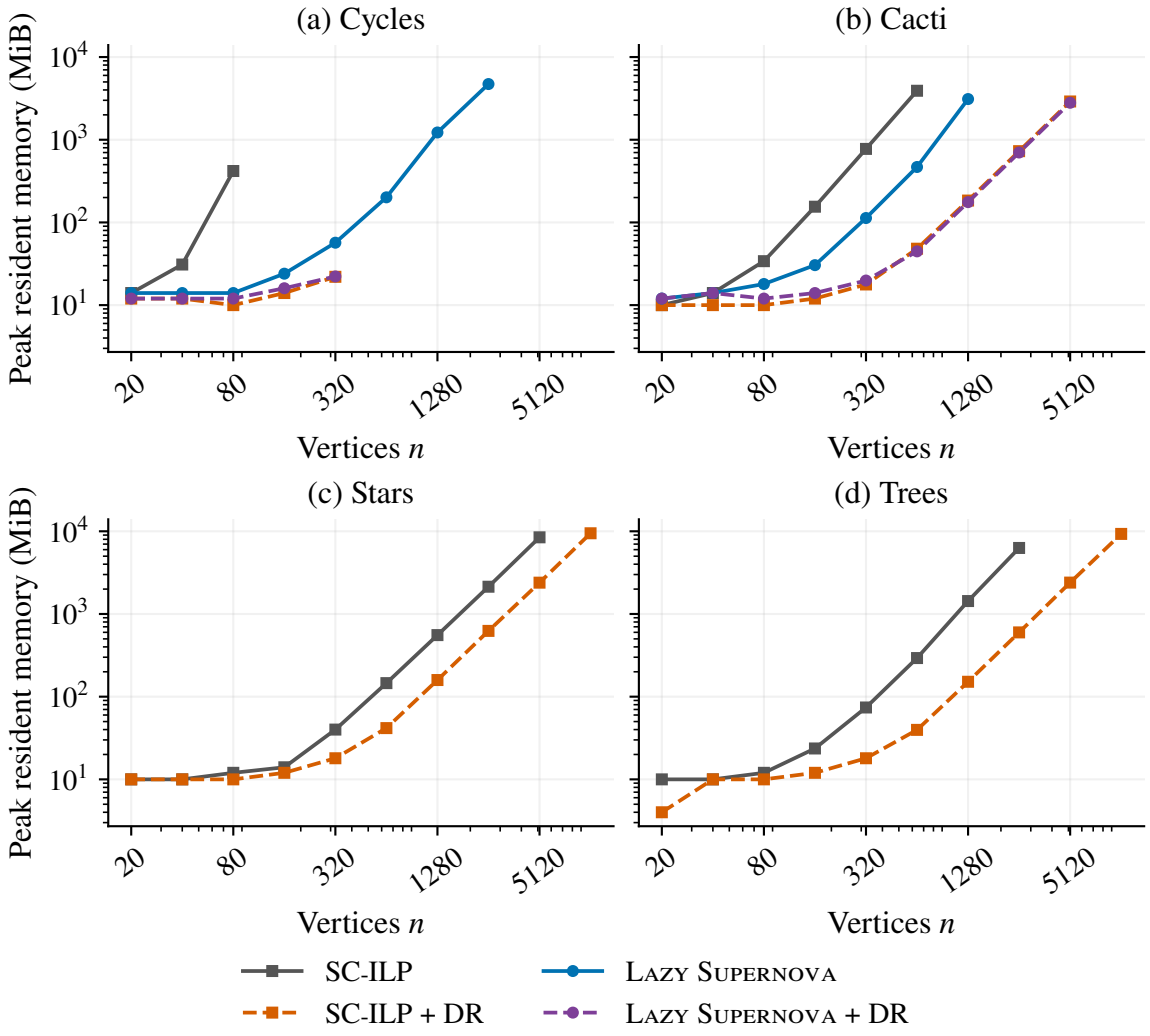


Figure 5.11: Peak resident memory of the exact solvers for link costs drawn uniformly from $(0, 1]$. Each point gives the median peak resident memory over five completed samples at a tested size, converted from 4096-byte pages to MiB. Sizes with fewer than five completions are omitted. Both axes are logarithmic and use common ranges across panels. + DR denotes enabled reductions; Lazy Supernova is shown only for cycles and cacti.

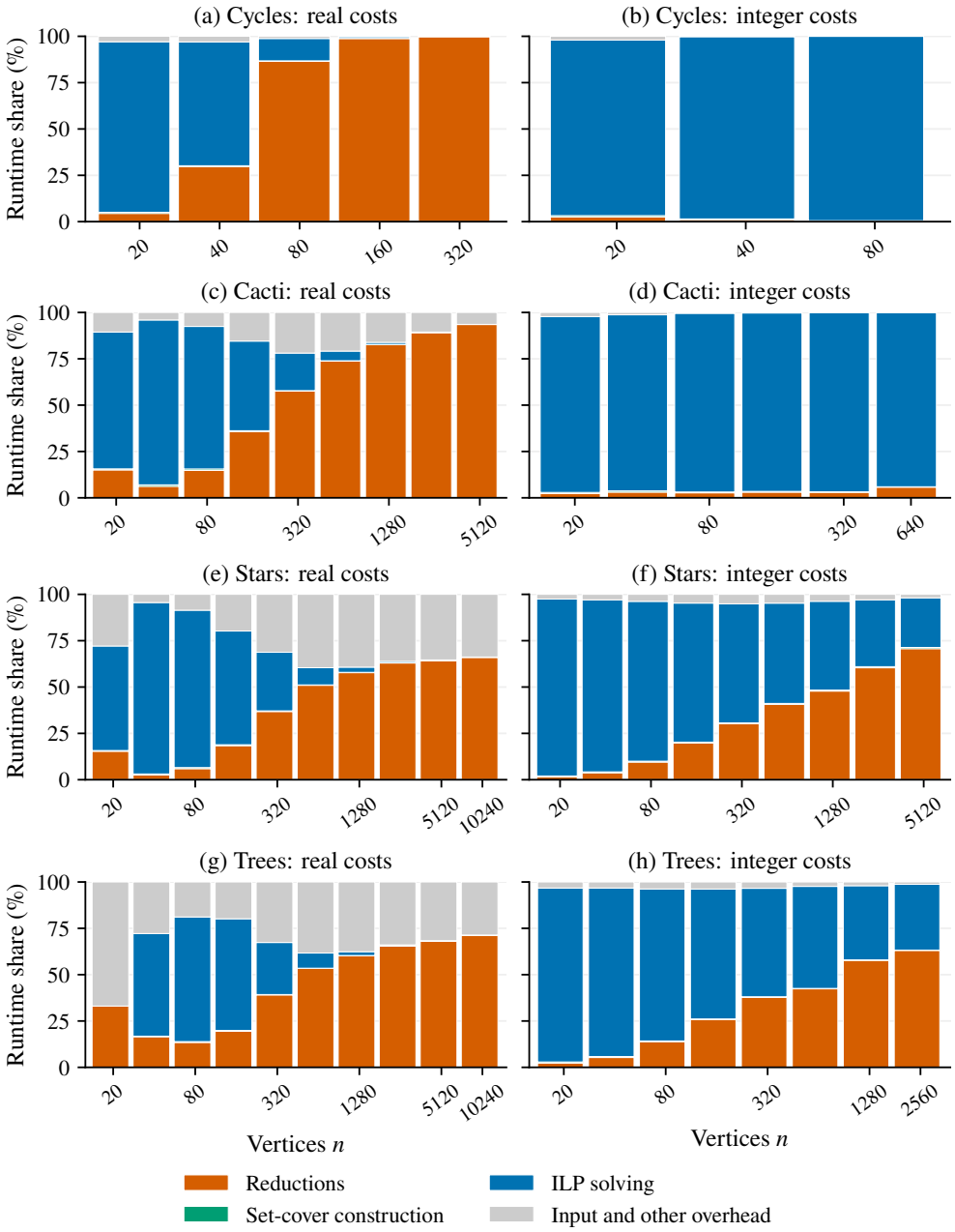


Figure 5.12: Runtime breakdown of SC-ILP with reductions enabled, by graph family and cost distribution. Each bar shows the mean of the per-run stage percentages over five completed samples at a tested size; sizes with fewer than five completions are omitted. Input and other overhead is the total runtime minus the three recorded stages. Construction and solving take zero time when reductions solve the instance completely. The percentage scale is shared, while logarithmic size ranges differ between panels.

5.6 State-of-the-Art Comparison

We compare our heuristics with MSTCONNECT and its local-search variant from Faraj, Großmann, Joos, Möller, and Schulz [6]. The comparison includes three configurations developed in this work: GREEDY SET COVER with CycSC representation, without reductions or trimming, GREEDY SUPERNOVA with trimming and without reductions, and ANXIOUS SUPERNOVA with ILP and reductions, without trimming. These are comparisons of the complete configurations, including their preprocessing costs. Unlike the hypothetical selector used earlier, every curve represents an executable algorithm. We include all four graph families to assess both the benefit of exploiting cycles and the performance on instances without longer cycles. Note, however, that GREEDY SUPERNOVA and ANXIOUS SUPERNOVA become equivalent to their base solvers on stars and trees, namely GREEDY SET COVER with CSR representation and SC-ILP, respectively. The local-search configuration uses depth 4, two trees, caching disabled, and an internal time limit of 270 seconds, within the external 300-second limit. Runs that return a solution after reaching the internal limit remain part of the comparison.

Figures 5.13 and 5.14 show median total runtimes at sizes with five completed samples. On real-cost cycles at $n = 5120$, GREEDY SUPERNOVA takes 9.39 seconds, compared with 26.97 seconds for MSTCONNECT, 62.21 seconds for CycSC, 79.15 seconds for ANXIOUS SUPERNOVA, and 274.61 seconds for MSTCONNECT with local search. On real-cost cacti at the same size, GREEDY SUPERNOVA also has the lowest median runtime among the available configurations, at 16.80 seconds, followed by CycSC at 27.29 seconds, MSTCONNECT at 30.50 seconds, and ANXIOUS SUPERNOVA at 90.15 seconds. MSTCONNECT with local search takes 274.60 seconds. The local-search runtimes near 270 seconds reflect its configured search budget.

Figures 5.15 and 5.16 show that the runtime differences are accompanied by different memory requirements. At $n = 5120$ with real costs, MSTCONNECT with local search uses about 1192–1193 MiB across the four families, compared with 2902–2914 MiB for ANXIOUS SUPERNOVA. The latter nevertheless uses less memory than GREEDY SUPERNOVA on real-cost cacti at this size: 2914 MiB versus 5158 MiB. The ordering changes for integer-cost cacti at $n = 2560$, where ANXIOUS SUPERNOVA uses 5361 MiB, compared with 1135 MiB for GREEDY SUPERNOVA, 842 MiB for CycSC, and 300 MiB for MSTCONNECT with local search. Thus, ANXIOUS SUPERNOVA's memory cost depends strongly on the instance family and cost distribution. This is consistent with the earlier observation that data reductions on cactus and tree graphs are very effective, especially for real costs, so the instances solved by ANXIOUS SUPERNOVA after reductions are much smaller than those solved by the competitors.

For solution quality, we restrict each panel to instances with a reference cost and a completed run from every configuration shown in that panel. Figures 5.17 and 5.18 report empirical cumulative distributions of the percentage cost above that reference. Each instance receives equal weight, independently of its size.

For real costs, ANXIOUS SUPERNOVA yields the best results in all four families. MST-CONNECT with local search has median gaps of 1.93% on cycles, 1.94% on cacti, 2.37% on stars, and 1.26% on trees. These are lower than the corresponding GREEDY SUPERNOVA medians except on stars, where GREEDY SUPERNOVA achieves 2.35%. CycSC has larger median gaps, ranging from 7.33% to 8.61%. Local search lowers the standalone MST-CONNECT median gap from 4.27% to 1.93% on cycles, from 3.86% to 1.94% on cacti, from 3.88% to 2.37% on stars, and from 2.74% to 1.26% on trees.

The integer-cost results give a different comparison with the existing heuristics. MST-CONNECT with local search has median gaps of 45.63% on cycles, 37.50% on cacti, 43.44% on stars, and 74.09% on trees. Standalone MSTCONNECT has median gaps of 60.94% on cycles, 58.33% on cacti, 60.00% on stars, and 84.18% on trees. In contrast, the median gaps of CycSC and GREEDY SUPERNOVA are at most 2.74% across these families. ANXIOUS SUPERNOVA again has the lowest median gaps: 0% on cycles, stars, and trees, and 0.18% on cacti. Thus, in these experiments, the extra work of ANXIOUS SUPERNOVA yields solutions close to the reference costs, while CycSC and GREEDY SUPERNOVA offer substantially shorter runtimes and, particularly for integer costs, remain competitive in solution quality.

From these results, we conclude that GREEDY SUPERNOVA gives a good trade-off between runtime and solution quality, in comparison with the other solvers. It is the fastest of the available configurations on all datasets and link distributions, while achieving competitive solution quality. However, it does not universally outperform the other configurations on every instance, in particular with regard to memory. MSTCONNECT with local search is also capable of providing better solutions on many of the instances, and since it iteratively searches for better solutions, it is possible to trade runtime for solution quality. In that case, a comparison of solution costs achieved by the local search over time would be of interest.

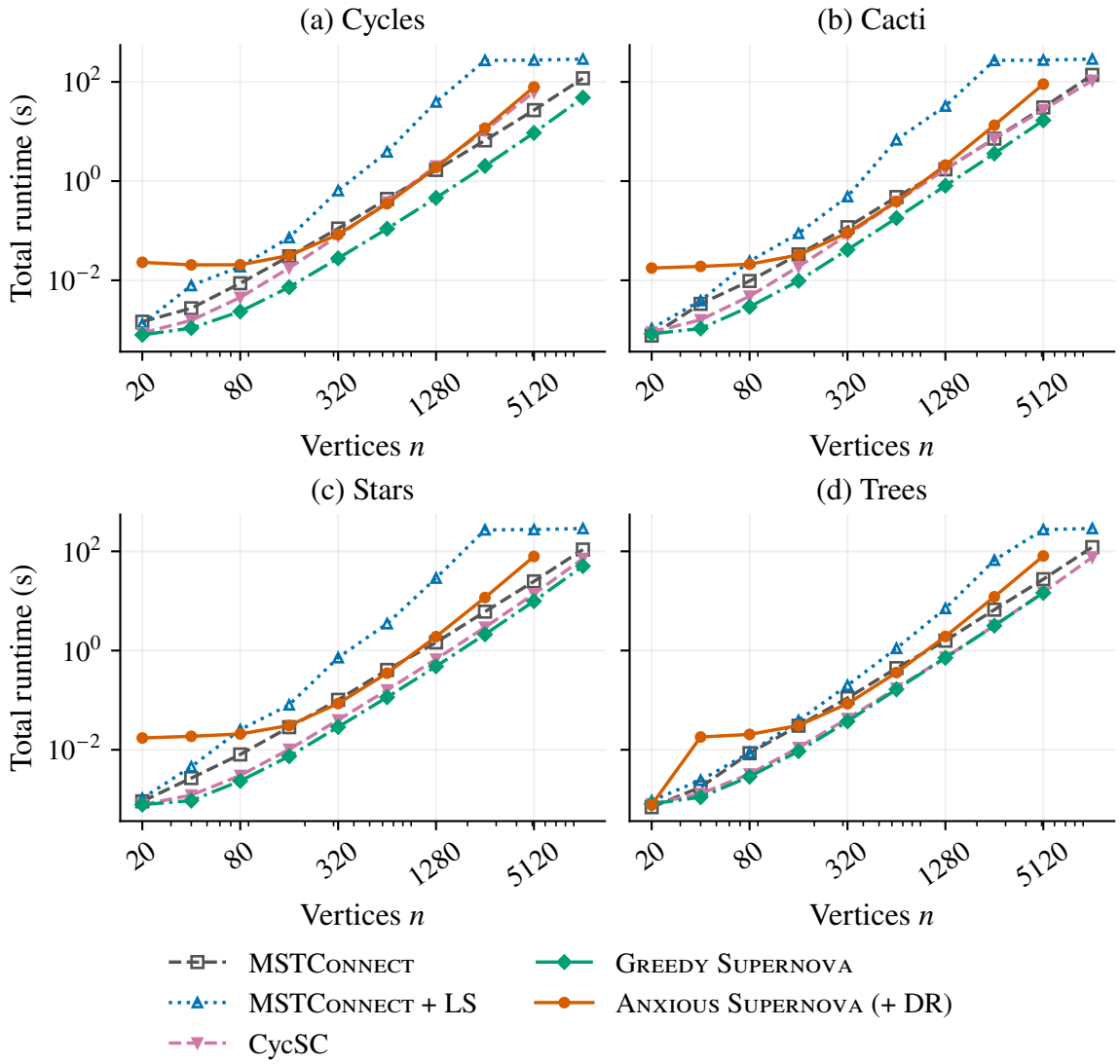


Figure 5.13: Total runtime of the heuristic configurations on real-cost instances. Each point is the median of five completed samples, including input, preprocessing, and all solving stages. Sizes with fewer than five completions are omitted. Only Anxious Supernova enables reductions; MSTConnect + LS has a 270-second internal search budget. Both axes are logarithmic, with common ranges across panels.

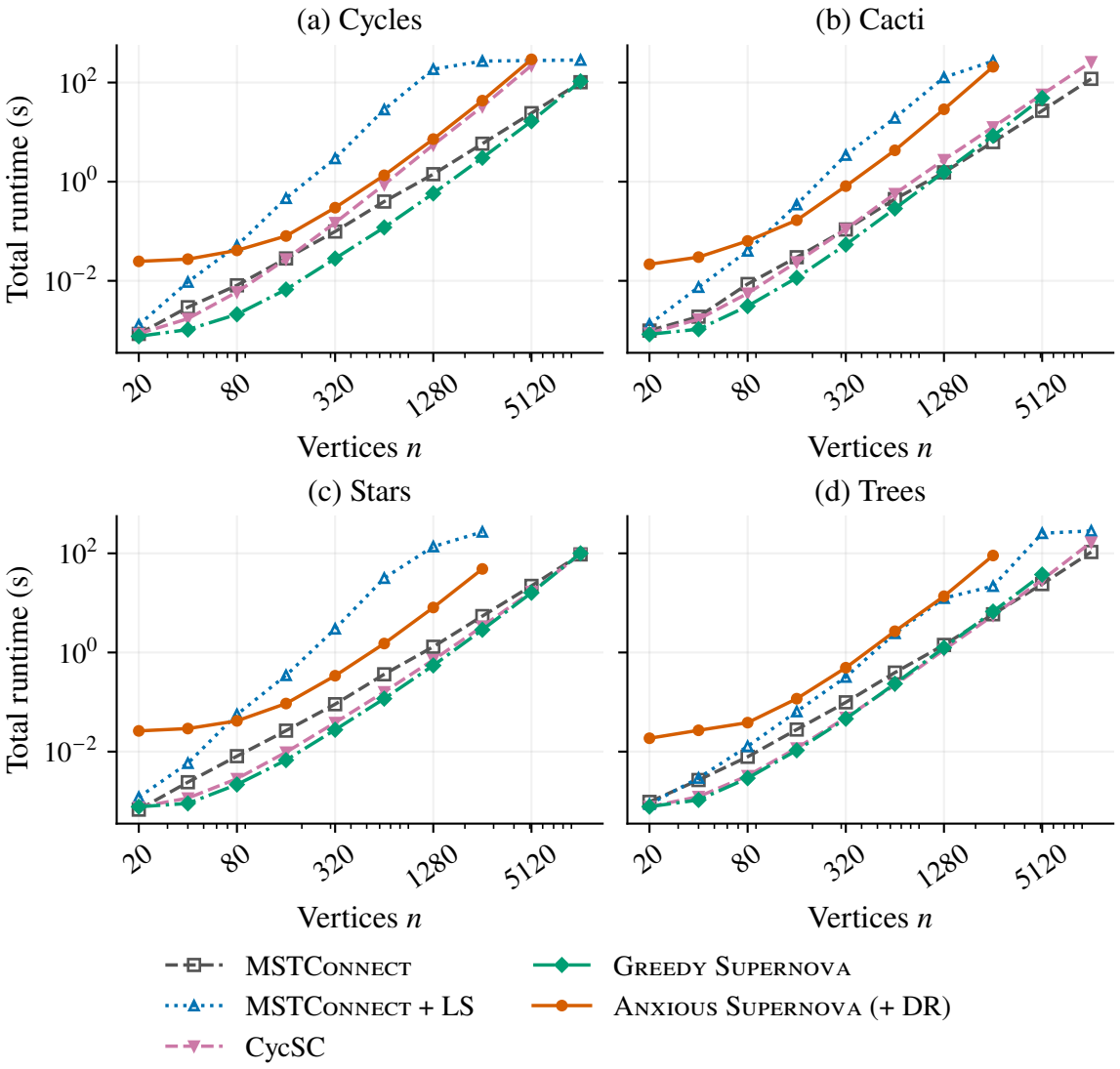


Figure 5.14: Total runtime of the heuristic configurations on integer-cost instances, using the conventions of Figure 5.13. Missing points indicate fewer than five completions or unavailable data.

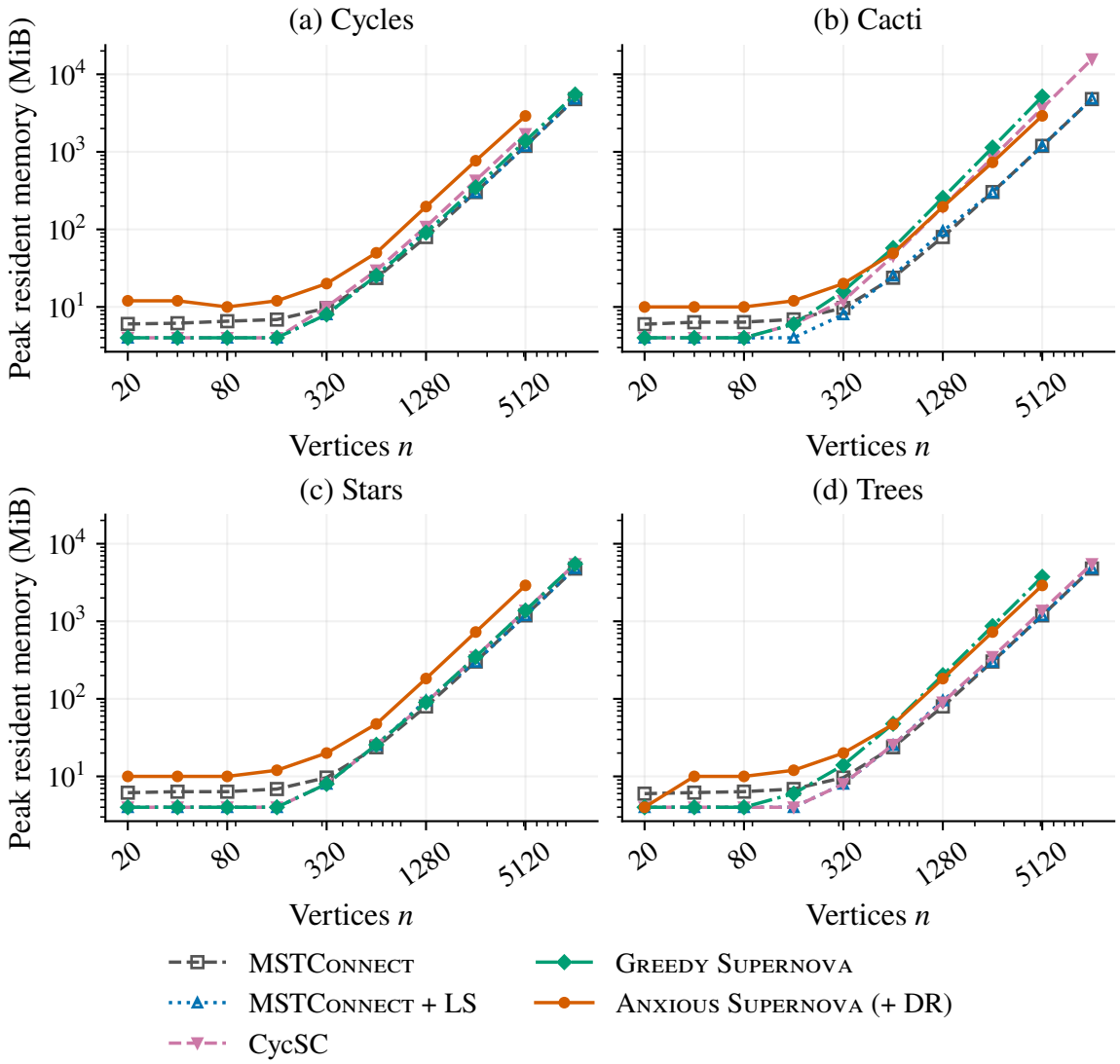


Figure 5.15: Peak resident memory of the heuristic configurations on real-cost instances. Each point is the median of five completed samples, converted from 4096-byte pages to MiB. Only Anxious Supernova enables reductions. Both axes are logarithmic, with common ranges across panels.

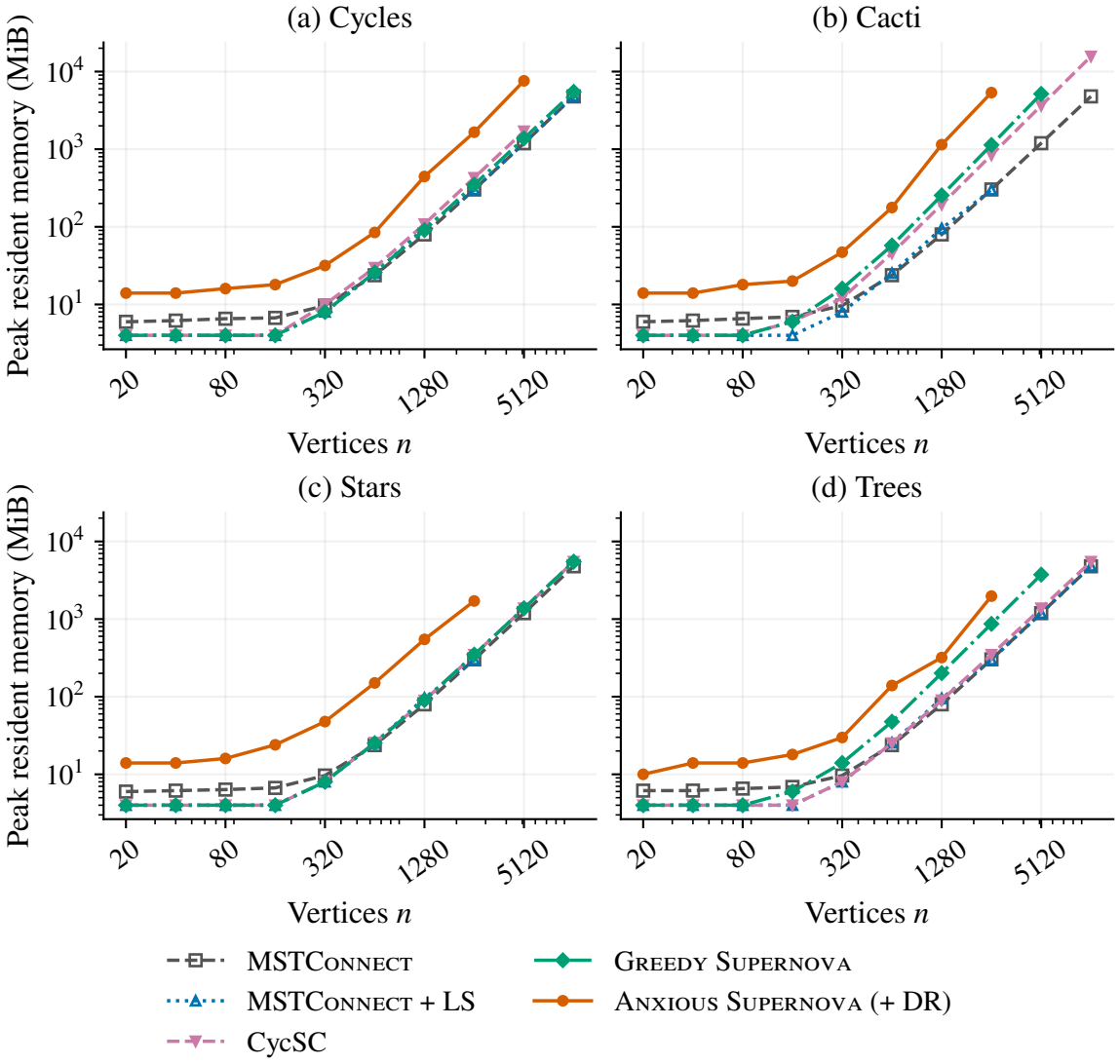


Figure 5.16: Peak resident memory of the heuristic configurations on integer-cost instances, using the conventions of Figure 5.15. Sizes with fewer than five completions are omitted.

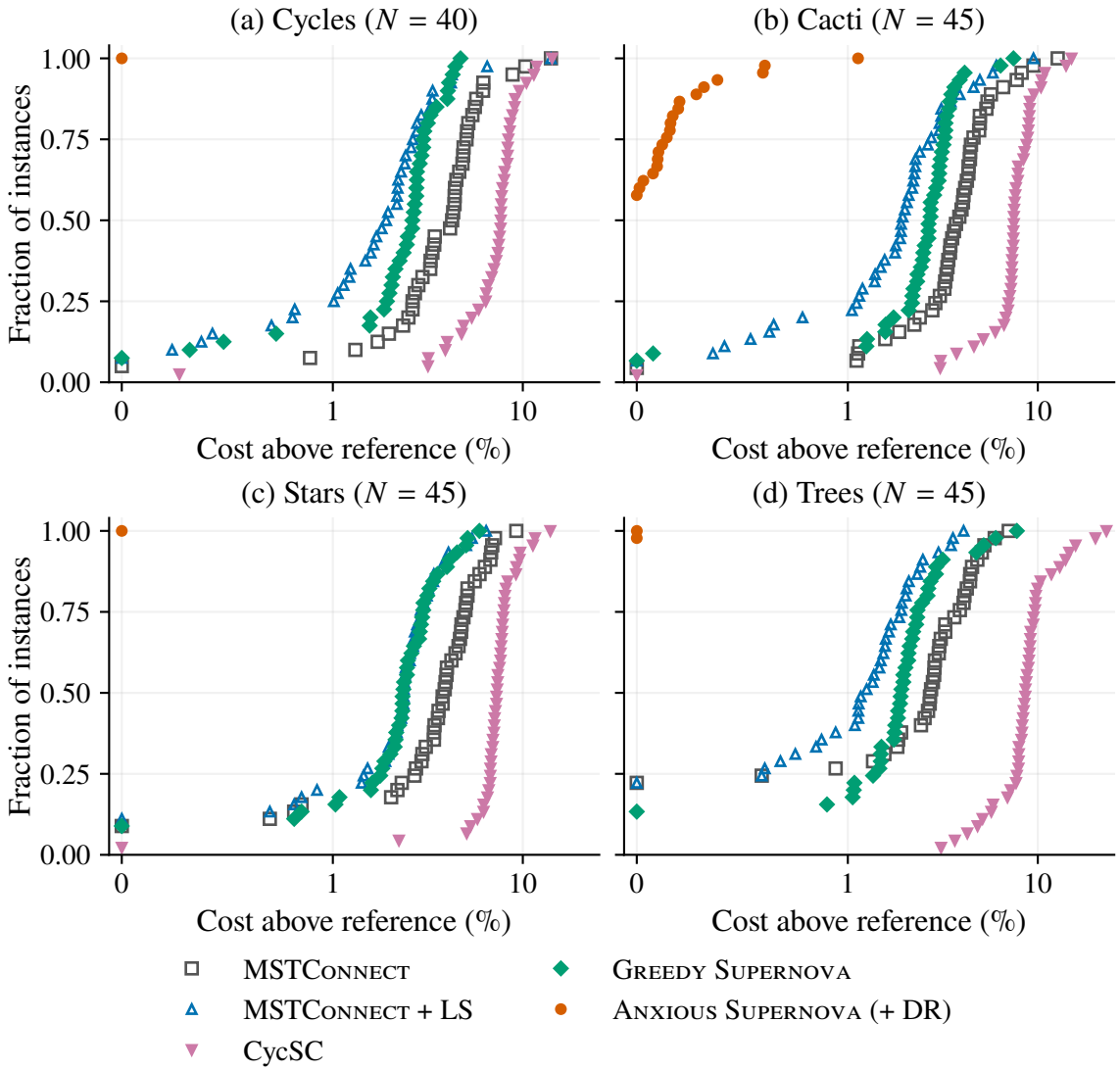


Figure 5.17: Solution quality of the heuristic configurations on real-cost instances. Each panel uses the same N reference instances for all five methods. Dots give the fraction within a given percentage above the reference; further left is better. Hollow MSTConnect markers keep overlaps visible. The horizontal scale is linear from 0 to 1% and logarithmic thereafter, with common ranges across panels.

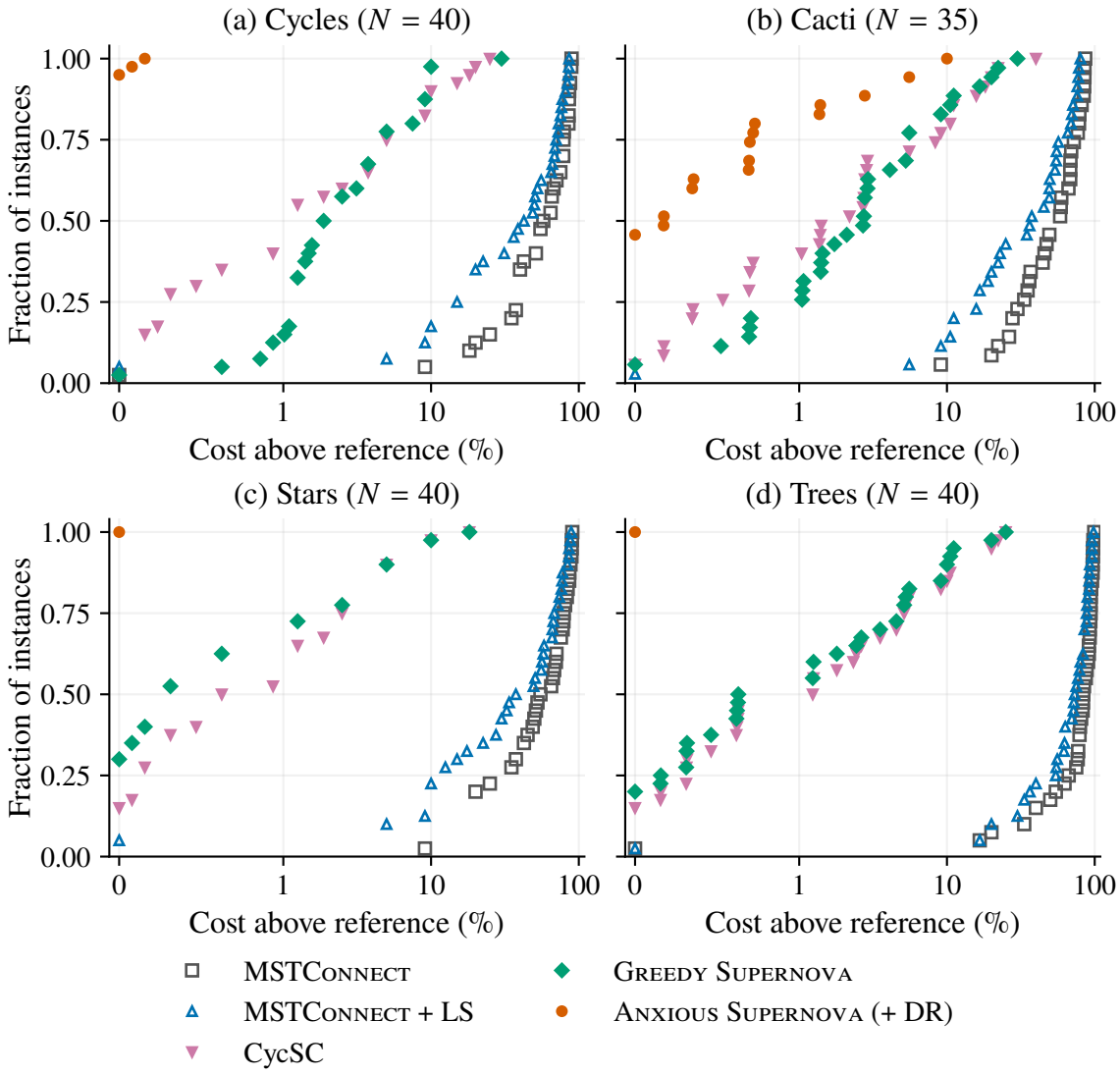


Figure 5.18: Solution quality of the heuristic configurations on integer-cost instances, using the conventions of Figure 5.17. Each panel uses the same N reference instances for all five methods.

CHAPTER 6

Discussion

In this chapter, we review and summarize our contributions, highlight the limitations of our work, and discuss directions for future research.

6.1 Conclusion

Viewing WCA as a structured WEIGHTED SET COVER problem provides a common basis for reductions, compact data structures, and exact and heuristic solvers. The cactus representation is central to this approach, as it exposes the minimum-cut constraints while allowing them to be processed without an explicit set cover instance. The experimental evaluation determines how far these structural advantages translate into runtime, memory, and solution quality improvements in practice.

The scope of this thesis is broad, so it is easy to lose sight of the overall picture. Our contributions are manifold. In this section, we summarize the main algorithmic contributions and highlight their most important experimental results.

We have developed a hybrid set cover representation, called CycSC, of the WCA problem that seeks to mitigate the deleterious effects of large cycles in the direct WSC reduction. It enables a decreased memory footprint and accelerates GREEDY SET COVER on cycles and cacti. On cycles with real-cost links at $n = 320$, CycSC requires 10 MiB of peak memory, compared with approximately 13 GiB for the explicit CSR representation. This representation benefits computational efficiency especially on cactus instances. CycSC completes execution for instances with up to 10240 vertices, four times the largest fully completed size of GWC.

We have implemented five reduction rules. The reduction rules allow us to remove links from the candidate set that can be replaced by links of equal or lower cost, to contract parts of the instance that are guaranteed to be superfluous, and to fix links that are guaranteed to be part of any feasible solution. Together, these reductions can substantially reduce the running time and memory footprint of exact solvers. They can also improve the solution

quality of heuristic solvers by removing links that would otherwise be selected in a suboptimal solution. These effects can most clearly be seen on cacti. On instances with 640 vertices, reductions lower the direct ILP's median runtime and peak memory by factors of 166.6 and 81.3, respectively. For GREEDY SET COVER with the CycSC representation, the reductions lower solution costs by medians of 3.19% to 5.89% across the four real-cost graph families, although preprocessing increases total runtime.

Finally, we combine all of these ideas into a single framework, SUPERNOVA. Although the framework is not yet implemented in its full generality, our restricted implementation can outperform state-of-the-art exact and heuristic solvers. On real-cost cycles at $n = 80$, LAZY SUPERNOVA without reductions is approximately 1158 times faster than the state-of-the-art exact ILP and extends the largest fully completed size from 80 to 2560 vertices, a factor of 32. GREEDY SUPERNOVA is up to 13.33 times faster than the best greedy algorithm.

Nevertheless, the empirical conclusions are necessarily limited to the graph families, link sets and cost distributions specified in Chapter 5. In particular, experiments on complete link sets with synthetic costs do not by themselves establish the same behavior for sparse candidate sets or application-specific costs. Link costs sampled from a uniform distribution may not reflect the cost structure of real-world instances. For instance, more distant vertices are likely to be connected by more expensive links, and the uniform distribution does not capture this correlation.

6.2 Future Work

The results suggest several directions for further research. So far, we have explored SUPERNOVA primarily with restrictions to the block-tree. The framework also permits other choices of minimum-cut constraints, which could be adapted to the structure of the cactus. This is particularly relevant to the synthetic cactus instances. Chapter 5 shows that committing to a block-tree solution can lead to poor final solutions on these instances, even when the subproblems are solved exactly. Choosing constraints that better capture the remaining cycle cuts may improve these initial decisions. Another direction is to run several combinations of solvers, reductions, and constraint choices in parallel. Such a portfolio could retain the cheapest feasible solution and the strongest lower bound valid for the original instance, combining the strengths of different configurations.

The link domination rules offer two complementary directions for improvement. Stronger rules could detect dominations that the current algorithms miss and thereby reduce the instance further. Conversely, cheaper rules that detect only a subset of these dominations may be preferable when the data reductions contribute substantially to the total runtime. In particular, fast heuristic solvers may benefit more from a few inexpensive reductions than from a more thorough search for dominated links.

The coverage rules underlying link domination could also guide local search. Starting from a feasible solution, we could search for a small set of candidate links whose combined

coverage contains the coverage of several selected links. Replacing those selected links preserves feasibility and improves the solution whenever the cost of the newly added links is smaller than the cost of the removed links. The structural rules developed in this thesis could help identify such exchanges without explicitly comparing all covered minimum cuts.

A further question is when to store cycle coverage explicitly and when to use the compact cycle representation. For short cycles, storing the link-to-cut incidences in CSR may be faster than deriving coverage from cycle projections. For longer cycles, the compact representation avoids storing many incidences. A hybrid representation could choose between the two for each cycle. Experiments could identify a useful cycle-length threshold and determine how it depends on the number and distribution of candidate links.

Any cactus can be expressed by a cycle and a set of contractions. This means we can represent the entire WCA instance and its solved state in a single cycle. CycSC from Section 4.1 could potentially be more efficient this way. This also allows SUPERNOVA to be applied to the graph as a whole, rather than targeting individual cycles.

There is a very simple and fast approach for solving instances with constant weights. One can simply follow a Hamiltonian cycle on the cactus and choose links connecting opposing cactus leaves. Perturbations of this approach could yield powerful solutions in case the ratio of the cheapest to the most expensive links is small.

Finally, faster removal of redundant links could improve the competitiveness of heuristic solvers on cycles and cacti. This is especially true for GREEDY CHEAPEST and solvers that yield solutions with many redundant links. The proposed approach is to build the link intersection graph of the selected links, as described in Section 4.2. To preserve coverage of all minimum cuts, the remaining links must touch every cactus leaf, while maintaining the connectivity of the link intersection graph. A removal test can therefore check that the link is not an articulation vertex of the link intersection graph and that neither endpoint is the last link incident to a cactus leaf. This graph contains only the selected links, which may make these checks cheaper than testing cut coverage explicitly.

Zusammenfassung

Konnektivitätsaugmentation verstärkt ein Netzwerk durch das Hinzufügen von Kanten zu minimalen Kosten. In dieser Arbeit untersuchen wir das Problem der WEIGHTED CONNECTIVITY AUGMENTATION (WCA): Gegeben sind ein Graph sowie eine Menge potenzieller zusätzlicher Kanten mit positiven Kosten; gesucht ist eine kostenminimale Teilmenge, deren Hinzufügen die Kantenzusammenhangszahl des Graphen um eins erhöht. Bereits eingeschränkte Varianten von WCA sind NP-vollständig. Umfangreiche theoretische Arbeiten haben zu immer stärkeren Approximationsgarantien geführt und schließlich eine $(1{,}5 + \epsilon)$-Approximation für das allgemeine Problem hervorgebracht. Diese Algorithmen sind jedoch nicht praxistauglich, und es wurde wiederholt gezeigt, dass deutlich einfachere Heuristiken ihnen überlegen sind. Daher betrachten wir WCA aus einer Algorithm-Engineering-Perspektive und entwickeln effiziente exakte und heuristische Algorithmen für praktische Instanzen. Unser Ansatz formuliert WCA als ein WEIGHTED SET COVER-Problem über den Schnitten minimalen Gewichts des Eingabegraphen und nutzt die Struktur des resultierenden Mengensystems systematisch aus. Auf dieser Grundlage entwickeln wir kompakte Datenstrukturen, problemspezifische Reduktionsregeln, optimierte Greedy-Algorithmen sowie spezialisierte exakte und heuristische Löser. Diese Techniken fassen wir in SUPERNOVA zusammen, einem neuen algorithmischen Framework für Konnektivitätsaugmentation. Eine umfangreiche experimentelle Evaluation auf synthetischen Graphen zeigt, dass SUPERNOVA bestehende State-of-the-Art-Verfahren übertreffen kann. Unsere exakten Algorithmen lösen deutlich größere Instanzen optimal als das direkte ILP-Vergleichsverfahren, während unsere heuristischen Algorithmen gegenüber bestehenden praktischen Verfahren Verbesserungen der Lösungsqualität oder der Laufzeit bieten. Unser exakter Löser LAZY SUPERNOVA ist auf Zyklen mit 80 Knoten ungefähr 1158-mal schneller als das direkte ILP ohne Reduktionen. Dies ist die größte Instanzgröße, die der beste bestehende Algorithmus bewältigen kann. Unter denselben Ressourcenbeschränkungen kann der Löser auch Instanzen mit bis zu 2560 Knoten lösen. Auf generierten Kaktusgraphen beschleunigen unsere Datenreduktionen den exakten State-of-the-Art-Löser um einen Faktor von bis zu $166{,}6$ und verachtfachen die größte Instanzgröße, für die sämtliche Versuche erfolgreich abgeschlossen werden.

Bibliography

[1] David Adjiashvili. Beating approximation factor two for weighted tree augmentation with bounded costs. *ACM Transactions on Algorithms*, 15(2):19:1–19:26, 2019.

[2] Jarosław Byrka, Fabrizio Grandoni, and Afrouz Jabal Ameli. Breaching the 2-approximation barrier for connectivity augmentation: A reduction to Steiner tree. *SIAM Journal on Computing*, 52(3):718–739, 2023.

[3] Federica Cecchetto, Vera Traub, and Rico Zenklusen. Bridging the gap between tree and connectivity augmentation: Unified and stronger approaches. *SIAM Journal on Computing*, 55(4):STOC21–26–STOC21–103, 2026.

[4] V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[5] Kapali P. Eswaran and R. Endre Tarjan. Augmentation problems. *SIAM Journal on Computing*, 5(4):653–665, 1976.

[6] Marcelo Fonseca Faraj, Ernestine Großmann, Felix Joos, Thomas Möller, and Christian Schulz. Engineering weighted connectivity augmentation algorithms. In *22nd International Symposium on Experimental Algorithms (SEA 2024)*, volume 301 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.

[7] Samuel Fiorini, Martin Groß, Jochen Könemann, and Laura Sanità. Approximating weighted tree augmentation via Chvátal–Gomory cuts. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 817–831. Society for Industrial and Applied Mathematics, 2018.

[8] Greg N. Frederickson and Joseph JáJá. Approximation algorithms for several graph augmentation problems. *SIAM Journal on Computing*, 10(2):270–283, 1981.

[9] Monika Henzinger, Alexander Noe, Christian Schulz, and Darren Strash. Finding all global minimum cuts in practice. In *28th Annual European Symposium on Algorithms (ESA 2020)*, volume 173 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 59:1–59:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

[10] Samir Khuller and Ramakrishna Thurimella. Approximation algorithms for graph augmentation. *Journal of Algorithms*, 14(2):214–225, 1993.

[11] Guy Kortsarz. A $4/3$ ratio approximation algorithm for the tree augmentation problem by deferred local-ratio and climbing. arXiv preprint, 2026. Version 2, 5 March 2026.

[12] Hiroshi Nagamochi, Yoshitaka Nakao, and Toshihide Ibaraki. A fast algorithm for cactus representations of minimum cuts. *Japan Journal of Industrial and Applied Mathematics*, 17(2):245–264, 2000.

[13] Dalit Naor, Dan Gusfield, and Charles Martel. A fast algorithm for optimally increasing the edge connectivity. *SIAM Journal on Computing*, 26(4):1139–1165, 1997.

[14] Petr Slavík. A tight analysis of the greedy algorithm for set cover. *Journal of Algorithms*, 25(2):237–254, 1997.

[15] Vera Traub and Rico Zenklusen. A $(1.5 + \epsilon)$-approximation algorithm for weighted connectivity augmentation. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, pages 1820–1833. Association for Computing Machinery, 2023.

[16] Vera Traub and Rico Zenklusen. Better-than-2 approximations for weighted tree augmentation and applications to Steiner tree. *Journal of the ACM*, 72(2):16:1–16:40, 2025.

[17] Toshimasa Watanabe, Toshiya Mashima, and Satoshi Taoka. The k-edge-connectivity augmentation problem of weighted graphs. In *Algorithms and Computation (ISAAC 1992)*, volume 650 of *Lecture Notes in Computer Science*, pages 31–40. Springer, 1992.