PROVABLY POWERFUL PARAMETERIZED PREPROCESSING TOWARDS PRACTICE

by

Yosuke Mizutani

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

Kahlert School of Computing

The University of Utah

December 2024

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Yosuke Mizutani**
has been approved by the following supervisory committee members:

Blair D. Sullivan ,	Chair(s)	**11 / 09 / 2024** Date Approved
Aditya Bhaskara ,	Member	**11 / 08 / 2024** Date Approved
Jeffrey M. Phillips ,	Member	**11 / 09 / 2024** Date Approved
Travis Bennett Martin ,	Member	**11 / 09 / 2024** Date Approved
Pål Grønås Drange ,	Member	**11 / 11 / 2024** Date Approved

by **Mary W. Hall** , Chair/Dean of
the Department/College/School of **Computing**
and by **Darryl P. Butt** , Dean of The Graduate School.

ABSTRACT

Designing efficient algorithms and studying computational complexity under specific conditions have long been the heart of computer science. While it seems too much to hope $\mathsf{P} = \mathsf{NP}$, we have discovered that many NP-hard problems can be solved in time, which is polynomial in the input size (n) but exponential in some "parameter" (k). This approach produces "parameterized" algorithms with running time $f(k) \cdot \text{poly}(n)$ for some computable function f and provides several novel perspectives for characterizing and solving NP-hard problems. My work spans several topics in this area, including preprocessing, structural parameters, and heuristics, and it is distinguished by a focus on practicality including implementation and verification on real-world datasets.

This dissertation includes edited versions of six papers, which have advanced the theory of parameterized algorithms, each meticulously designed to impact real-world applications. The first two papers give new structured insights that allow more effective preprocessing for some given parameter. The SIAM Conference on Applied and Computational Discrete Algorithms (ACDA) '23 paper introduced an improved kernel for EXACT WEIGHTED CLIQUE DECOMPOSITION, and tested with biomedical datasets to identify groups of genes that consistently coact. In the International Symposium on Parameterized and Exact Computation (IPEC) '24 paper, we showed that we can efficiently find a structure that contains part of an optimal solution to ODD CYCLE TRANSVERSAL.

The next two papers concern structural graph parameters. In the IPEC '22 paper, we resolved open questions on parameterized complexity with respect to modular-width and clique-width for problems that occur in sociology and biology, including DENSEST k-SUBGRAPH. A relatively new parameter twin-width, unlike modular-width or clique-width, characterizes many real-world graphs and thus has gained a lot of attention. We designed and implemented a solver to determine the twin-width of a given graph, which won the PACE Challenge and the associated PACE Theory Award in 2023.

The last two papers study a specific problem, GRAPH INSPECTION, arising in robotics.

The International Workshop on the Algorithmic Foundations of Robotics (WAFR) '24 paper introduces parameterized algorithms along with a framework to enable scaling and establishes their effectiveness on simulated inspection tasks. Finally, we have a preprint that introduces an application of arithmetic circuits and randomized algorithms, which requires only polynomial-space.

To summarize, we present several new parameterized algorithms and carefully engineered implementations that improve the state of the art in theory and practice.

CONTENTS

LIST OF TABLES

ACKNOWLEDGEMENTS

I thank Dr. Blair D. Sullivan for her close supervision and regular support that made my life easier. She gave me ample opportunities for my unforgettable travels to many cities around the world during my happy PhD years. I am also fortunate to have the strongest supervisory committee members. I learned from Dr. Aditya Bhaskara the joy of teaching when I was a teaching assistant. Dr. Jeffrey M. Phillips always asked me inspiring questions, guiding my research directions. I would not have been a coach of University's Coding Club and a regional judge for ICPC without Dr. Travis B. Martin's assistance. Dr. Pål G. Drange helped me for successful collaboration and our valuable visit to Bergen, Norway. Finally, I appreciate my colleagues in Theory in Practice, Dr. Shweta Jain, Dr. Brian Lavallee, Madison Cooley, Cole Perschon, Annie Staker, Andrew Fraser, Alex Crane, David Dursteler, and Daniel Coimbra Salomao, for making my great experience possible in Salt Lake City, both in work and life.

CHAPTER 1

INTRODUCTION

Designing efficient algorithms and studying computational complexity under specific conditions have long been the heart of computer science. Many NP-hard graph problems become tractable on certain graph classes such as trees and cographs, but unfortunately, such graphs rarely show up in real-world datasets. Hence, solvers need to employ various heuristics and other algorithmic techniques (e.g., approximation) in practice. Modern solvers are equipped with sophisticated preprocessing algorithms on top of the core optimization tools. Such preprocessing may reduce the number of variables and constraints, but those numbers alone do not explain the often exponential speed-ups made by applying effective preprocessing steps. This is because the running time of modern algorithms for NP-hard problems is often not exponential in the total input size. Rather, it is exponential in some *parameter* k and scales polynomially with the input size.

The crux of this approach—known as *parameterized algorithms*—is that, although we often cannot expect a polynomial-time algorithm in the input size unless P=NP, there may be an algorithm to solve a problem in time $f(k) \cdot \text{poly}(n)$ for some parameter k and a computable function f. Such an algorithm is called *fixed-parameter tractable* (FPT) and offers one avenue for solving NP-hard problems. For typical optimization problems, the standard (or natural) parameter would be the size of the solution we are looking for. In other cases, structural parameters (a measure of some property of input instances) are better choices. In this spirit, dozens of *width* parameters have been developed, inspired by the renowned *treewidth* [14].

So, what exactly is "parameterized preprocessing"? A short answer could be *kernelization*; kernels—"smaller sized" instances equivalent to the original problem instance—are a key tool in the field and provide a notion of preprocessing with guarantees. But there are also limitations, especially when the solution size is large. We, therefore, need to

develop additional techniques to scale. A common theme in my research is the exploration of effective preprocessing steps, possibly with unconventional parameters, that lead to practical parameterized algorithms.

Another component of my body of work is a collection of parameterized algorithms in practice. Technical tools brewed in the parameterized algorithms community provide new perspectives for tackling real-world problems. Particularly, this dissertation exhibits applications in biology and robotics, incorporating significant effort of algorithm engineering and experiments.

1.1 Overview

The rest of this chapter consists of a list of software developed (Section 1.2) and preliminaries that are relevant to all of my work (Section 1.3). We first introduce mathematical symbols and (mostly graph-theoretic) notation (Section 1.3.1) and then present a brief summary of parameterized algorithms and complexity (Section 1.3.2). In Section 1.3.4, we illustrate an overview of structural parameters and define selected relevant examples.

The next six chapters discuss independent topics, each of which corresponds to a conference paper. Broadly speaking, Chapters 2 and 3 concern preprocessing, Chapters 4 and 5 consider structural parameters, and Chapters 6 and 7 address a problem that emerged from robotics applications.

In Chapter 2, we study the EXACT WEIGHTED CLIQUE DECOMPOSITION problem. We prove the existence of a kernel of size $k \cdot 2^k$, improving the best known 4^k. We describe implementing our algorithm, together with new reduction and search rules, and observe over 100x speed-up on biologically-inspired benchmark datasets.

Chapter 3 introduces an alternate notion of preprocessing due to Donkers and Jansen [16] (*antler decompositions* for FEEDBACK VERTEX SET) and elaborates how we can identify in FPT time a similar structure (*tight Odd Cycle Cuts*) for ODD CYCLE TRANSVERSAL. This work is in collaboration with Eindhoven University of Technology, Netherlands.

For structural parameters, Chapter 4 details our work on "happy set" problems originated from an open question involving modular-width. We resolved the parameterized complexity of two problems: MAXIMUM HAPPY SET (MAXHS) and DENSEST k-SUBGRAPH (DkS). We designed FPT algorithms and proved that MAXHS is FPT by both modular-

width and clique-width, and also that DkS is FPT by neighborhood diversity, cluster deletion number, and twin cover number.

One of the trendiest structural parameters is *twin-width* [10], which intuitively measures a distance to a cograph. In Chapter 5, we introduce our award-winning twin-width solver Hydra Prime and its specifics. Further, an ongoing project on ML-based twin-width solvers is summarized in Section 5.6; this is joint work with University College Dublin, Ireland, Goethe University Frankfurt, Germany, and Technische Hochschule Mittelhessen, Germany.

Chapters 6 and 7 concern the same problem, GRAPH INSPECTION, a graph-theoretic approach to motion planning in robotics. In collaboration with the University of Bergen, Norway and Birkbeck, University of London, UK, we designed, implemented, and evaluated several algorithms & heuristics for GRAPH INSPECTION. ILP (Integer Linear Programming) and DP (Dynamic Programming) approaches are detailed in Chapter 6, showing better performance than the state-of-the-art solver. In Chapter 7, we introduce an algorithm using algebraic circuits, which has a clear advantage in asymptotic memory usage over other algorithms. The content in Section 7.3.3 is original to this dissertation and not included in the preprint on arXiv [6].

Lastly, we provide an overall summary and future directions in Chapter 8.

1.2 Software

We regularly developed and engineered implementations of our algorithms to assess their effectiveness on realistic problem instances. All software is open source, fully documented, and capable of reproducing our experiments. The following is a list of accompanying software.

- DeCAF (Chapter 2): `https://github.com/TheoryInPractice/DeCAF`
  A solver for EXACT WEIGHTED CLIQUE DECOMPOSITION employing novel kernelization techniques.

- Hydra Prime (Chapter 5): `https://github.com/TheoryInPractice/hydraprime`
  A PACE-2023-winning TWIN-WIDTH solver.

- Robotic Brewing (Chapters 6 and 7):

`https://github.com/TheoryInPractice/robotic-brewing`

A GRAPH INSPECTION solver implementing multiple algorithms (dynamic programming, ILP, algebraic, heuristics).

1.3 Preliminaries

This section introduces versatile mathematical notation and background theories that serve as a basis for all of my work. Project-specific technical preliminaries are included in each chapter.

1.3.1 Notation

For a set S, the notation 2^S indicates the power set of S. We use the standard notation for mathematical structures; $\mathbb{Z}$ is the set of all integers, $\mathbb{Z}_{\geq 0}$ is the set of non-negative integers, and $\mathbb{Z}_+ = \mathbb{N}$ is the set of positive integers. Similarly, $\mathbb{R}$ is the set of real numbers, $\mathbb{R}_{\geq 0}$ is the set of non-negative reals, and so on. For a field $\mathbb{F}$ and a set X, we write $\mathbb{F}[X]$ for the set of all polynomials in the variable X with coefficients in $\mathbb{F}$.

We extensively use the standard $\mathcal{O}(\cdot)$ notation for analyzing asymptotic behavior of functions. The $\mathcal{O}^*(\cdot)$ notation hides polynomial factors, and the $\tilde{\mathcal{O}}(\cdot)$ notation hides polylogarithmic factors in $\mathcal{O}(\cdot)$.

We generally follow the textbook by Diestel [15] for standard graph-theoretic definitions and notation. Unless otherwise specified, we consider finite, undirected, simple graphs. Such a graph $G = (V, E)$ consists of a set V (also denoted by $V(G)$) of vertices and a set $E \subseteq \binom{V}{2}$ (also denoted by $E(G)$) of edges. For ease of notation, we write uv for an undirected edge $\{u, v\} \in E$; note that $uv = vu$. We write $n(G) := |V|$ for the number of vertices and $m(G) := |E|$ for the number of edges in the graph. When it is clear from context, we write $n = n(G)$ and $m = m(G)$. We use $N_G(v) = N(v) := \{u \in V \mid uv \in E\}$ and $N_G[v] = N[v] := N_G(v) \cup \{v\}$ to denote the open and closed neighborhoods of a vertex v, respectively. We write $\deg_G(v) = \deg(v) := |N_G(v)|$ for the degree of a vertex v. For a directed graph G, we write $N_G^-(v) = N^-(v)$ and $N_G^+(v) = N^+(v)$ for the in-neighbors and the out-neighbors of v, respectively. Similarly, we write $\deg_G^-(v) = \deg^-(v)$ and $\deg_G^+(v) = \deg^+(v)$ for the in-degree and the out-degree of v, respectively. For a vertex set $S \subseteq V$ we define its open neighborhood as $N_G(S) := (\bigcup_{v \in S} N_G(v)) \setminus S$ and its closed

neighborhood as $N_G[S] := \bigcup_{v \in S} N_G[v]$. The subgraph of G induced by a vertex set $S \subseteq V$ is the graph $G[S]$ on vertex set S with edges $\{uv \in E \mid u, v \in S\}$. We use $G - S$ as a shorthand for $G[V \setminus S]$ and write $G - v$ instead of $G - \{v\}$ for singletons.

A *walk* in a graph G is a sequence of (not necessarily distinct) vertices $(v_1, \ldots, v_k)$ such that $v_i v_{i+1} \in E$ for each $i < k$. The walk is *closed* if we additionally have $v_k v_1 \in E$. In some chapters, we may write $(v_0, v_1, \ldots, v_k)$ or $v_0 v_1 \ldots v_k$ with $v_0 = v_k$ to describe a closed walk. A *cycle* is a closed walk whose vertices are all distinct. The *length* of a cycle $(v_1, \ldots, v_k)$ is k. A *path* is a walk whose vertices are all distinct. The *length* of a path $(v_1, \ldots, v_k)$ is $k - 1$. The vertices v_1, v_k are the *endpoints* of the path. For two (not necessarily disjoint) vertex sets S, T of a graph G, we say that a path $P = (v_1, \ldots, v_k)$ in G is an (S, T)-path if $v_1 \in S$ and $v_k \in T$. If one (or both) of S and T contains only one element, we may write this single element instead of the singleton set consisting of it. In some settings, we are given edge weights $w : E \to \mathbb{R}_{\geq 0}$. Otherwise, let $w(e) = 1$ for all $e \in E$. The *weight* of a walk $(v_1, \ldots, v_k)$ is the sum of the weights of its edges, i.e., $\sum_{i=1}^{k-1} w(v_i v_{i+1})$. The *distance* between two vertices $u, v \in V$, denoted by $d(u, v)$, is the minimum weight across all walks between u and v. The distance between a vertex v and a vertex set $S \subseteq V$ is the minimum distance between v and any vertex in S, i.e., $d(v, S) = d(S, v) := \min_{u \in S} d(v, u)$. We say vertices u and v are *twins* if they have the same neighbors, i.e., $N(u) \setminus \{v\} = N(v) \setminus \{u\}$. Further, they are called *true twins* if $uv \in E$ and *false twins* otherwise.

1.3.2 Parameterized Algorithms

My work spans several topics in the area of parameterized algorithms, which was introduced by Downey and Fellows in the 1990's [17] and has developed into a rich area of research. A number of mathematical tools and techniques have been developed [14, 34, 37], fine-grained complexity classes have been established [18, 19], and parameterized algorithms have been applied to practical solvers for a wide variety of NP-hard problems [38, 32].

1.3.2.1 Traditional complexity classes. Before discussing fine-grained complexity classes, let us recall several fundamental complexity classes for decision problems. Informally, the class P (NP, resp.) is the set of all decision problems that can be solved by a deterministic (non-deterministic, resp.) Turing machine in polynomial time. Similarly,

the class PSPACE is the set of all decision problems that can be solved by a (deterministic[1]) Turing machine using a polynomial amount of space. It is known that $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE}$. The class NP-hard is the set of problems that are at least as hard as the hardest problems in NP.

The class co-NP is the set of problems whose complement is in NP. For any complexity class $\mathcal{K}$, Karp and Lipton [28] introduced the non-uniform class $\mathcal{K}/\mathsf{poly}$, where an algorithm might differ per input size. Informally, the class $\mathcal{K}/\mathsf{poly}$ consists of the problems that would be in $\mathcal{K}$ if we have access to a polynomial-size *advice* string for each input length.

1.3.2.2 Parameterized complexity. We refer to the textbook by Cygan *et al.* [14] for an introduction to parameterized complexity theory. The goal of parameterized algorithms is to capture the exponential part of the problem complexity within a parameter, making the rest of the computation polynomial in the input size. For an alphabet Σ and the set of all strings Σ^*, a *parameterized* problem $L \subseteq \Sigma^* \times \mathbb{N}$ is called *fixed-parameter tractable* (FPT) if there exists an algorithm $\mathcal{A}$, a computable function $f : \mathbb{N} \to \mathbb{N}$, and a constant $c \in \mathbb{N}$ such that, given $(I, k) \in \Sigma^* \times \mathbb{N}$, the algorithm $\mathcal{A}$ correctly decides whether $(I, k) \in L$ in time bounded by $f(k) \cdot |(I, k)|^c$. Similarly, a parameterized problem is called *slice-wise polynomial* (XP) if there is an algorithm to solve the problem in time $f(k) \cdot |(I, k)|^{g(k)}$ for computable functions $f, g : \mathbb{N} \to \mathbb{N}$. The complexity classes containing all fixed-parameter tractable and slice-wise polynomial problems are called FPT and XP, respectively.

As for lower-bounds, the granular W-hierarchy was introduced to capture the exact complexity of various hard parameterized problems [19]. For $t \geq 1$, a parameterized problem $\mathcal{P}$ belongs to the class W[t] if there is a parameterized reduction from $\mathcal{P}$ to a problem called WEIGHTED CIRCUIT SATISFIABILITY with $\mathcal{C}_{t,d}$, the class of all boolean circuits with weft at most t and depth at most d. It is believed that $\mathsf{FPT} \subsetneq \mathsf{W}[1]$, and this is a stronger assumption than the P$\neq$NP conjecture (FPT$\neq$W[1] implies P$\neq$NP).

In this context, I will introduce two key notions relevant to my work: *kernelization* and *structural parameters*.

1.3.2.3 Kernelization. Nearly all practical software to deal with NP-hard problems incorporates a dedicated problem-specific preprocessing step. The goal of preprocessing

[1] Due to Savitch's theorem [3], allowing a non-deterministic Turing machine does not add any extra power.

is to solve efficiently the "easy parts" of a problem instance and reduce it to its computationally difficult "problem kernel." Formal analyses of effective preprocessing had been put aside for a long time, but with the emergence of parameterized complexity, the "lost continent" [14] of efficient algorithms called *kernelization* was finally unveiled.

Kernelization works by using *(data) reduction rules*, which are polynomial-time algorithms that map an instance (I, k) to an *equivalent*[2] instance (I', k'). A *kernelization algorithm* has an additional requirement that the final instance size after applying all reduction rules is bounded by some computable function $g : \mathbb{N} \to \mathbb{N}$ of k, the original parameter. We say the problem admits a polynomial (respectively, linear) kernel if g is polynomial (linear). The following lemma shows that kernelization is another way of defining fixed-parameter tractability.

Lemma 1.1 (Lemma 2.2 [14])**.** *A parameterized problem is FPT if and only if it admits a kernelization algorithm.*

1.3.3 Showing Intractability

One of the main themes in complexity theory is the distinction between P and $\mathsf{NP}\backslash\mathsf{P}$; what decision problems are unlikely to be solved in polynomial time? This led to the notion of NP*-complete* problems, which are the "hardest problems" in NP. To date, thousands of computational problems in a variety of disciplines have been shown to be NP-complete [2]. Each of them can be solved in polynomial time if and only if $\mathsf{P} = \mathsf{NP}$. The study of NP-completeness involves *reductions*, a means of relating the computational complexity of two different problems.

1.3.3.1 NP**-hardness Reductions.** There are several ways to show that a problem is NP-hard. Here we introduce *Turing reductions*, *Cook reductions*, and more restrictive *Karp reductions*.

A decision problem L is *Turing reducible* to another problem L' if there is a Turing machine M that, given an oracle for deciding L', can correctly decide L. An *oracle* for L' is a "black box" that decides L' in constant time. If M runs in polynomial time, then the reduction is called a *Cook reduction* [11]. It is straightforward to see that if L is *Cook reducible*

[2]The equivalence of two instances of Q, referred to the *safeness* of a reduction rule, is proven by showing $(I, k) \in Q$ if and only if $(I', k') \in Q$.

to L', then L' is at least as hard as L, as far as polynomial-time algorithms are concerned. For example, if L is NP-complete, then L' is NP-hard. We use this kind of reduction for Lemma 3.34 in Chapter 3.

Another common type of reduction is a *many-one* reduction. A many-one reduction from a decision problem L to another decision problem L' is an algorithm that transforms an instance I of L into an instance I' of L' such that I is a yes-instance of L if and only if I' is a yes-instance of L'. A polynomial-time many-one reduction is called a *Karp reduction* [2]. Since a Karp reduction implies a Cook reduction, the former is stronger than the latter. Hence, we use Karp reductions by default, for example, Lemma 3.35 in Chapter 3 and throughout Chapter 6.

1.3.3.2 Parameterized Reductions. To show the intractability of a parameterized problem, the standard toolbox in NP-completeness theory may be insufficient. Hence, we use the following analogous notion of reduction that transfers fixed-parameter tractability.

Let $A, B \subseteq \Sigma^* \times \mathbb{N}$ be two parameterized problems. A *parameterized reduction* from A to B is an algorithm that, given an instance (I, k) of A, outputs an instance (I', k') of B, satisfying the following:

1. (I, k) is a yes-instance of A if and only if (I', k') is a yes-instance of B,
2. $k' \leq g(k)$ for some computable non-decreasing function g, and
3. the running time of the algorithm is $f(k) \cdot |I|^{\mathcal{O}(1)}$ for some computable non-decreasing function f.

With this definition, parameterized reductions translate fixed parameter tractability as in the following theorem:

Theorem 1.2 (Theorem 13.2 [14])**.** *If there is a parameterized reduction from A to B and B is* FPT*, then A is* FPT *as well.*

We use parameterized reductions for Theorem 3.37 in Chapter 3.

1.3.4 Structural Parameters

Although the most "natural" parameter is the solution size for an optimization problem, many recent FPT algorithms utilize *structural parameters*. A structural parameter of

a graph[3] is an invariant that describes the structure of a given graph. This can be a graph-theoretic property such as *maximum degree* or *diameter*, but the most studied one is *treewidth*, a byproduct of the acclaimed graph minor theorems by Robertson & Seymour [35]. Treewidth measures how well the cut structure of a graph resembles a tree and admits a number of FPT algorithms using dynamic programming over its *tree decomposition*. One of the most prominent results involving treewidth is Courcelle's theorem [12], stating that computing any graph properties definable in the monadic second-order logic of graphs is FPT when parameterized by the treewidth.

Today, we have a diverse, growing ecology of structural parameters. As illustrated in Figure 1.1, structural parameters, solutions to optimization problems, and distances to specific graph classes (all of which are used in parameterized algorithm design) are often related to one another. Such relations are present if a parameter p is provably upper-bounded by a function of another parameter q for any graphs. In this case, we say q is stronger than p. There are trade-offs between the strength of a parameter and fixed parameter tractability. Graphs having a bounded weaker parameter form a larger graph class, but more problems become intractable. For instance, the DENSEST k-SUBGRAPH problem introduced in Section 3.1 is FPT when parameterized by the vertex cover number but is W[1]-hard by the weaker parameter clique-width. Such boundaries are problem-specific, and finding these boundaries is an active area of research in parameterized complexity [13, 24, 4]. There are many open questions in the picture of fine-grained fixed parameter tractability [24, 26].

1.3.4.1 Treewidth. Treewidth measures how a graph resembles a tree and admits FPT algorithms for a number of NP-hard problems, such as WEIGHTED INDEPENDENT SET, DOMINATING SET, and STEINER TREE [14]. Treewidth is defined by the following notion of tree decomposition.

Definition 1.3 (treewidth [14])**.** A *tree decomposition* of a graph G is a pair $(T, \{X_t\}_{t \in V(T)})$, where T is a tree and $X_t \subseteq V(G)$ is an assigned vertex set for every node t, such that the following three conditions hold:

[3]There are structural parameters of other mathematical structures, such as matrices, hypergraphs, and geometries.

- $\bigcup_{t \in V(T)} X_t = V(G)$.
- For every $uv \in E(G)$, there exists a node t such that $u, v \in X_t$.
- For every $u \in V(G)$, the set $T_u = \{t \in V(T) : u \in X_t\}$ induces a connected subtree of T.

The width of a tree decomposition is defined to be $\max_{t \in V(T)} |X_t| - 1$, and the *treewidth* of a graph G, denoted by tw, is the minimum possible width of a tree decomposition of G.

1.3.4.2 Clique-width. Clique-width is a generalization of treewidth which can capture dense but structured graphs. Intuitively, a graph with bounded clique-width k can be built from single vertices by joining structured parts, where vertices are associated by at most k labels, and we can treat the vertices with the same label as a group.

Definition 1.4 (clique-width [13])**.** For a positive integer w, a w-labeled graph is a graph whose vertices are labeled by integers in $\{1, \ldots, w\}$. The *clique-width* of a graph G, denoted by cw, is the minimum w such that G can be constructed by repeated application of the following operations:

- (O1) Introduce $i(v)$: add a new vertex v with label i.
- (O2) Union $G_1 \oplus G_2$: take a disjoint union of w-labeled graphs G_1 and G_2.
- (O3) Join $\eta(i, j)$: take two labels i and j, and then add an edge between every pair of vertices labeled by i and by j.
- (O4) Relabel $\rho(i, j)$: relabel the vertices of label i to label j.

This construction naturally defines a rooted binary tree, called a cw-expression tree G, where G is the root and each node corresponds to one of the above operations.

1.3.4.3 Neighborhood diversity. Neighborhood diversity is a parameter introduced by Lampis [31], which measures the number of twin classes.

Definition 1.5 (neighborhood diversity [31])**.** The *neighborhood diversity* of a graph $G = (V, E)$, denoted by nd, is the minimum number w such that V can be partitioned into w sets of twin vertices.

By definition, each set of twins, called a *module*, is either a clique or an independent set.

1.3.4.4 Cluster deletion number. Cluster (vertex) deletion number is the distance to a cluster graph, defined to be a collection of disjoint cliques.

Definition 1.6 (cluster deletion number)**.** A vertex set X is called a *cluster deletion set* if $G[V \setminus X]$ is a cluster graph. The *cluster deletion number* of G, denoted by cd, is the size of the minimum cluster deletion set in G.

1.3.4.5 Vertex cover number. The vertex cover number is the solution size of the classic VERTEX COVER problem.

Definition 1.7. A vertex set $X \subseteq V$ is a vertex cover of $G = (V, E)$ if for every edge $uv \in E$ either $u \in X$ or $v \in X$. The *vertex cover number* of G, denoted by vc, is the size of the minimum vertex cover of G.

1.3.4.6 Twin cover number. The notion of twin cover is introduced by Ganian [25] and offers a generalization of vertex cover number.

Definition 1.8 (twin cover number [25])**.** A vertex set $X \subseteq V$ is a *twin cover* of $G = (V, E)$ if for every edge $uv \in E$ either (1) $u \in X$ or $v \in X$, or (2) u and v are true twins. The *twin cover number*, denoted by tc, is the size of the minimum twin cover of G.

1.3.4.7 Modular-width. Modular-width is a parameter introduced by Gajarský *et al.* [24] to generalize simpler notions on dense graphs while avoiding the negative results brought by moving to the full generality of clique-width (e.g., many problems FPT for treewidth become W[1]-hard for clique-width [21, 22, 23]). Modular-width is defined using the concept of modular decomposition.

Definition 1.9 (modular-width [24])**.** Any graph can be produced via a sequence of the following operations:

- (O1) Introduce: Create an isolated vertex.
- (O2) Union $G_1 \oplus G_2$: Create the disjoint union of two graphs G_1 and G_2.
- (O3) Join: Given two graphs G_1 and G_2, create the complete join G_3 of G_1 and G_2. That is, a graph G_3 with vertices $V(G_1) \cup V(G_2)$ and edges $E(G_1) \cup E(G_2) \cup \{uv : u \in G_1, v \in G_2\}$.

- (O4) Substitute: Given a graph G with vertices $v_1, \ldots, v_n$ and graphs $G_1, \ldots, G_n$, create the *substitution* of $G_1, \ldots, G_n$ in G. The substitution is a graph $\mathcal{G}$ with vertex set $\bigcup_{1 \leq i \leq n} V(G_i)$ and edge set $\bigcup_{1 \leq i \leq n} E(G_i) \cup \{uw : u \in G_i, w \in G_j, (v_i, v_j) \in E(G)\}$. Each graph G_i is substituted for a vertex v_i, and all edges between graphs corresponding to adjacent vertices in G are added.

These operations, taken together in order to construct a graph, form a *parse-tree* of the graph. The width of a graph is the maximum size of the vertex set of G used in operation (O4) to construct the graph. The *modular-width*, denoted by mw, is the minimum width such that G can be obtained from some sequence of operations (O1)-(O4).

Finding a parse-tree of a given graph, called a *modular decomposition*, can be done in linear-time [39]. See Figure 1.2 for an illustration of modular decomposition. Gajarský *et al.* also give FPT algorithms parameterized by modular-width for PARTITION INTO PATHS, HAMILTONIAN PATH, HAMILTONIAN CYCLE and COLORING, using bottom-up dynamic programming along the parse-tree [24].

1.3.4.8 Twin-width. Twin-width is a novel graph parameter introduced in 2020 by Bonnet *et al.* [10] as a generalization of the *permutation width* for classes of permutations defined by Guillemot and Marx [27]. Twin-width measures a graph's structural complexity, quantifying the similarity between a graph and *cographs*, or *complement reducible graphs*, which can be reduced to a single vertex by repeatedly merging pairs of twins.

Although computing twin-width is NP-hard [7], the parameter has gained great interest because bounded twin-width graphs are ubiquitous. Well-studied graph classes such as bounded clique-width and bounded rank-width have bounded twin-width [10], and it is speculated that real-world graphs tend to have small twin-widths [5]. Also, twin-width has led to important theoretical results. Model checking in first-order logic (FO MODEL CHECKING) in graphs of bounded twin-width is FPT [10]. Recent results include FPT algorithms for 3-COLORING [9] and k-INDEPENDENT SET [8], as well as polynomial-time approximation algorithms for MINIMUM DOMINATING SET [8].

One can define twin-width via sequences of trigraphs. A *trigraph* G has a vertex set $V(G)$ and two disjoint edge sets: $E(G)$, its set of *black edges*, and $R(G)$, its set of *red edges*. The *black graph* $\mathcal{B}(G)$, the *red graph* $\mathcal{R}(G)$, and the *total graph* $\mathcal{T}(G)$ of a trigraph

are the graphs defined by $\mathcal{B}(G) = (V(G), E(G))$, $\mathcal{R}(G) = (V(G), R(G))$, and $\mathcal{T}(G) = (V(G), E(G) \cup R(G))$. A *red neighbor* (*black neighbor*, respectively) of a vertex $v \in V(G)$ in a trigraph G is any neighbor of v in $\mathcal{R}(G)$ (in $\mathcal{B}(G)$, respectively).

A (vertex) *contraction* in a trigraph G consists of merging two (not necessarily adjacent) vertices $u, v \in V(G)$ into a single vertex with a new label w and updating the trigraph into G' as follows. We have $V(G') = V(G) \setminus \{u, v\} \cup \{w\}$. For red edges, we set $R(G') = E(\mathcal{R}(G) - \{u, v\}) \cup \{wx \colon x \in (N_{\mathcal{B}(G)}(u) \triangle N_{\mathcal{B}(G)}(v)) \cup N_{\mathcal{R}(G)}(u) \cup N_{\mathcal{R}(G)}(v)\}$, where $\triangle$ denotes the symmetric difference of two sets. That is, the red edges not incident to u or v remain the same, and we introduce red edges between w and $x \in V(G) \setminus \{u, v\}$ if x is a black neighbor of either u or v (but not both), or G already contains red edge ux or vx. Finally, we have $E(G') = E(\mathcal{B}(G) - \{u, v\}) \cup \{wx \colon x \in N_{\mathcal{B}(G)}(u) \cap N_{\mathcal{B}(G)}(v)\}$. Here, a common black neighbor x of u and v transfers to the black edge wx. A *contraction sequence* of an n-vertex graph G is a sequence of trigraphs $G = G_n, \ldots, G_1$ such that G_i is obtained from G_{i+1} by performing one contraction[4]. Notice that G_i contains exactly i vertices. A *d-sequence* is a contraction sequence in which every vertex of each trigraph has at most d red neighbors. The *twin-width* of G, denoted by $\mathsf{tww}(G)$ is the minimum integer d such that G admits a d-sequence.

1.3.4.9 Properties of structural parameters. Finally, we note the relationship among the parameters defined above, which establishes the hierarchy shown in Figure 1.1.

Proposition 1.10 ([4, 24]). *Let* $\mathsf{cw}, \mathsf{tw}, \mathsf{cd}, \mathsf{nd}, \mathsf{tc}, \mathsf{vc}, \mathsf{mw}$ *be the clique-width, tree-width, cluster deletion number, neighborhood diversity, twin cover number, vertex cover number, and modular-width of a graph G, respectively. Then the following inequalities hold*[5]*: (i)* $\mathsf{cw} \leq 2^{\mathsf{tw}+1} + 1$*; (ii)* $\mathsf{tw} \leq \mathsf{vc}$*; (iii)* $\mathsf{nd} \leq 2^{\mathsf{vc}} + \mathsf{vc}$*; (iv)* $\mathsf{cw} \leq 2^{\mathsf{cd}+3} - 1$*; (v)* $\mathsf{cd} \leq \mathsf{tc} \leq \mathsf{vc}$*; (vi)* $\mathsf{mw} \leq \mathsf{nd}$*; (vii)* $\mathsf{mw} \leq 2^{\mathsf{tc}} + \mathsf{tc}$*; and (viii)* $\mathsf{cw} \leq \mathsf{mw} + 2$.

1.3.4.10 Computing structural parameters. In spite of the availability of FPT algorithms for NP-hard problems, such algorithms often require corresponding decompositions (e.g., a tree decomposition for treewidth, a cw-expression tree for clique-width,

[4] In some literature, a contraction sequence refers directly to a sequence of pairs of vertices to contract rather than trigraphs.

[5] $\mathsf{cw} \leq 2$ when $\mathsf{mw} = 0$; otherwise, $\mathsf{cw} \leq \mathsf{mw} + 1$.

a contraction sequence for twin-width, etc.) as input. This is a big obstacle when we implement a solver; we often find theoretical results impractical due to the time needed to obtain this decomposition.

In fact, it is known that computing treewidth, clique-width, and twin-width of a graph is NP-hard [1, 20, 7]. Naturally, exact and approximate algorithms for computing those parameters have been one of the most active areas in the parameterized algorithms community [29, 30, 36]. An exception is modular-width; finding a parse-tree of a given graph, called a *modular decomposition*, can be done in linear-time [39]. Other polynomial-time computable structural parameters include degree statistics (min/max degree, density, etc.), degeneracy (linear-time), girth[6], distance measures (radius, diameter, eccentricity), centrality measures[7], maximum matching, and minimum cut, etc. Thanks to such tractability, these parameters are often used as "features" for machine learning [33].

[6] $\mathcal{O}(\min\{n^{2.38}, m^{1.41}\})$.

[7] Degree centrality in $\mathcal{O}(n^2)$, closeness centrality in $\mathcal{O}(nm + n^2)$, eigenvector centrality in $\mathcal{O}(n^3)$, betweenness centrality in $\mathcal{O}(nm + n^2)$, etc.

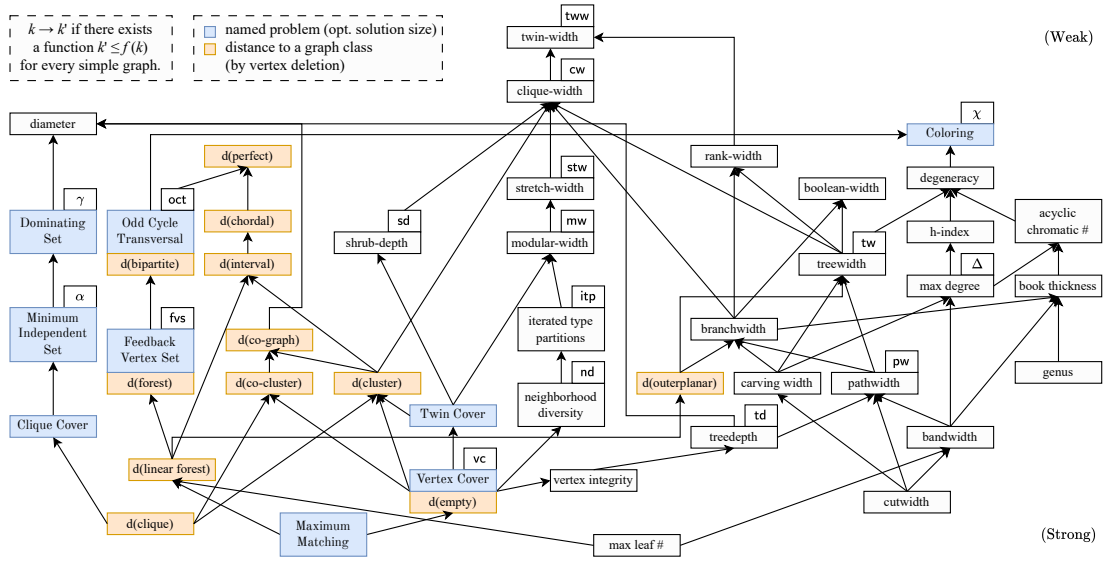


Figure 1.1: An overview of relevant structural graph parameters, depicted with solutions to optimization problems (shaded in blue) and distances to specific graph classes (orange), i.e., the minimum number of vertices to remove to obtain a desired graph. Arrows indicate relations, from the strong (bottom) to the weak (top).

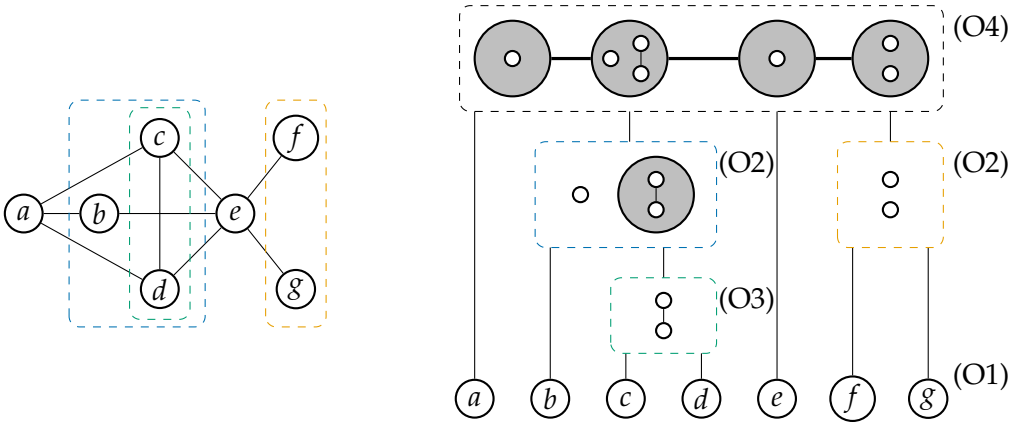


Figure 1.2: An example graph G (left) with modular-width 4 and its modular decomposition (right). The parse-tree has G as the root, and its nodes correspond to operations (O1)-(O4). Notice that each node also represents a *module*—module members have the same neighbors outside the module.

REFERENCES

[1] S. ARNBORG, D. G. CORNEIL, AND A. PROSKUROWSKI, *Complexity of Finding Embeddings in a k-Tree*, SIAM Journal on Algebraic Discrete Methods, 8 (1987), pp. 277–284.

[2] S. ARORA AND B. BARAK, *Computational Complexity: A Modern Approach*, Cambridge University Press, 2006.

[3] S. ARORA AND B. BARAK, *Computational complexity: a modern approach*, Cambridge University Press, 2009.

[4] Y. ASAHIRO, H. ETO, T. HANAKA, G. LIN, E. MIYANO, AND I. TERABARU, *Parameterized algorithms for the happy set problem*, Discrete Applied Mathematics, 304 (2021), pp. 32–44.

[5] M. BANNACH AND S. BERNDT, *PACE Solver Description: The PACE 2023 Parameterized Algorithms and Computational Experiments Challenge: Twinwidth*, in 18th International Symposium on Parameterized and Exact Computation (IPEC 2023), Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, pp. 35:1–35:14.

[6] M. BENTERT, D. C. SALOMAO, A. CRANE, Y. MIZUTANI, F. REIDL, AND B. D. SULLIVAN, *Graph Inspection for Robotic Motion Planning: Do Arithmetic Circuits Help?*, 2024. arXiv:2409.08219 [cs].

[7] P. BERGÉ, E. BONNET, AND H. DÉPRÉS, *Deciding twin-width at most 4 is NP-complete*, Dec. 2021. arXiv:2112.08953 [cs, math].

[8] E. BONNET, C. GENIET, E. J. KIM, S. THOMASSÉ, AND R. WATRIGANT, *Twin-width III: Max Independent Set, Min Dominating Set, and Coloring*, in 48th International Colloquium on Automata, Languages, and Programming (ICALP 2021), vol. 198 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2021, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 35:1–35:20.

[9] E. BONNET, E. J. KIM, A. REINALD, AND S. THOMASSÉ, *Twin-width VI: the lens of contraction sequences*, in Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Proceedings, Society for Industrial and Applied Mathematics, 2022, pp. 1036–1056.

[10] E. BONNET, E. J. KIM, S. THOMASSÉ, AND R. WATRIGANT, *Twin-width I: tractable FO model checking*, in 2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS), 2020, pp. 601–612.

[11] S. A. COOK, *The complexity of theorem-proving procedures*, in Proceedings of the third annual ACM symposium on Theory of computing, STOC '71, New York, NY, USA, 1971, Association for Computing Machinery, pp. 151–158.

[12] B. COURCELLE, *The monadic second-order logic of graphs. I. Recognizable sets of finite graphs*, Information and Computation, 85 (1990), pp. 12–75.

[13] B. COURCELLE AND S. OLARIU, *Upper bounds to the clique width of graphs*, Discrete Applied Mathematics, 101 (2000), pp. 77–114.

[14] M. CYGAN, F. V. FOMIN, L. KOWALIK, D. LOKSHTANOV, D. MARX, M. PILIPCZUK, M. PILIPCZUK, AND S. SAURABH, *Parameterized Algorithms*, Springer, 2015.

[15] R. DIESTEL, *Graph Theory*, vol. 173 of Graduate Texts in Mathematics, Springer-Verlag, Berlin, third ed., 2005.

[16] H. DONKERS AND B. M. JANSEN, *Preprocessing to reduce the search space: Antler structures for feedback vertex set*, Journal of Computer and System Sciences, 144 (2024).

[17] R. G. DOWNEY AND M. R. FELLOWS, *Fixed-parameter tractability and completeness*, Congr. Numer., 87 (1992), pp. 873–921.

[18] ——, *Fixed-Parameter Tractability and Completeness I: Basic Results*, SIAM Journal on Computing, 24 (1995), pp. 873–921.

[19] ——, *Fixed-parameter tractability and completeness II: On completeness for W[1]*, Theoretical Computer Science, 141 (1995), pp. 109–131.

[20] M. R. FELLOWS, F. A. ROSAMOND, U. ROTICS, AND S. SZEIDER, *Clique-Width is NP-Complete*, SIAM Journal on Discrete Mathematics, 23 (2009), pp. 909–939.

[21] F. V. FOMIN, P. A. GOLOVACH, D. LOKSHTANOV, AND S. SAURABH, *Clique-width: On the Price of Generality*, in Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, 2009, pp. 825–834.

[22] ——, *Algorithmic lower bounds for problems parameterized by clique-width*, in Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete algorithms, SODA '10, USA, 2010, Society for Industrial and Applied Mathematics, pp. 493–502.

[23] ——, *Intractability of Clique-Width Parameterizations*, SIAM Journal on Computing, 39 (2010), pp. 1941–1956.

[24] J. GAJARSKÝ, M. LAMPIS, AND S. ORDYNIAK, *Parameterized algorithms for modular-width*, in Parameterized and Exact Computation, G. Gutin and S. Szeider, eds., Springer International Publishing, 2013, pp. 163–176.

[25] R. GANIAN, *Improving Vertex Cover as a Graph Parameter*, Discrete Mathematics & Theoretical Computer Science, Vol. 17 no.2 (2015).

[26] G. C. M. GOMES, M. R. GUEDES, AND V. F. DOS SANTOS, *Structural Parameterizations for Equitable Coloring: Complexity, FPT Algorithms, and Kernelization*, Algorithmica, 85 (2022), pp. 1912–1947.

[27] S. GUILLEMOT AND F. SIKORA, *Finding and counting vertex-colored subtrees*, Algorithmica, 65 (2013), pp. 828–844.

[28] R. M. KARP AND R. J. LIPTON, *Some connections between nonuniform and uniform complexity classes*, in Proceedings of the twelfth annual ACM symposium on Theory of computing - STOC '80, Los Angeles, California, United States, 1980, ACM Press, pp. 302–309.

[29] T. KORHONEN, *A Single-Exponential Time 2-Approximation Algorithm for Treewidth*, in 2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS), 2022, pp. 184–192. ISSN: 2575-8454.

[30] T. KORHONEN AND D. LOKSHTANOV, *An Improved Parameterized Algorithm for Treewidth*, 2023. arXiv:2211.07154 [cs].

[31] M. LAMPIS, *Algorithmic Meta-theorems for Restrictions of Treewidth*, Algorithmica, 64 (2012), pp. 19–37.

[32] R. P. MUNIASAMY, R. NASRE, AND N. S. NARAYANASWAMY, *Accelerating Computation of Steiner Trees on GPUs*, International Journal of Parallel Programming, 50 (2022), pp. 152–185.

[33] O. NIGGEMANN AND B. STEIN, *A meta heuristic for graph drawing: learning the optimal graph-drawing method for clustered graphs*, in Proceedings of the working conference on Advanced visual interfaces, AVI '00, New York, NY, USA, 2000, Association for Computing Machinery, pp. 286–289.

[34] B. A. REED, K. SMITH, AND A. VETTA, *Finding odd cycle transversals*, Operations Research Letters, 32 (2004), pp. 299–301.

[35] N. ROBERTSON AND P. D. SEYMOUR, *Graph minors. II. Algorithmic aspects of tree-width*, Journal of Algorithms. Academic Press Inc., 7 (1986), pp. 309–322.

[36] A. SCHIDLER AND S. SZEIDER, *A SAT Approach to Twin-Width*, in 2022 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX), Proceedings, Society for Industrial and Applied Mathematics, 2022, pp. 67–77.

[37] A. SCHRIJVER, *Combinatorial optimization: polyhedra and efficiency*, vol. 24, Springer, 2003.

[38] H. TAMAKI, *Positive-instance driven dynamic programming for treewidth*, Journal of Combinatorial Optimization, 37 (2019), pp. 1283–1311.

[39] M. TEDDER, D. CORNEIL, M. HABIB, AND C. PAUL, *Simpler linear-time modular decomposition via recursive factorizing permutations*, in Automata, Languages and Programming, Berlin, Heidelberg, 2008, Springer Berlin Heidelberg, pp. 634–645.

CHAPTER 2

SMALLER KERNEL FOR EXACT WEIGHTED CLIQUE DECOMPOSITION

This chapter exemplifies kernelization for real-world applications. Mining groups of genes that consistently co-express is an important problem in biomedical research, where it is critical for applications such as drug-repositioning and designing new disease treatments [10, 31, 39, 26, 11]. Cooley *et al.* modeled this problem as EXACT WEIGHTED CLIQUE DECOMPOSITION (EWCD) in which, given an edge-weighted graph G and a positive integer k, the goal is to decompose G into at most k (overlapping) weighted cliques so that an edge's weight is exactly equal to the sum of weights for cliques it participates in. They show that EWCD is fixed-parameter-tractable, giving a 4^k-kernel alongside a backtracking algorithm (together called `cricca`) to iteratively build a decomposition. Unfortunately, because of inherent exponential growth in the space of potential solutions, `cricca` is typically able to decompose graphs only when $k \leq 11$. In this work, we establish reduction rules that exponentially decrease the size of the kernel (from 4^k to $k2^k$) for EWCD. In addition, we use insights about the structure of potential solutions to give new search rules that speed up the decomposition algorithm. At the core of our techniques is a result from combinatorial design theory called *Fisher's inequality* that characterizes set systems with restricted intersections. Experimental evaluation of our kernelization and decomposition algorithms (together called `DeCAF`) on a corpus of biologically-inspired data showed that in most cases `DeCAF` leads to over 80% reduction in the size of the kernel and orders of magnitude improvement in the time required to obtain a decomposition relative to `cricca`. As a result, `DeCAF` scales to instances with $k \geq 17$.

In this work, I was responsible for discussing and verifying our theoretical results centering on Fisher's inequality and conducting all computational experiments, including engineering algorithms, designing heuristics, and generating datasets. Jain, Sullivan, and I

published our work "An Exponentially Smaller Kernel for Exact Weighted Clique Decomposition" at the SIAM Conference on Applied and Computational Discrete Algorithms (ACDA23) [17].

2.1 Introduction

Network analysis has proven to be a very effective tool in biomedical research, in which phenomena such as the interactions between proteins and genes find natural representation as graphs [6, 20, 36, 15]. In gene co-expression analysis, for example, vertices represent genes and edges represent pairwise correlation between genes. Scientists are often interested in finding groups (modules) of genes that consistently co-act, which manifest as dense subgraphs or cliques in co-expression networks. The discovery of such modules is critical in understanding disease mechanisms and in the development of new therapies for diseases, especially when the primary genes associated with a disease may not be amenable to drugs [10, 31, 39, 26, 11].

A recent line of work [7, 12] models the module identification problem as EXACT WEIGHTED CLIQUE DECOMPOSITION (EWCD). In this problem, we are given a positive integer k and a graph G whose edges have positive weights. The goal is to find a decomposition of the vertices of the graph into at most k positive-weighted (possibly overlapping) cliques such that each edge participates in a set of cliques whose weights sum to its own. The cliques containing edge uv represent the modules in which genes u and v co-express, and clique weights correspond to the module's strength of effect on co-expression. Note that one can obtain a trivial decomposition by assigning every edge to its own 2-clique with matching weight, but this does not lead to any useful insights about the system. Hence, previous work has relied on the principle of parsimony and aimed to find the smallest number of cliques into which the graph is decomposable.

Although our work on EWCD is primarily motivated by gene co-expression analysis, a weighted clique cover is also used in applications such as finding the graphlet decomposition of a weighted graph [32], community detection [18, 5], identifying fraud rings [24], spotting suspicious reviews [16] etc., where heuristics and machine learning-based methods have been employed to address its combinatorial intractability. We anticipate our algorithm will be transferable to many of these other settings with minor modifications.

While EWCD is NP-hard, Cooley *et al.* [7] recently showed that it admits a *kernel*[1] of size 4^k. A kernelization algorithm is a polynomial-time routine that produces a (smaller) *equivalent* instance—i.e., the kernelized instance is a yes-instance iff the given instance is a yes-instance. The kernelization technique of Cooley *et al.* reduces an arbitrary sized instance of EWCD to an equivalent instance of (an annotated version of EWCD) with at most 4^k vertices. Cooley *et al.* also describe an algorithm for obtaining a valid decomposition of a kernelized instance (if one exists). In practice, their kernelization and decomposition algorithms (together called `cricca`[2]) are able to solve EWCD for graphs with $k \leq 11$ cliques in less than an hour. However, many co-expression networks have dozens of modules. Thus, a natural question is:

Does there exist a smaller kernel and/or a faster decomposition algorithm for the EXACT WEIGHTED CLIQUE DECOMPOSITION *problem?*

We answer this question in the affirmative, giving a $k2^k$-kernel and a faster decomposition algorithm (together called `DeCAF`) which in practice give at least two orders of magnitude reduction in running time over `cricca`[3]. Our kernelization technique uses a generalization of Fisher's inequality (from combinatorial design theory). We implement our algorithms and empirically evaluate them on an expansion of the corpus used in [7], demonstrating significantly improved practicality. We first summarize our contributions, then briefly describe the key ideas behind our approaches in Section 2.3.

2.1.0.1 Smaller kernel. We give a new kernel reduction rule (K-RULE 2+) that leads to a kernel of size at most $k2^k$. This is an exponential reduction in the size of the kernel compared to that of `cricca` which gives a kernel of size 4^k in the worst case. In practice, the kernel obtained using `DeCAF` is at least an order of magnitude smaller than the kernel obtained using `cricca` (Figure 2.1), which then reduces runtime for the downstream decomposition algorithm.

---

[1]The kernel of [7] is technically a *compression*: instances are reduced to equivalent instances of a closely-related problem.

[2]They also give an integer partitioning-based decomposition algorithm for the restricted case of integral weights.

[3]To be precise, over `cricca*`, an optimized version of `cricca` with updated search rules.

2.1.0.2 New search rules. The decomposition approach of [7] for EWCD is a backtracking algorithm that iteratively searches the space of potential solutions. Every time the algorithm builds a partial solution, it invokes an LP-solver to determine the weights of the cliques. Based on our insights about the structure of such solutions, we are able to design several new SEARCH RULES (S-RULEs) that prune away large parts of the search tree reducing the number of times the LP-solver needs to be called. In conjunction with the smaller kernel, these decrease the number of runs of the LP-solver by up to three orders of magnitude (Figure 2.2).

2.1.0.3 Faster solution to EWCD. Figure 2.3 shows the ratio of the total time taken by `DeCAF` and the total time taken by `cricca` for solving EWCD for several graphs with varying ground-truth k. Because of the smaller kernel and new S-RULEs, we are able to obtain up to two orders of magnitude reduction in the total running time, and the amount of reduction increases as k grows.

2.1.0.4 Scale to larger k. `DeCAF` enables decomposition[4] of graphs with k over 50% larger than `cricca` (see Figures 2.4 and 2.5).

2.2 Preliminaries

(A)EWCD

Input: a graph $G = (V, E)$, a non-negative weight function w_e for $e \in E$, (a special set of vertices $S \subseteq V$, a non-negative weight function w_v for $v \in S$), and a positive integer k.

Problem: a set of at most k cliques $C_1, \ldots C_k$ with clique weights $\gamma_1, \ldots \gamma_k \in \mathbb{R}^+$ such that $w_{uv} = \sum_{i:uv \in C_i} \gamma_i$ for all $uv \in E$ (and $w_v = \sum_{i:v \in C_i} \gamma_i$ for all $v \in S$) if one exists, otherwise output Infeasible.

The problems EWCD and its generalization ANNOTATED EWCD (AEWCD) which allows some vertices to be annotated i.e., have positive weights, were first introduced by Cooley *et al.* as a combinatorial model for discovering modules in gene co-expression graphs [7]. EWCD is the special case of AEWCD when the set $S = \varnothing$. The authors showed that an instance of EWCD can be reduced to an equivalent instance of AEWCD with at

[4]Subject to a 3600s timeout (matching that in [7]).

most 4^k vertices [7]. Their reduction techniques are based on those of Feldman *et al.* [12], who consider a closely related problem WEIGHTED EDGE CLIQUE PARTITION (WECP) and its generalization ANNOTATED WECP (AWECP). WECP is the special case of EWCD when the clique weights are restricted to be 1. Specifically, for a graph G with edge weights w_e and $k \in \mathbb{Z}^+$, (G, k) is a yes-instance of WECP if and only if there is a multiset of at most k cliques such that each edge appears in exactly w_e cliques.

Both these works build on the linear algebraic techniques of [3], and give equivalent matrix problem formulations in which matrices are allowed to have wildcard entries denoted by $\star$. For $a, b \in \mathbb{R} \cup \{\star\}$, let $a \overset{\star}{=} b$ if either $a = b$ or $a = \star$ or $b = \star$. We use M_i to represent the i^{th} row of a matrix M and M_{ij} to represent the element in the i^{th} row and the j^{th} column. For matrices $M1$ and $M2$, we write $M1 \overset{\star}{=} M2$ if $M1_{ij} \overset{\star}{=} M2_{ij}$ for each i, j. The reformulation of AEWCD (given by [7]) is called BINARY SYMMETRIC WEIGHTED DECOMPOSITION WITH DIAGONAL WILDCARDS (BSWD-DW).

BSWD-DW

Input: a symmetric matrix $A \in \left(\mathbb{R}_0^+ \cup \{\star\}\right)^{n \times n}$ with wildcards appearing on a subset of diagonal entries, and a positive integer k

Problem: a matrix $B \in \{0,1\}^{n \times k}$ and a diagonal matrix $W \in (\mathbb{R}_0^+)^{k \times k}$ such that $A \overset{\star}{=} BWB^T$ if such (B,W) exist, otherwise output NO.

Essentially, the goal is to find an $n \times k$ binary matrix B and a $k \times k$ diagonal matrix with non-negative elements, W such that $A \overset{\star}{=} BWB^T$. The row vector B_i represents the membership information of the i^{th} vertex, i.e., $B_{i,j}$ represents whether the vertex i is a member of the j^{th} clique. We call B_i the *signature* of the vertex i. In the diagonal matrix W, each column (and row) represents a clique from the solution and the element $W_{i,i}$ represents the weight of the i^{th} clique. The matrix A represents the weighted adjacency matrix of G i.e., $A_{i,j}$ represents the weight of the edge (i, j). If edge (i, j) does not exist in G, then $A_{i,j} = 0$. The wildcard entries in A are used for vertices with no weight restrictions. EWCD is thus the special case of BSWD-DW where all the diagonal entries are wildcards.

We say that two distinct vertices u and v in G are $\star$-twins if they are adjacent and satisfy $A_u \overset{\star}{=} A_v$. We partition the vertices of G (and correspondingly the rows of A) into sets of

vertices called *blocks* such that vertices u and v belong to the same block iff u and v are $\star$-twins. [12] showed that blocks are essentially equivalence classes.

2.2.1 Related Work

There is a rich history of work emphasizing the importance of mining patterns in gene co-expression analysis [38, 39, 37, 31] in which information about pairwise correlations for all genes in organisms [27] is used to derive useful knowledge about sets of genes whose expression is consistently modulated across the same tissues or cell types. Typically, unsupervised network-based learning approaches [23, 34, 10] are used for mining such modules. The first combinatorial model of module identification was EXACT WEIGHTED CLIQUE DECOMPOSITION (EWCD), introduced in [7] in ACDA21. They gave a 4^k-kernel and two parameterized algorithms for obtaining the decomposition of a graph into weighted cliques, one based on linear programming (LP) that works for real-valued weights, and an integer partitioning-based algorithm for the restricted case of integral weights. Both algorithms performed comparably so we use the unrestricted LP-based algorithm.

If clique weights are restricted to all being 1, EWCD is equivalent to WEIGHTED EDGE CLIQUE PARTITION (WECP) as studied by [12]. Their work builds upon the linear algebraic techniques of Chandran *et al.* [3] for solving the BICLIQUE PARTITION problem. WECP itself generalizes EDGE CLIQUE PARTITION (ECP) [21] which seeks a set of cliques containing each edge at least once (but no constraint on the maximum number of occurrences). It is known that ECP admits a k^2-kernel in polynomial time [30].

On the matrix factorization side, several symmetric [40, 4, 29] and asymmetric variants [28, 13, 14, 2, 19, 3] have been studied. However, they typically either do not allow wildcard entries, or allow B to be any non-negative matrix (not just binary) and hence they are unable to model the clique decomposition problem, or in the case of asymmetric variants, correspond to finding a partition of the edges of a bipartite graph into bicliques (complete bipartite graphs) instead of cliques [3].

2.3 Main Ideas

The starting point for this work is the algorithm of [7] for EWCD. The algorithm first preprocesses the graph to remove isolated vertices from G and adjusts k accordingly (each isolated vertex must be a unique clique in every valid decomposition). It then uses the kernelization techniques of [12] to obtain a smaller equivalent instance of AEWCD. On this kernelized instance, it runs a (parameterized) clique decomposition algorithm that searches the space of clique membership signatures for each vertex. Note that the kernel reduction rules of [12] were proposed for the AWECP problem in which clique weights are restricted to be 1. [7] showed that the same rules can be used to obtain a kernel for AEWCD. We will follow a similar sequence by first giving a kernel reduction rule for AWECP and then showing that it can also be used for AEWCD. Thus, we first consider the setup of the AWECP problem.

The kernelization technique of [12] works by applying two reduction rules to blocks of $\star$-twins which either prune away many of the vertices from the block or act as easy checks for a no-instance. An important property of blocks is that they induce a complete subgraph with uniform edge-weights, since the vertices in a block are all $\star$-twins.

- KERNEL RULE 1 [[12]] If there are more than 2^k blocks, then output that the instance is a no-instance.

- KERNEL RULE 2 [informal, [12]] If there is a block D of size greater than 2^k, then pick two distinct $i, j \in D$. Convert G into an instance G' of AEWCD by setting the weight of vertex i equal to the weight of edge (i, j) and removing every vertex in D from G except i. Then (G, k) is a yes-instance iff (G', k) is a yes-instance.

For KERNEL RULE 1 (K-RULE 1), the authors [12] show that if G is a yes-instance, then any pair of vertices u, v in G such that u and v have different signatures in the solution must belong to different blocks. Since there can be at most 2^k possible signatures (binary vectors of length k) there can be at most 2^k blocks in a yes-instance.

For KERNEL RULE 2 (K-RULE 2), they first prove that if G is a yes-instance then for any block of vertices, there exists a solution in which all the vertices from the block either have the same signature or all have pairwise distinct signatures. Since there can be at most 2^k distinct signatures, if the number of vertices in a block is greater than 2^k then by

the pigeonhole principle, the signatures cannot all be distinct. Hence, there must exist a solution in which all the vertices in the block have the same signature. In such a case, we can keep just one representative vertex from the block to get a smaller instance.

Note that this means that any two vertices of such a block must participate in exactly the same set of cliques. For this to happen, for vertices u, v in the block, the weight w of the edge uv must equal the number of cliques that contain u and v in the solution. Any solution for the reduced instance that is extendable to a solution for the original instance must ensure that the representative vertex v is part of exactly w cliques. To enforce this condition the weight of the representative vertex is set to w in the reduced instance.

Thus, after applying K-RULEs 1 and 2 there are at most 2^k blocks with at most 2^k vertices in each block. In this way, the authors obtain a kernel of size 4^k. Although this kernel was proposed by [12] for WECP in which the clique weights have to all be 1, [7] showed that the same techniques apply even when the cliques are allowed non-unit weights.

One way to get a smaller kernel would be to reduce the number of vertices in a block. Some challenges in achieving this are highlighted in Section 2.3.1. The main observation in this work is that if a decomposition of a uniform-weighted complete subgraph on t vertices consists of a non-trivial clique (i.e., a clique spanning strictly $< t$ vertices), then the number of cliques in the decomposition must be $\geq t$. Specifically, we show that if G is a yes-instance then for any block with more than k vertices, all vertices of the block must have identical signatures. Thus, we apply K-RULE 2 to every block with $\geq k+1$ vertices. Since there are at most 2^k blocks and each block has at most k vertices, we obtain a $k2^k$-kernel. Although this gives a kernel for the WECP problem (in which cliques are constrained to have unit weight), similar to [7] we show that this works for EWCD as well. The running time of the kernelization algorithm remains unchanged at $\mathcal{O}(n^2 \log n)$. In practice, we observe that this K-RULE gives an order of magnitude reduction in the size of the kernel, which subsequently helps speed up the decomposition.

SEARCH RULES **(S-RULEs):** The decomposition algorithm of [7] uses backtracking to assign signatures to vertices one by one. While doing so, it checks to make sure that the new signature is compatible with the vertices that have already been assigned a signature. If no compatible signature is found for a vertex, the algorithm backtracks. It continues this process until all vertices have a valid assignment or it determines that no valid assignment

can be found. Having rules that can quickly detect that the current (partial) assignment cannot be extended to a valid decomposition helps us prune away branches of the search tree and reduce the running time. We propose several search reduction rules that restrict the set of allowed signatures, thus speeding-up the decomposition algorithm.

Our first S-RULE pertains the order in which we consider the vertices for signature assignment, which can significantly affect how much of the search tree must be explored. We tested several strategies, and found that a `push_front` approach in which vertices from reduced blocks are considered before those in non-reduced blocks was most effective. A justification for this strategy is in Section 2.5, and the empirical evaluation is shown in Section 2.6.2.

Our second S-RULE comes from the straightforward observation that when assigning a signature to a vertex, one must respect its non-neighbor relationships. More specifically, when finding a signature for a vertex v, for all u such that uv is not an edge, $B_u B_v^T = 0$. That is, if u and v are non-neighbors, they cannot share a clique. Thus, we only test those signatures for v that have no cliques in common with those of its non-neighbors.

Our third and final S-RULE makes use of the fact that if the graph is a yes-instance then there exists a solution in which all vertices in each block have either identical signatures or pairwise distinct signatures. Thus, when assigning signatures to the vertices of a block, either we assign a unique signature to all vertices in the block or assign the same signature to all vertices in the block. This eliminates assignments in which a block has the same signature appearing on more than 1 but not all vertices in the block. The improved kernel along with these S-RULEs speeds up decomposition by two orders of magnitude. Sections 2.4 and 2.5 give formal proofs of our results.

2.3.1 Challenges

One way to get a smaller kernel would be to reduce the number of vertices in a block. Any block (definitionally) consists exclusively of $\star$-twins, thus, the weight of every edge within the block must be the same. In other words, the vertices in the block form a complete subgraph with uniform edge-weights. If the weights are all 1, it is known that a decomposition of the complete subgraph on t vertices either consists of a clique containing all the vertices (called a trivial clique), or consists of at least t cliques (Theorem 6 of [30]).

In other words, if the given graph is a yes-instance and $t > k$, then the clique covering the block must be trivial. In such a case, we can simply remove all vertices but one from the block and apply the same reduction as in K-RULE 2.

When the edge weights are > 1, if we could show that for any decomposition S of such a complete subgraph, there exists a set of cliques $C \subset S$ such that C is a decomposition of a complete subgraph of edge-weight 1 on t vertices, then using induction and by the above theorem, either all cliques in S are trivial cliques or $|S| \geq wt$, where w represents the weight of the edges in the block. However, this is not necessarily true. For example, consider a complete subgraph on 4 vertices where weight of each edge is 2. The set consisting of 4 cliques (each clique consisting of a different set of 3 vertices) is a valid decomposition of the complete subgraph but no subset of these cliques corresponds to a decomposition of a 4-clique with unit-weighted edges. Moreover, in the more general case where a complete subgraph does not have uniform weights, there exist weighted complete subgraphs that can be decomposed using $< t$ non-trivial cliques.

2.4 Smaller Kernel

Similar to [7], we will first consider the setup of the AWECP problem (in which the cliques are constrained to have weight 1), and prove correctness of our new reduction rule. We will then show that the rule remains valid in the case of AEWCD.

In the case of WECP (EWCD) since the vertices do not have weights, if a solution for an instance contains a singleton clique (i.e., a clique of size 1), we can safely remove such a clique from the solution. The remaining set of cliques also forms a solution (recall that isolated vertices have already been removed in the preprocessing step). Thus, in the rest of this chapter, we restrict our attention to solutions that do not contain singleton cliques. Note that, on the other hand an instance of AWECP (AEWCD) obtained from an instance of WECP (EWCD) *can* have singleton cliques to satisfy vertex weights. However, such vertices must necessarily be the representative vertices of reduced blocks.

We now formally define two types of $\star$-twins.

Definition 2.1 (Identical and fraternal twins)**.** Given a yes-instance (G, k) of AWECP, $\star$-twins u and v in G are called *identical twins* if there exists *no* solution in which u and v have distinct signatures, and *fraternal twins* otherwise.

Note that there can be solutions in which fraternal twins have identical signatures – we only require that in *some* solution they have different ones. Moreover, since a block consists of $\star$-twins, if any two vertices in a block are identical twins, then all must be.

We call such blocks *identical blocks*.

K-RULE 2 from [12] implies that all blocks with $> 2^k$ vertices are identical blocks. Our main insight is that a broader set of blocks must have this property. More specifically, any block with $> k$ vertices must be an identical block. To prove this, we will use a result from combinatorial design theory known as the non-uniform Fisher's inequality:

Theorem 2.2 (restated from [25])**.** *Let w be a positive integer and let $A = \{A_1, ..., A_t\}$ be a family of subsets of $U = \{e_1, ..., e_r\}$. If $|A_i \cap A_j| = w$ for each $1 \leq i < j \leq t$, then $|A| = t \leq r = |U|$.*

Essentially, the non-uniform Fisher's inequality states that if we have a set of r elements and we form t subsets of these elements such that any two subsets intersect in exactly w elements, then the number of subsets can be at most the number of elements. Fisher's inequality was first proposed in the context of Balanced Incomplete Block Design (BIBD) (See [33] and [1]). The uniform version was first proposed by Ronald Fisher and Majumdar [22] showed that the inequality holds even in the non-uniform case. The inequality has since been proven and applied in many different problem areas. In fact, De Caen and Gregory [9] showed the following corollary which, as we will show below, directly corresponds to the problem we're considering.

Let K_t represent the unweighted, complete graph on t vertices, and wK_t denote the complete multigraph on t vertices where the multiplicity of every edge is w. Let us say that a partition R of the edge-set of wK_t into cliques is *non-trivial* if there exists any clique in R that is not K_t. Corollary 2.3 implies if R is non-trivial then $t \leq |R|$. This can also be viewed as a generalization of the clique partition theorem of De Bruijn and Erdös [8] for arbitrary w (the result of [8] was for $w = 1$).

Corollary 2.3 (restated from [9])**.** *Let R be a partition of the edge-set of wK_t into non-empty cliques. If not all cliques in R are K_t then $|R| \geq t$.*

We are now ready to prove our main theorem.

Theorem 2.4. *Let (G, k) be a yes-instance of* AWECP *and D be a block in G. If $|D| \geq k+1$ then D must be an identical block.*

Proof. Suppose D is not identical. Since G is a yes-instance, there exists a solution such that not all vertices from D have the same signature i.e., not all vertices from D appear in the same cliques. Let S be the multiset of cliques in such a solution. Let $S_D \subseteq S$ be the multiset of cliques in which vertices from D appear. From every clique $C \in S_D$, delete all vertices that are not in D and call the resultant multiset R. Thus, R is a multiset of subsets of D. Since any subset of vertices in a clique also forms a (smaller) clique, R is a multiset of cliques. One can think of the cliques in R as the "projection" of the cliques in S_D onto D. Since not all vertices from D appear in the same cliques, R must consist of a non-trivial clique.

We first show that the non-trivial cliques in R cannot all be singleton cliques. Suppose there is a singleton clique in R consisting of $v \in D$. Then there exists a non-singleton clique C in S_D such that $C \cap D = v$. Moreover, since C is not singleton, there exists a vertex $v' \in C \setminus D$. Let $u \neq v$ be a vertex in D. Since u and v are $\star$-twins and v is a neighbor of v', u must also be a neighbor of v' and $w_{uv'} = w_{vv'}$. Thus, there must exist a clique $C' \in S_D, u, v' \in C', v \notin C'$. In other words, every vertex $u \in D, u \neq v$ must be a part of some clique that v is not a part of. If each such clique projects into a singleton clique in R, then $|R| \geq |D| \geq k+1$ which is a contradiction since G is a yes-instance. Thus, there must exist a non-singleton, non-trivial clique in R i.e., there must exist a clique containing > 1 but not all vertices from D.

Let w be the weight of all edges among the vertices in D. Let H be an unweighted, complete multigraph on $|D|$ vertices in which the multiplicity of every edge is w. It is easy to see that R is a partition of the edge-set of the multigraph H into cliques. If R consists of a non-singleton, non-trivial clique then by Corollary 2.3, $|D| \leq |R|$, which is a contradiction because G is a yes-instance (implying $|R| \leq k$ and $|D| \geq k+1$). Thus, R cannot consist of non-trivial cliques.

In other words, R must be trivial i.e., every clique in R must be a $K_{|D|}$. Thus, every vertex of D must be in the exact same set of cliques in S and have the same signature in the solution matrix B. Thus, D must be an identical block. □

Hence, we can reduce it to a representative vertex, leading to our enhancement of K-RULE 2:

- KERNEL RULE 2+: If there is a block D of size greater than k, then apply the reduction of K-RULE 2.

We further show that K-RULE 2+ gives a valid kernel even in the case of AEWCD. Theorem 3.1 gives the following corollary.

Corollary 2.5. *Let (G, k) be a yes-instance of AEWCD and D be a block in G. If $|D| \geq k+1$ then D must be an identical block.*

Proof. Let (A, k) be a yes-instance of AEWCD, and let (B, W) where $BWB^T = A$ be a solution for (A, k). Let $\bar{A} = BB^T$ be an instance of AWECP got by making the clique weights 1. Clearly, $(\bar{A}, k)$ is a yes-instance of AWECP. Thus, Theorem 3.1 gives that the block D in B must be identical. But the same B is a solution for A. Thus, D must be identical even for the AEWCD instance (A, k). □

Essentially, for any yes-instance (A, k) of AEWCD, there is an equivalent yes-instance $(\bar{A}, k)$ of AWECP that has the same B. A block being identical is only a function of the cliques in the decomposition, not the weights of the cliques. Thus, if a block is identical in $\bar{A}$, it is identical also in A.

The rest of the proof of correctness closely follows that of Rule 2 in [7] and proceeds in two parts.

Lemma 2.6. *Let (A', k) be the reduced instance constructed by applying* K-RULE 2+ *to (A, k). Then if (B', W') is a solution for (A', k) then the (B, W) constructed by* K-RULE 2+ *is indeed a solution to (A, k).*

Proof. We will prove that $B_u^T W B_v \stackrel{\star}{=} A_{uv}$ for all $u, v \in V(G)$, which guarantees validity. Let D be a block reduced by K-RULE 2+ and let i be the representative vertex. There are 3 cases:

- $u, v \notin D$: Then $B_u^T W B_v = {B'_u}^T W B'_v \stackrel{\star}{=} A'_{uv} = A_{uv}$.
- $u \in D, v \notin D$: Then $B_u^T W B_v = {B'_u}^T W B'_i \stackrel{\star}{=} A'_{ui} = A_{ui}$.

- $u, v \in D$: If $A_{uv} = \star$ then this case is trivial. Assume $A_{uv} \neq \star$. Lemma 7 of [12] states that any two non-$\star$ entries in the same block of matrix A must be equal. Hence, $B_u^T W B_v = {B'_i}^T W B'_i \stackrel{\star}{=} A'_{ii} = A_{ij} = A_{uv}$.

This completes the proof that (B, W) is a solution to (A, k). □

Lemma 2.7. *If (A, k) is a yes-instance, then the reduced instance (A', k) produced by* K-RULE 2+ *is a yes-instance.*

Proof. Let (B, W) be a solution of (A, k). Since the block D contains more than k rows, by Corollary 2.5, D must be an identical block. Thus, there exist row indices $p, q \in D$ such that $B_p = B_q$. We define a solution (B', W) for (A', k) as $B'_u := B_u$ for all $u \in V(G') \setminus \{i\}$ and $B'_i := B_p$, where i is the representative vertex of D.

To prove that (B', W) is indeed a valid solution for (A', k), it is sufficient to show that ${B'_u}^T W B'_v \stackrel{\star}{=} A'_{uv}$ for all $u, v \in V(G')$. There are 3 cases to consider:

- $u, v \neq i$: Then ${B'_u}^T W B'_v = {B_u}^T W B_v \stackrel{\star}{=} A_{uv} = A'_{uv}$.
- $u = i$,$v \neq i$: Then ${B'_i}^T W B_v = {B_p}^T W B_v \stackrel{\star}{=} A_{pv} = A_{iv} = A'_{iv}$, where the second-to-last equality follows from p and i being in the same block D.
- $u = v = i$: In this case, ${B'_i}^T W B'_i = {B_p}^T W B_p = B_p^T W B_q = A_{pq} \stackrel{\star}{=} A_{ij} = A'_{ii}$. As in the proof of Lemma 2.6 the $\stackrel{\star}{=}$ here follows because any two entries (that are not $\star$) in the same block of matrix A are equal (Lemma 7 of [12]). The third equality here is an equality (and not only a '$\stackrel{\star}{=}$ equivalence') as A_{pq} is not a diagonal entry.

This completes the proof that if (A, k) is a yes-instance then (A', k) is also a yes-instance. □

Thus, the kernelization algorithm of `DeCAF` applies K-RULE 1 of [7] and K-RULE 2+ to get a kernel that is at most $k2^k$ in size. This leads to our main result:

Theorem 2.8. *Given a graph G on n vertices and a positive integer k, the kernelization algorithm of* `DeCAF` *finds a kernel for* EXACT WEIGHTED CLIQUE DECOMPOSITION *of worst-case size $k2^k$ in time $\mathcal{O}(n^3)$.*

Proof. The kernelization algorithm sorts the rows in A to form blocks and reduces each block that has $\geq k+1$ vertices. The sorting of rows of A and division into blocks takes time $\mathcal{O}(n^3)$ and the application of K-RULE 1 and K-RULE 2+ take time $\mathcal{O}(n)$. Thus, the time complexity of the kernelization algorithm is $\mathcal{O}(n^3)$, matching that of [7]. □

2.5 Faster Decomposition Algorithm

Once a kernelized instance is obtained, one can run the decomposition algorithm of [7], shown in Algorithm 1 on it. The algorithm assigns signatures to vertices one-by-one, iteratively building a solution. When only a subset of all vertices have been assigned a signature, we call this a *partial assignment*. When trying to find a compatible signature for a vertex, the algorithm searches the entire space of 2^k possible signatures for that vertex. For every signature that the algorithm considers for a vertex, the clique weights as given by the weight matrix W may need to change. `iWCompatible` (Algorithm 3) checks if the weight matrix W is compatible with the current partial assignment. If not, to find a compatible new set of weights, the algorithm builds a Linear Program (LP) which encodes the partial assignment and the edge-weights (`InferCliqWts-LP`, Algorithm 4). If the LP returns a solution, the algorithm updates the weight matrix. If the LP fails to return a solution, it means that no feasible weight matrix exists for this partial assignment. In this case, the algorithm backtracks.

Algorithm 1: `CliqueDecomp-LP`

```
for P ∈ {0,1}^{2k×k} do
    initialize B̃ to a n × k null matrix
    b, i ← 1
    while b ≤ 2k do
        B̃_i ← P_b
        b ← b + 1
        W ← InferCliqWts-LP (A, B̃)
        if W is not null matrix then
            (B, i) ← FillNonBasis (A, B̃, W)
            if i = n + 1 then
                return (B, W)
        else
            b ← 2k + 1                    // null W; break out of while
return No
```

The main insight of [7] was that the pseudo-rank of B is at most $2k$ and that once the basis vectors of B have been guessed correctly, there will be no need to backtrack when filling in the signatures of other vertices. They showed that we need to run the LP only when the algorithm backtracks and that this happens for at most 2^{2k^2} partial assignments. `FillNonBasis` (Algorithm 2) shows the pseudocode for filling in the signatures for the non-basis vectors. Note that every time the algorithm picks a new set of basis vectors (P), the existing assignment of signatures (even non-basis vectors) are discarded. After an exhaustive search if no valid assignment is found, the algorithm returns that the instance is a no-instance.

Algorithm 2: `FillNonBasis` $(A, \widetilde{B}, W)$

```
1 B ← B̃
2 while B has a null row do
3 |   let B_i be the first null row
4 |   for v ∈ {0,1}^k do
5 |   |   if iWCompatible (A, B, W, i, v) then
6 |   |   |   B_i ← v
7 |   |   |   goto line 2
8 |   return (B, i)                    // there is no (i,W)-compatible v
9 return (B, n+1)                      // B has no null row
```

Algorithm 3: `iWCompatible` $(A, \widetilde{B}, W, i, v)$

```
1 for each non-null row B_j do
2 |   if v^T W B_j ≠ A_ij then
3 |   |   return false
4 if v^T W v ≠* A_ii then
5 |   return false
6 return true
```

We now design several search reduction rules that can help to quickly prune away branches that cannot lead to a solution.

- SEARCH RULE 0: *Assign signatures to reduced blocks before non-reduced blocks.* We observed during experiments that the order in which we assign signatures to vertices can significantly impact how fast the algorithm terminates. Ideally, we would like the

Algorithm 4: `InferCliqWts-LP` $(A, \widetilde{B})$

let $\gamma_1, \cdots, \gamma_k \geq 0$ be variables of the LP
for *all pairs of non-null rows* $\widetilde{B}_i, \widetilde{B}_j$ *s.t.* $A_{ij} \neq \star$ **do**
  add LP constraint $\sum_{1 \leq q \leq k} \widetilde{B}_{iq} \widetilde{B}_{jq} \gamma_q = A_{ij}$
if *the LP is infeasible* **then**
  return *the null matrix*
else
  return *the diagonal matrix given by* $\gamma_1, \ldots, \gamma_k$

first k vertices to be as close to an independent set as possible (since non-neighbors significantly restrict potential valid signatures, see S-RULE 1). Since vertices within a block must be neighbors, we wanted an order that hit many distinct blocks quickly, yet was compatible with the engineering required for S-RULE 2 (below). This led to the strategy `push_front`, which assigns signatures to all vertices which are representatives of reduced blocks (which necessarily have size 1) before proceeding to those in non-reduced blocks. We validated our choice by empirically evaluating this against several other orders including the `arbitrary` approach in [7]; details are in Section 2.6.2.

- SEARCH RULE 1: *For every vertex, generate only those signatures that don't share a clique with the non-neighbors of that vertex.* The main idea behind S-RULE 1 is that any two non-neighbors should not share a clique. Thus, for non-neighbors u and v, $B_u {B_v}^T = 0$. Whenever the algorithm is searching for a signature to assign to a new vertex, it generates the list of cliques that are "forbidden" for that vertex based on the signatures of the vertex's non-neighbors. It then uses this list to generate only those signatures that respect the "forbidden" cliques. In many cases, this drastically reduces the number of signatures to be tried.

- SEARCH RULE 2: Vertices across blocks must have unique signatures. The cliques that a vertex participates in cannot be a proper subset of the cliques its $\star$-twin participates in. Make all signatures in a block either identical or pairwise distinct.

We establish the correctness of our rules using the following Lemmas.

Lemma 2.9. *If u and v belong to different blocks then $B_u \neq B_v$.*

Proof. For contradiction, assume $B_u = B_v$. Since u and v belong to different blocks, they must not be $\star$-twins. Thus, either u is not adjacent to v or $A_u \stackrel{\star}{\neq} A_v$.

If u is not adjacent to v, $A_{uv} = B_u W B_v^T = 0$. Since, $B_u = B_v$ and since we do not allow cliques to have weight 0, $B_u W B_v^T = 0$ iff $B_u = B_v = 0$. Thus, it must be the case that $B_u = B_v = 0$. However, since the graph has already been preprocessed to remove all isolated vertices, the remaining vertices must all have at least one edge adjacent to them and hence must be a part of at least one clique. Thus, $B_u, B_v \neq 0$ which is a contradiction.

Now consider the case when u and v are adjacent but $A_u \stackrel{\star}{\neq} A_v$. Since $B_u = B_v$, $B_u W B^T = B_v W B^T$. Since $B_u W B^T \stackrel{\star}{=} A_u$ and $B_v W B^T \stackrel{\star}{=} A_v$, this implies that $A_u \stackrel{\star}{=} A_v$, a contradiction. □

Lemma 2.10. *Let (G, k) be a yes-instance of AEWCD. There exists a solution B such that for every pair of $\star$-twins u, v in G, $B_u \not\subset B_v$ and vice versa.*

Proof. Suppose there exist $\star$-twins u and v and a solution B such that $B_u \subset B_v$. Then there exists a clique C in the solution such that $v \in C, u \notin C$. If C is a singleton clique or C has weight 0, then we can remove C from the solution. Clearly, the remaining set of cliques would still be a solution and the theorem would be true. So assume C has positive weight and is not a singleton clique. Thus, there exists a vertex $v' \in C, v' \neq v$. Since u and v are $\star$-twins and v is a neighbor of v', u must also be a neighbor of v' and $w_{uv'} = w_{vv'}$. Let $S_{uv'}$ and $S_{vv'}$ be the set of cliques in the solution in which edges uv' and vv' participate, respectively. Then since $B_u \subset B_v$, $S_{uv'} \subset S_{vv'}$. Moreover, $w_{uv'} = \sum_{i:uv' \in C_i} \gamma_i$ where γ_i represents the weight of clique i. Similarly, $w_{vv'} = \sum_{i:vv' \in C_i} \gamma_i$. But since the cliques have positive weights and $S_{uv'} \subset S_{vv'}, w_{uv'} < w_{vv'}$ which is a contradiction. □

We know from Theorem 3.1 that blocks having $> k$ vertices must be identical blocks. However, blocks having $\leq k$ vertices can be fraternal. Moreover, by Lemma 11 of [12] we know that if the given instance is a yes-instance then there exists a solution in which all vertices in a block have either identical signatures or pairwise distinct signatures. We show that if the given instance is a yes-instance of AEWCD then there exists a solution in which this condition is *simultaneously* true for all blocks.

Theorem 2.11. *If (G, k) is a yes-instance of AEWCD, there exists a solution in which every block*

of G has either identical signatures or pairwise distinct signatures.

Proof. Consider any solution (B, W) to the given instance and suppose there exists a block D whose vertices have signatures that are neither all unique nor all pairwise-disjoint (if such a block does not exist then the theorem is trivially true). Thus, there exist vertices v_1, v_2, v_3 in D such that $B_{v_1} = B_{v_2} \neq B_{v_3}$. Consider the $n \times k$ matrix B' got by setting $B'_{v_1} = B_{v_1}$, $B'_{v_2} = B_{v_2}$, $B'_{v_3} = B_{v_1}$ and $\forall x \neq v_1, v_2, v_3, B'_x = B_x$ i.e., B' has the same signatures as B for all vertices except v_3 and B'_{v_3} is set to B_{v_1}. To prove that (B', W) is indeed a valid solution for (G, k), it is sufficient to show that $B'_u W B'^T_v \stackrel{\star}{=} A_{uv}$ for all vertices u, v in G. There are 3 cases to consider:

- $u, v \neq v_3$: $B'_u W B'^T_v = B_u W B_v^T \stackrel{\star}{=} A_{uv}$.
- $u = v_3$,$v \neq v_3$: $B'_u W B_v^T = B_{v_1} W B_v^T \stackrel{\star}{=} A_{v_1 v} = A_{v_3 v}$, where the last equality follows from v_1 and v_3 being in the same block D.
- $u = v = v_3$: in this case, $B'_{v_3} W B'^T_{v_3} = B_{v_1} W B_{v_3}^T \stackrel{\star}{=} A_{v_1 v_3} \stackrel{\star}{=} A_{v_3 v_3}$. The $\stackrel{\star}{=}$ here follows because any two entries (that are not $\star$) in the same block of matrix A are equal (Lemma 7 of [12]).

We can apply this iteratively to all vertices in D and to other blocks until all blocks either have identical signatures or pairwise distinct signatures. □

Thus, we only need to search over those assignments that satisfy the conditions of Theorem 2.11. In Line 5 in `CliqueDecomp-LP` and Line 4 in `FillNonBasis`, when assigning a signature to a vertex we ensure that all vertices in its block have either identical or unique signatures. This eliminates assignments in which a block consists of some repeated signature but not all identical signatures, thereby reducing the search space.

2.5.0.1 Running time. The overall running time of `DeCAF` is $\mathcal{O}(4^{k^2} 2^k k^4 (k^2 2^{3k} + k^3 L))$, where L is the number of bits required for input representation [35]. If $L = \mathcal{O}(2^k)$ this comes to $\mathcal{O}(4^{k^2} k^6 16^k)$ and for `cricca` is $\mathcal{O}(4^{k^2} k^2 32^k)$. In practice we are able to get at least two orders of magnitude speedup.

The `for` loop in Line 1 of Algorithm 1 has at most 2^{2k^2} iterations. `FillNonBasis` takes time $\mathcal{O}(n^2 k 2^k)$, where n is the number of vertices in the kernelized instance. Algorithm `InferCliqWts-LP` solves an LP with k variables and at most $4k^2$ constraints which can be

solved in $\mathcal{O}(k^4L)$ time where L is the number of bits required for input representation [35]. `iWCompatible` runs in time $\mathcal{O}(nk)$ and hence, the total time taken by `FillNonBasis` is $\mathcal{O}(n^2k2^k)$. Hence, the total time taken by `CliqueDecomp-LP` without the S-RULEs is $\mathcal{O}(2^{2k^2}2k(k^4L + n^2k2^k))$. Since $n \leq 4^k$, the total running time of `CliqueDecomp-LP` as given by [7] is $\mathcal{O}(4^{k^2}k^2(32^k + k^3L))$. Thus, if $L = \mathcal{O}(2^k)$, this comes to $\mathcal{O}(4^{k^2}k^232^k)$.

S-RULE 0 pushes singleton vertices from reduced blocks to front in time $\mathcal{O}(n)$. S-RULE 1 when assigning a signature to a vertex v, loops over the non-neighbors of v and generates the list of forbidden cliques in time $\mathcal{O}(nk)$. S-RULE 2 when assigning a signature to a vertex, compares with the signatures of other vertices in the block in $\mathcal{O}(nk)$ time. Thus, the running time of `CliqueDecomp-LP` with S-RULEs is $\mathcal{O}(2^{2k^2}2nk^2(k^4L + n^2k2^k))$. Due to the new kernel, we can set $n = k2^k$. This gives an overall running time of $\mathcal{O}(4^{k^2}2^kk^4(k^22^{3k} + k^3L))$.

2.6 Experimental Results

We compared the performance of our kernelization and decomposition algorithms with those of [7]. We use the publicly available Python package of [7] provided by its authors and implement our new algorithms as an extension. We ran all experiments on identical hardware, equipped with 40 CPUs (Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz) and 191000 MB of memory, and running CentOS Linux release 7.9.2009. We used Gurobi Optimizer 9.1.2 as the LP solver, parallelized using 40 threads. A timeout of 3600 seconds per instance was used in all experiments. Code and data to replicate all experiments are available at `https://github.com/TheoryInPractice/DeCAF`.

2.6.0.1 Notation. We use K_D and S_D to denote our kernelization and decomposition algorithms, respectively. We use K_C to denote the kernelization algorithm of [7], and S_C (S_{C^*}) for the decomposition algorithm of [7] *without (with)* S-RULE 0. Thus, `cricca` represents the combination (K_C, S_C), `cricca*` is (K_C, S_{C^*}) and `DeCAF` is(K_D, S_D). We use n_C and n_D to denote number of vertices remaining in the graph (the kernel) after applying K_C and K_D respectively. For some experiments, we subselect instances based on their seed, as detailed in the experiment description.

2.6.0.2 Datasets. The authors of [7] evaluate `cricca` on two biology-inspired synthetic

datasets (LV and TF, seeds 0-19). Details about how these datasets were created are in Section 7 and Appendix D of [7]. We use the same datasets and expand the corpus for our experiments using the same procedure for generating the graphs as [7] (with different seeds), noting that validating scalability and correctness in the exact setting is a necessary precursor to evaluation on noisy real-world data with unknown ground truth. Detailed instructions for generating the data are in our codebase.

We assume that every graph has been preprocessed to remove isolated vertices. Each (preprocessed) graph G has a ground-truth k—the minimum k such that (G, k) is a yes-instance. We distinguish this from k_{in}, which denotes the number of desired cliques input to the algorithms; each (G, k_{in}) denotes a unique instance of EWCD. Details about these instances can be found in Table 2.1. As in [7], we restrict our main corpus to graphs with $3 \leq k \leq 11$; for $k \in [5, 7]$, we generated multiple instances for each G using several values of k_{in} above and below k (E.3 in [7]). The number of unique G are shown in column "#G" and the number of (G, k_{in}) instances are shown in the column "#(G, k_{in})". The columns n and m denote the average number of vertices and edges across all instances per k value. For ease of comparison of running times, we do not include instances that timed out on any combination of K_C, K_D, S_{C^*} and S_D in Table 2.1. A detailed breakdown of timeouts is in Section 2.6.5.

2.6.1 Smaller Kernel

To evaluate the effect of KERNEL RULE 2+, we compared the kernel size and the running time of K_D with those of K_C. We used the entire dataset (seeds 0-19) of ground-truth k between 3 and 11 (inclusive), with input k (k_{in}) varied between $0.4k$ and $1.6k$. Because the difference between two versions is only the threshold value used in K-RULE 2 and K-RULE 2+, it holds that $n_D \leq n_C$. Further, if K_C *solves* the problem, that is, it finds the instance a no-instance, then so does K_D.

Figure 2.1 plots the ratio n_D / n_C for each instance, colored by k_{in}. We can see that most instances got shrunk to less than 20% of n_C. Also we observe that our kernel size n_D decreases (1) as n increases and (2) as k_{in} increases. The former is intuitive; K-RULE 2+ is more effective when the instance is larger, removing more vertices from the graph. The latter is explained by the fact that K_C is effective only for blocks that are exponential in size with respect to k_{in} and thus, has less impact when k_{in} is large, where as K_D is able to reduce

all blocks with $> k_{in}$ vertices and hence is effective even when k_{in} is large.

For the running time of the kernelization process, we computed the relative running time i.e., the ratio of the running time of K_D to that of K_C. The first and third quantiles were 0.99 and 1.01, respectively, and the maximum was 1.24. We conclude that there is no significant difference in running times.

2.6.2 Effects of Vertex Reordering

We found that vertex ordering has a significant impact on the efficiency of the decomposition process, especially when we obtain a smaller kernel. We experimented with the following orderings.

`arbitrary` is the baseline strategy used in `cricca`. It keeps one arbitrary vertex from each block that is reduced and does not reorder the vertices. `push_front` is the one adopted as S-RULE 0. It keeps one arbitrary vertex from each block that is reduced and moves the representative vertex to the front of the ordering. `push_back` does the opposite; it keeps one arbitrary vertex from each block that is reduced and moves the representative vertex to the back. `keep_first` keeps the earliest vertex from each block that is reduced and does not reorder the vertices.

To clearly distinguish the effect of ordering (S-RULE 0) from the other S-RULEs, we ran S_{C^*} (S_C with S-RULE 0) on seed 0 instances, where we experimented with each of the above orderings. We obtained results for both K_D and K_C. Figure 2.6 plots the log-scale distribution of running time of (K_D, S_{C^*}) for the different ordering strategies compared to `arbitrary`. `push_front` gave the most speedup (10 to 100 times) and did not time out on any instances. We believe this is because singleton vertices from reduced blocks tend to be non-neighbors which helps to quickly detect infeasible assignments. We observed similar behavior with the (K_C, S_{C^*}) combination as shown in Figure 2.7 in Section 2.6.5.

2.6.3 Faster Decomposition

We show results comparing the time taken by `cricca*` and the time taken by `DeCAF` for decomposing all instances reported in Table 2.1. We also do an ablation study and report the effect of the smaller kernel on the running time, with and without search rules.

2.6.3.1 Comparison with `cricca*`. Figure 2.3 gives the ratio of the running time of `cricca*` and the running time of `DeCAF` for different k (lower is better). This includes the

kernelization time as well as the time required for decomposition. For small k, we do not see a significant reduction in runtime but for larger k we are able to obtain up to two orders of magnitude reduction in the running time, and the trend indicates that the reduction in runtime increases with k. Note that we do not include here information about instances for which either `cricca*` or `DeCAF` timeout. In the entire corpus of 2556 instances, 297 instances timed out for `cricca*` and 3 for `DeCAF`. More detailed information about these instances is in Section 2.6.5 (Table 2.2). Note that the order in which the algorithm considers vertices to assign signatures to impacts the running time. In some cases, despite the smaller kernel and `push_front` ordering, the running time can go up. Such instances have a ratio > 1 in Figure 2.3 and are extremely rare.

Figure 2.2 shows the ratio of the number of times the LP solver was invoked by `cricca*` and the number of times the LP solver was invoked by `DeCAF`. Because of the search rules and the smaller kernel, we are able to obtain up to three orders of magnitude reduction in the number of runs of the LP solver. Similar to running time, we see an increasing trend in the reduction in the search space with k.

2.6.3.2 Ablation study. To study the relative impact of the smaller kernel and the search rules on the running time of the decomposition algorithm, we tested all four combinations of the kernels (K_C and K_D) and decomposition algorithms (S_{C^*} and S_D). Here, we only consider the running time of the decomposition algorithms and not the kernelization algorithms and do not include numbers for any instance that timed out across all four combinations. Table 2.1 shows the average reduction obtained in all 4 combinations for different k. The smaller kernel has a greater impact on the runtime compared to that of the search rules. Moreover, as expected, the average reduction increases as k increases.

2.6.4 Scalability

In this section, we evaluate the overall scalability of our implementation. We consider two input k values, (1) $k_{\text{in}} = k$ for yes-instances and (2) $k_{\text{input}} = \lceil 0.8k \rceil$ for no-instances. In Figure 2.4, we compare four configurations, the combinations `cricca*`, `DeCAF` and two k_{in} values for $k \in [5, 6, 7]$ with the instances of seeds $[0, 9]$. We observe that no-instances take longer than yes-instances in both versions. For `cricca*`, the median runtime reaches the time limit (3600 seconds) with $k = 7$ for no-instances. In contrast, `DeCAF` achieves a

median of 13 seconds. For yes-instances, for $k = 7$ there is an order of magnitude difference in the runtimes of the two algorithms.

To test for scalability, we ran `DeCAF` on 12 randomly-selected instances for each $k \in [8, 17]$. Figure 2.5 shows the distribution of running time for the yes and no instances. The median runtime for no-instances with `DeCAF` hits the time limit at $k = 9$. Compare this with the time taken by `cricca*` in Figure 2.4; it already hits the time limit at $k = 7$. Since both `cricca*` and `DeCAF` timeout for $k \geq 9$, we focus on the yes-instances. For yes-instances, `DeCAF` finished within 1000 seconds for all k except for a few "hard" instances with $k = 13, 16$ (this may happen because the worst-case running time is not polynomially bounded). Compare this with Figure 7 in [7] which shows that `cricca` already hits the time limit at $k = 9$. We conclude that with `DeCAF` we are able to scale to at least $1.5\times$ larger k than `cricca` for yes-instances. Details on the number of LP runs are shown in Figures 2.8 and 2.9.

2.6.5 Additional Experimental Results

This section includes supplemental information on instances that timed out, as well as the results of vertex reordering strategies on K_C and the number of LP runs on the corpus of instances from Section 6.4.

2.6.5.1 Timeouts. The number of timeouts with `cricca*` as well as `DeCAF` for different k are given in Table 2.2.

2.6.5.2 Distribution on LP runs when scaling k. In Figures 2.8 and 2.9, we include data on the number of executions of the LP solver on the corpus used in Section 6.4, broken out by yes- and no-instances.

2.6.5.3 Impact of vertex reordering on K_C. We also evaluated all four vertex ordering strategies using kernelized instances produced by the algorithm in [7] using the instances with seed 0. Figure 2.7 presents the results. While none of them significantly out-performed `arbitrary`, we note that `push_front` also did not degrade performance.

2.7 Conclusion

We gave a $k2^k$ kernel for EWCD, an exponential reduction over the best-known approach (a 4^k kernel). It remains open whether a polynomial kernel exists for EWCD. We

also describe new search rules that reduce the decomposition search space for the problem. Our algorithm `DeCAF` achieves two orders of magnitude speedup over the state-of-the-art algorithm, `cricca`; it reduces the number of LP runs by up to three orders of magnitude. Since our approach prunes away large parts of the search space, we are able to scale to solve instances with a larger number of ground-truth cliques (k) than previously possible, though the approach struggles to solve no-instances in this setting. We believe additional rules for quickly detecting infeasibility will help the algorithm to achieve similar scalability when $k_{in} < k$. A natural next step is to consider optimization variants in the non-exact setting.

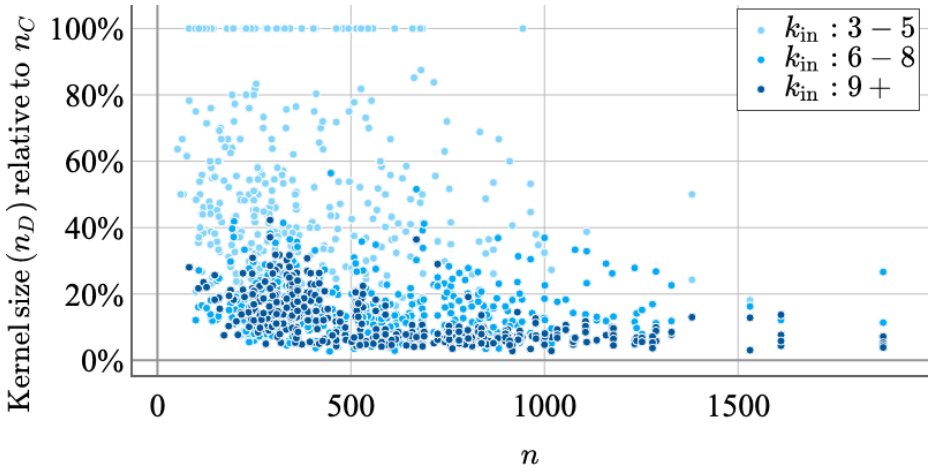


Figure 2.1: Kernel size of K_D relative to that of K_C on entire corpus (seeds 0-19), sorted by instance size (prior to kernelization) and colored by k_{in}.

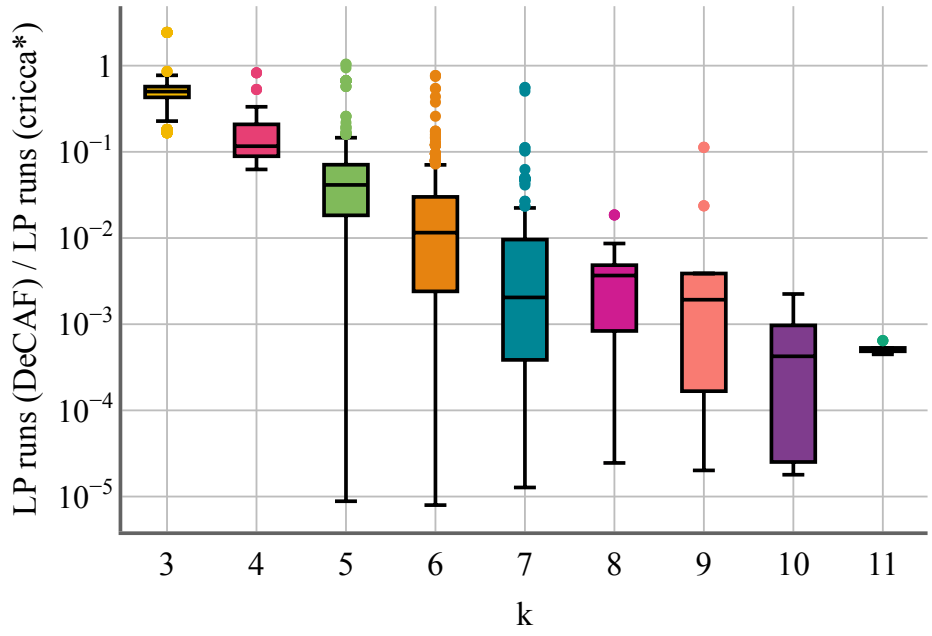


Figure 2.2: `DeCAF` reduces the number of LP runs of `cricca*` by two orders of magnitude. We plot the ratio of number of LP runs on all instances that completed in 3600s with both algorithms. Instances are binned by k, the number of cliques in the desired decomposition, highlighting that the reduction in LP runs increases with the parameter value.

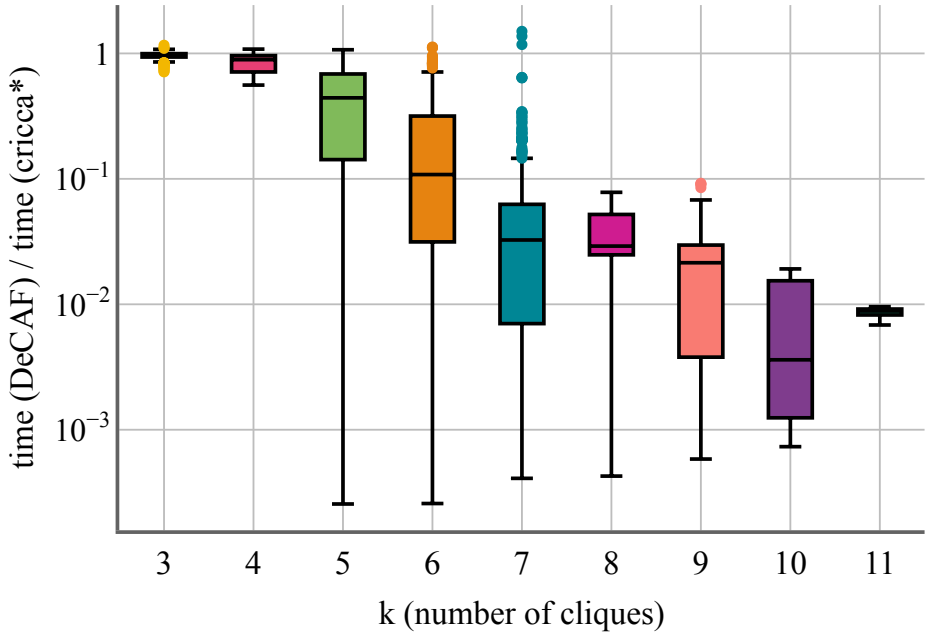


Figure 2.3: `DeCAF` improves the runtime of `cricca*` by two orders of magnitude. We plot the ratio of execution time on all instances that completed in 3600s with both algorithms. Instances are binned by k, the number of cliques in the desired decomposition, highlighting that the improvement in runtime increases with the parameter value.

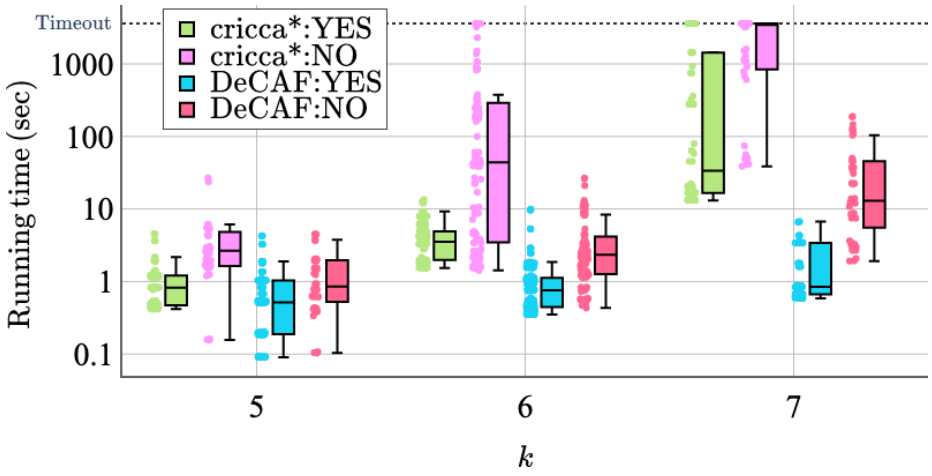


Figure 2.4: Runtime distribution (log-scale) for `cricca*` and `DeCAF` on corpus of YES ($k_{\text{in}} = k$) and NO ($k_{\text{in}} = \lceil 0.8k \rceil$) instances with seeds $[0, 9]$ and $5 \leq k \leq 7$.

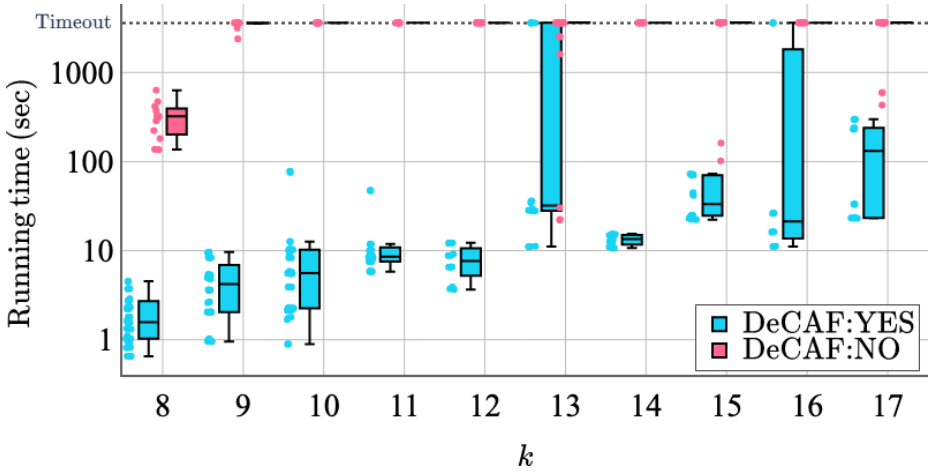


Figure 2.5: Runtime distribution (log-scale) for `DeCAF` on corpus of YES ($k_{\text{in}} = k$) and NO ($k_{\text{in}} = \lceil 0.8k \rceil$) instances with seeds $[0, 9]$ and $8 \leq k \leq 17$.

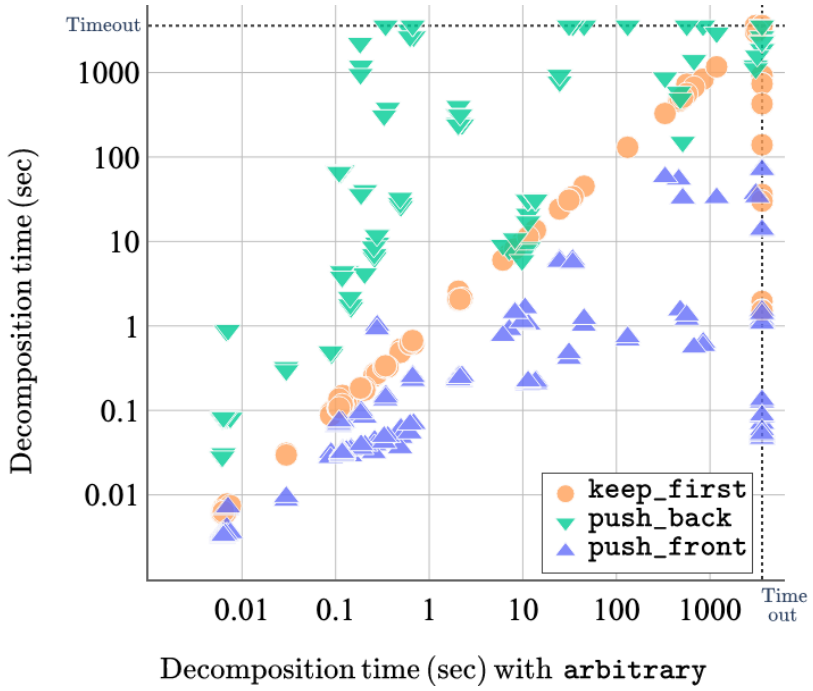


Figure 2.6: Log-log plot of decomposition runtime distribution on instances with seed 0 using different vertex reordering strategies (and kernel K_D). Times relative to `arbitrary`.

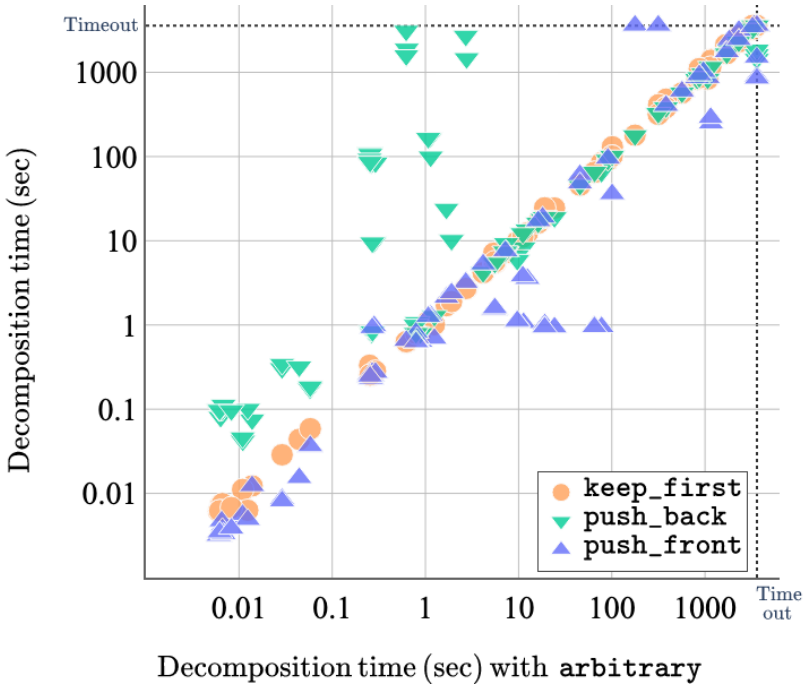


Figure 2.7: Log-scale plot of decomposition running time distribution with different vertex reordering strategies compared to `arbitrary`. Tested with the instances of seed 0, $k \in [3, 11]$ and the original kernels (K_C).

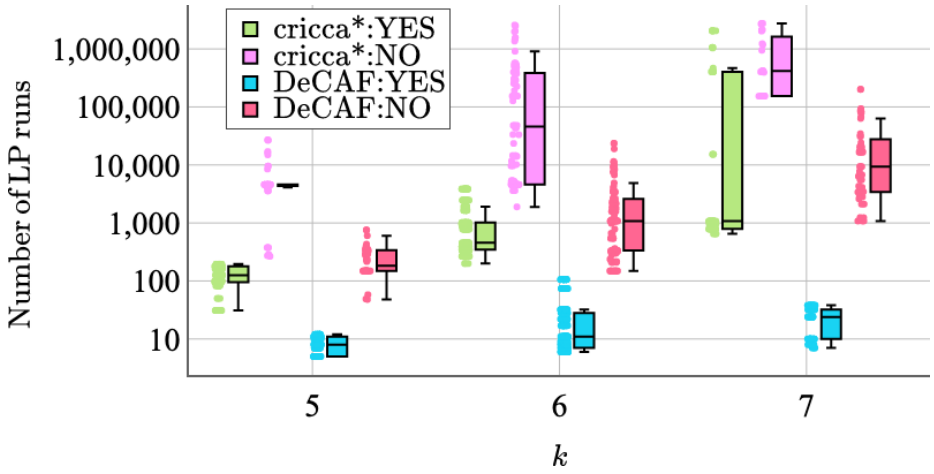


Figure 2.8: Distribution of number of LP runs (log-scale) for `cricca*` and `DeCAF` on corpus of YES ($k_{\text{in}} = k$) and NO ($k_{\text{in}} = \lceil 0.8k \rceil$) instances with seeds $[0, 9]$ and $5 \leq k \leq 7$.

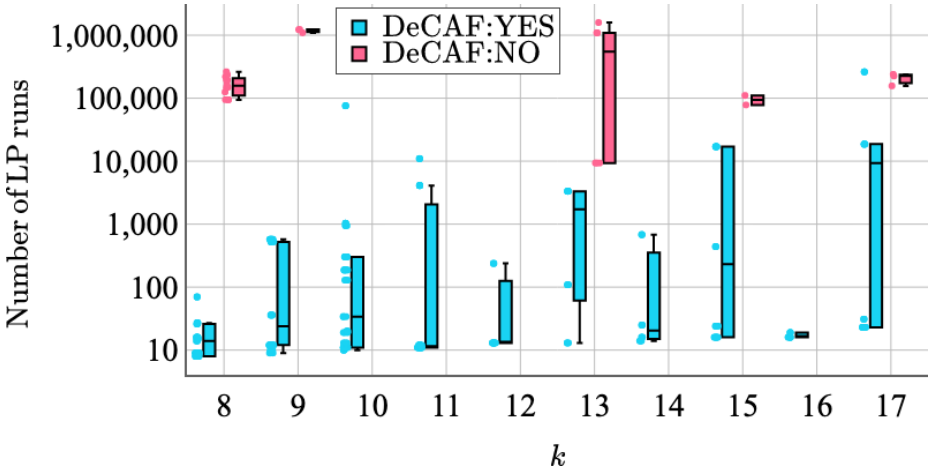


Figure 2.9: Distribution of number of LP runs (log-scale) for `DeCAF` on corpus of YES ($k_{\text{in}} = k$) and NO ($k_{\text{in}} = \lceil 0.8k \rceil$) instances with seeds $[0, 9]$ and $8 \leq k \leq 17$.

Table 2.1: Corpus number of instances and average sizes grouped by k value (seeds $[0, 9]$ and $k \in [3, 11]$). Right columns show average runtime reduction for combinations of kernels K_C, K_D and decomposition algorithms S_{C^*}, S_D.

					K_C		K_D	
					S_{C^*}	S_D	S_{C^*}	S_D
					`cricca*`			`DeCAF`
k	$\#G$	$\#(G, k_{in})$	n	m	speedup			
3	321	321	198.21	38.72	1.00	2.40	2.23	2.45
4	225	225	223.13	146.69	1.00	3.60	4.74	8.81
5	300	564	313.85	2677.86	1.00	6.30	147.91	669.55
6	360	734	369.28	10182.26	1.00	11.83	158.88	1379.24
7	136	274	542.28	28477.22	1.00	13.06	500.44	6061.28
8	31	31	459.77	22834.32	1.00	17.79	169.95	3421.30
9	27	27	824.78	61877.89	1.00	6.45	317.75	9316.60
10	24	24	778.25	116381.50	1.00	22.96	134.39	6876.59
11	18	18	953.00	99219.67	1.00	12.66	128.11	13475.95

Table 2.2: Table showing the number of instances in the main corpus that timed out for `cricca*` and `DeCAF` for $3 \leq k \leq 11$.

k	3	4	5	6	7	8	9	10	11
`cricca*`	0	0	21	46	132	20	30	27	21
`DeCAF`	0	0	0	0	0	0	3	0	0

REFERENCES

[1] L. BABAI AND P. FRANKL, *Linear algebra methods in combinatorics*, University of Chicago, 1988.

[2] F. BAN, V. BHATTIPROLU, K. BRINGMANN, P. KOLEV, E. LEE, AND D. P. WOODRUFF, *A PTAS for ℓ_p-low rank approximation*, in Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, 2019, pp. 747–766.

[3] S. CHANDRAN, D. ISSAC, AND A. KARRENBAUER, *On the parameterized complexity of biclique cover and partition*, in 11th International Symposium on Parameterized and Exact Computation (IPEC 2016), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[4] S. CHEN, Z. SONG, R. TAO, AND R. ZHANG, *Symmetric boolean factor analysis with applications to instahide. arxiv.* preprint, 2021.

[5] M. CHIANG, H. LAM, Z. LIU, AND V. POOR, *Why steiner-tree type algorithms work for community detection*, in Artificial Intelligence and Statistics, PMLR, 2013, pp. 187–195.

[6] L. COLLADO-TORRES, A. NELLORE, K. KAMMERS, S. ELLIS, M. TAUB, K. HANSEN, A. JAFFE, B. LANGMEAD, AND J. LEEK, *Reproducible RNA-seq analysis using recount2*, Nat. Biotechnol., 35 (2017), pp. 319–321.

[7] M. COOLEY, C. S. GREENE, D. ISSAC, M. PIVIDORI, AND B. D. SULLIVAN, *Parameterized algorithms for identifying gene co-expression modules via weighted clique decomposition*, in Proceedings of the 2021 SIAM Conference on Applied and Computational Discrete Algorithms, ACDA 2021, Virtual Conference, July 19-21, 2021, 2021, pp. 111–122.

[8] N. G. DE BRUIJN AND P. ERDÖS, *On a combinatorial problem*, Proceedings of the Section of Sciences of the Koninklijke Nederlandse Akademie van Wetenschappen te Amsterdam, 51 (1948), pp. 1277–1279.

[9] D. DE CAEN AND D. GREGORY, *Partitions of the edge-set of a multigraph by complete subgraphs*, Congressus Numerantium, 47 (1985), pp. 255–263.

[10] C. A. DE LEEUW, J. M. MOOIJ, T. HESKES, AND D. POSTHUMA, *Magma: Generalized gene-set analysis of gwas data*, PLOS Computational Biology, 11 (2015), p. e1004219.

[11] M. DOZMOROV, K. CRESSWELL, S. BACANU, C. CRAVER, M. REIMERS, AND K. S. KENDLER, *A method for estimating coherence of molecular mechanisms in major human disease and traits*, BMC Bioinformatics, 21 (2020), p. 473.

[12] A. E. FELDMANN, D. ISAAC, AND A. RAI, *Fixed-parameter tractability of the weighted edge clique partition problem*, in 15th International Symposium on Parameterized and Exact Computation (IPEC 2020), Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[13] F. V. FOMIN, P. A. GOLOVACH, D. LOKSHTANOV, F. PANOLAN, AND S. SAURABH, *Approximation schemes for low-rank binary matrix approximation problems*, ACM Transactions on Algorithms (TALG), 16 (2019), pp. 1–39.

[14] F. V. FOMIN, P. A. GOLOVACH, AND F. PANOLAN, *Parameterized low-rank binary matrix approximation*, Data Mining and Knowledge Discovery, 34 (2020), pp. 478–532.

[15] C. S. GREENE, A. KRISHNAN, A. K. WONG, E. RICCIOTTI, R. A. ZELAYA, D. S. HIMMELSTEIN, R. ZHANG, B. M. HARTMANN, E. ZASLAVSKY, S. C. SEALFON, ET AL., *Understanding multicellular function and disease with human tissue-specific networks*, Nature genetics, 47 (2015), pp. 569–576.

[16] P. JAIN, S.-T. CHEN, M. AZIMPOURKIVI, D. H. CHAU, AND B. CARBUNAR, *Spotting suspicious reviews via (quasi-) clique extraction*, arXiv preprint arXiv:1509.05935, (2015).

[17] S. JAIN, Y. MIZUTANI, AND B. D. SULLIVAN, *An Exponentially Smaller Kernel for Exact Weighted Clique Decomposition*, in SIAM Conference on Applied and Computational Discrete Algorithms (ACDA23), Proceedings, Society for Industrial and Applied Mathematics, 2023, pp. 119–133.

[18] E. K. KAO, S. T. SMITH, AND E. M. AIROLDI, *Hybrid mixed-membership blockmodel for inference on realistic network interactions*, IEEE Transactions on Network Science and Engineering, 6 (2018), pp. 336–350.

[19] R. KUMAR, R. PANIGRAHY, A. RAHIMI, AND D. WOODRUFF, *Faster algorithms for binary matrix factorization*, in International Conference on Machine Learning, PMLR, 2019, pp. 3551–3559.

[20] A. LACHMANN, D. TORRE, A. B. KEENAN, K. M. JAGODNIK, H. J. LEE, L. WANG, M. C. SILVERSTEIN, AND A. MA'AYAN, *Massive mining of publicly available rna-seq data from human and mouse*, Nature Communications, 9 (2018), p. 1366.

[21] S. MA, W. WALLIS, AND J. WU, *The complexity of the clique partition number problem*, Congr. Numer, 67 (1988), pp. 59–66.

[22] K. N. MAJUMDAR, *On some theorems in combinatorics relating to incomplete block designs*, The Annals of Mathematical Statistics, 24 (1953), pp. 377–389.

[23] W. MAO, E. ZASLAVSKY, B. M. HARTMANN, S. C. SEALFON, AND M. CHIKINA, *Pathway-level information extractor (plier) for gene expression data*, Nature Methods, 16 (2019), pp. 607–610.

[24] S. MASIHULLAH, M. NEGI, J. MATTHEW, AND J. SATHYANARAYANA, *Identifying fraud rings using domain aware weighted community detection*, in International Cross-Domain Conference for Machine Learning and Knowledge Extraction, Springer, 2022, pp. 150–167.

[25] R. MATHEW AND T. K. MISHRA, *A combinatorial proof of fisher's inequality*, Graphs and Combinatorics, 36 (2020), pp. 1953–1956.

[26] J. MENCHE, A. SHARMA, M. KITSAK, S. D. GHIASSIAN, M. VIDAL, J. LOSCALZO, AND A.-L. BARABÁSI, *Uncovering disease-disease relationships through the incomplete interactome*, Science, 347 (2015), p. 1257601.

[27] D. MERCATELLI, L. SCALAMBRA, L. TRIBOLI, F. RAY, AND F. M. GIORGI, *Gene regulatory network inference resources: A practical overview*, Biochimica et Biophysica Acta (BBA) - Gene Regulatory Mechanisms, 1863 (2020), p. 194430.

[28] P. MIETTINEN AND S. NEUMANN, *Recent developments in boolean matrix factorization.* preprint, 2020.

[29] F. MOUTIER, A. VANDAELE, AND N. GILLIS, *Off-diagonal symmetric nonnegative matrix factorization*, Numerical Algorithms, pp, (2021), pp. 1–25.

[30] E. MUJUNI AND F. ROSAMOND, *Parameterized complexity of the clique partition problem*, in Proceedings of the fourteenth symposium on Computing: the Australasian theory-Volume 77, 2008, pp. 75–78.

[31] M. R. NELSON, H. TIPNEY, J. L. PAINTER, J. SHEN, P. NICOLETTI, Y. SHEN, A. FLORATOS, P. C. SHAM, M. J. LI, J. WANG, ET AL., *The support of human genetic evidence for approved drug indications*, Nature genetics, 47 (2015), pp. 856–860.

[32] H. A. SOUFIANI AND E. AIROLDI, *Graphlet decomposition of a weighted network*, in Artificial Intelligence and Statistics, PMLR, 2012, pp. 54–63.

[33] D. R. STINSON, *Introduction to balanced incomplete block designs*, Combinatorial Designs: Constructions and Analysis, (2004), pp. 1–21.

[34] J. N. TARONI, P. C. GRAYSON, Q. HU, S. EDDY, M. KRETZLER, P. A. MERKEL, AND C. S. GREENE, *Multiplier: A transfer learning framework for transcriptomics reveals systemic features of rare disease*, Cell Systems, 8 (2019), pp. 380–394.e4.

[35] P. M. VAIDYA, *Speeding-up linear programming using fast matrix multiplication*, in 30th Annual Symposium on Foundations of Computer Science, 1989, pp. 332–337.

[36] K. VENKATESAN, J.-F. RUAL, A. VAZQUEZ, U. STELZL, I. LEMMENS, T. HIROZANE-KISHIKAWA, T. HAO, M. ZENKNER, X. XIN, K.-I. GOH, ET AL., *An empirical framework for binary interactome mapping*, Nature methods, 6 (2009), pp. 83–90.

[37] P. M. VISSCHER, N. R. WRAY, Q. ZHANG, P. SKLAR, M. I. MCCARTHY, M. A. BROWN, AND J. YANG, *10 years of GWAS discovery: Biology, function, and translation*, Am. J. Hum. Genet., 101 (2017), pp. 5–22.

[38] X. XIAO, A. MORENO-MORAL, M. ROTIVAL, L. BOTTOLO, AND E. PETRETTO, *Multi-tissue analysis of co-expression networks by higher-order generalized singular value decomposition identifies functionally coherent transcriptional modules*, PLoS genetics, 10 (2014), p. e1004006.

[39] X. YAN, M. MEHAN, Y. HUANG, M. WATERMAN, P. YU, AND X. J. ZHOU, *A graph-based approach to systematically reconstruct human transcriptional regulatory modules*, Bioinformatics, 23 (2007), pp. i577–i586.

[40] Z. Y. ZHANG, Y. WANG, AND Y. Y. AHN, *Overlapping community detection in complex networks using symmetric binary matrix factorization*, Physical Review E, 87 (2013), p. 6.

CHAPTER 3

REDUCING THE SEARCH SPACE FOR ODD CYCLE TRANSVERSAL

In the previous chapter, we saw an example of kernels effectively reducing the search space. In this chapter, we study an alternative notion of preprocessing that is applicable even when the solution size is not small. Donkers and Jansen introduced *antler decompositions* for FEEDBACK VERTEX SET and established a framework of preprocessing "beyond kernelization" [5]. Collaborating with Bart M. P. Jansen[1], Blair D. Sullivan[2], and Ruben Verhaegh[1], we extended their idea to another problem, ODD CYCLE TRANSVERSAL. We presented our new results at the International Symposium on Parameterized and Exact Computation (IPEC) '24 and published the paper "Preprocessing to Reduce the Search Space for Odd Cycle Transversal" [14]. I was in charge of establishing technical results and writing the proof of about the half of key lemmas/theorems. For statements marked (⧫), the proof is mostly contributed by coauthors and not included in this dissertation. Please refer to the full paper for details.

3.1 Introduction

The NP-hard ODD CYCLE TRANSVERSAL problem asks for a minimum vertex set whose removal from an undirected input graph G breaks all odd cycles, and thereby yields a bipartite graph. Finding odd cycle transversals has important applications, for example in computational biology [10, 23] and adiabatic quantum computing [8, 9]. ODD CYCLE TRANSVERSAL parameterized by the desired solution size k has been studied intensively, leading to important advances such as *iterative compression* [21] and *matroid-based kernelization* [16, 17]. The randomized kernel due to Kratsch and Wahlström [17, Lemma 7.11]

[1]Eindhoven University of Technology, Netherlands.

[2]University of Utah, USA.

is a polynomial-time algorithm that reduces an n-vertex instance (G, k) of ODD CYCLE TRANSVERSAL to an instance (G', k') on $\mathcal{O}((k \log k \log\log k)^3)$ vertices, that is equivalent to the input instance with probability at least 2^{-n}. Experiments with this matroid-based kernelization, however, show disappointing preprocessing results in practice [20]. This formed one of the motivations for a recent line of research aimed at preprocessing that reduces the *search space* explored by algorithms solving the reduced instance, rather than preprocessing aimed at reducing the *encoding size* of the instance (which is captured by kernelization). To motivate our work, we present some background on this topic.

A *kernelization* of size $f : \mathbb{N} \to \mathbb{N}$ for a parameterized problem $\mathcal{P}$ is a polynomial-time algorithm that reduces any parameterized instance (I, k) to an instance (I', k') with the same YES/NO answer, such that $|I'|, k' \leq f(k)$. It therefore guarantees that the size of the instance is reduced in terms of the complexity parameter k. It does not directly ensure a reduction in the *search space* of the follow-up algorithm that is employed to solve the reduced instance. Since the running times of FPT algorithms for the natural parameterization of ODD CYCLE TRANSVERSAL [10, 21, 18] depend exponentially on the size of the sought solution, the size of the search space considered by such algorithms can be reduced significantly by a preprocessing step that finds some vertices S that belong to an optimal solution for the input graph G: the search for a solution of size k on G then reduces to the search for a solution of size $k - |S|$ on $G - S$. Researchers therefore started to investigate in which situations an efficient preprocessing phase can guarantee finding part of an optimal solution.

One line of inquiry in this direction aims at finding vertices that not only belong to an optimal solution, but are even required for building a c-approximate solution [2, 15]; such vertices are called *c-essential*. This has resulted in refined running time guarantees, showing that an optimal odd cycle transversal of size k can be found in time $2.3146^{k-\ell} \cdot n^{\mathcal{O}(1)}$, where ℓ is the number of vertices in the instance that are essential for making a 3-approximate solution [2]. Another line of research, more relevant to the subject of this chapter, aims at finding vertices that belong to an optimal solution when there is a simple, locally verifiable certificate of the existence of an optimal solution containing them. So far, the latter direction has been explored for VERTEX COVER (where a *crown decomposition* [1, 7] forms such a certificate), and for the (undirected) FEEDBACK VERTEX SET problem (where

an *antler decomposition* [5]) forms such a certificate.

A *crown decomposition* (see Figure 3.1) of a graph G consists of a partition of its vertex set into three parts: the *crown* I (which is required to be a non-empty independent set), the *head* H (which is required to contain all neighbors of I), and the *remainder* $R = V(G) \setminus (I \cup H)$, such that the graph $G[I \cup H]$ contains a matching M of size $|H|$. Since I is an independent set, this matching partners each vertex of H with a private neighbor in I. The existence of a crown decomposition shows that there is an optimal vertex cover (a minimum-size vertex set intersecting all edges) that contains all vertices of H and none of I: any vertex cover contains at least $|M| = |H|$ vertices from $I \cup H$ to cover the matching M, while H covers all the edges of G that can be covered by selecting vertices from $I \cup H$. Hence a crown decomposition forms a polynomial-time verifiable certificate that there is an optimal vertex cover containing all vertices of H. It facilitates a reduction in search space for VERTEX COVER: graph G has a vertex cover of size k if and only if $G - (I \cup H)$ has one of size $k - |H|$. A crown decomposition can be found in polynomial time if it exists, which yields a powerful reduction rule for VERTEX COVER [1].

Inspired by this decomposition for VERTEX COVER, Donkers and Jansen [5] introduced the notion of an *antler decomposition* of a graph G. It is a partition of the vertex set into three parts: the *antler* A (which is required to induce a non-empty acyclic graph), the *head* H (which is required to contain *almost* all neighbors of A: for each tree T in the forest $G[A]$, there is at most one edge that connects T to a vertex outside H), and the *remainder* $R = V(G) \setminus (A \cup H)$, while satisfying an additional condition in terms of an integer z that represents the *order* of the antler decomposition. In its simplest form for $z = 1$ (we discuss $z > 1$ later), the additional condition says that the graph $G[A \cup H]$ should contain $|H|$ vertex-disjoint cycles. Since $G[A]$ is acyclic, each of these cycles contains exactly one vertex of H. They certify that any feedback vertex set of G contains at least $|H|$ vertices from $A \cup H$. Since A induces an acyclic graph, and all cycles in G that enter a tree T of $G[A]$ from R must leave A from H, the set H intersects all cycles of G that contain a vertex of $A \cup H$. Hence there is an optimal feedback vertex set containing H. By finding an antler decomposition we can therefore reduce the problem of finding a size-k solution in G to finding a size-$(k - |H|)$ solution in $G - (A \cup H)$, and therefore reduce the search space for algorithms parameterized by solution size.

Donkers and Jansen proved that, assuming P≠NP, there unfortunately is no polynomial-time algorithm to find an antler decomposition if one exists [5, Theorem 3.4]. However, they gave a *fixed-parameter tractable* preprocessing algorithm, parameterized by the size of the head. There is an algorithm that, given a graph G and integer k such that G contains an antler decomposition (A, H, R) with $|H| = k$, runs in time $2^{\mathcal{O}(k^5)} \cdot n^{\mathcal{O}(1)}$ and outputs a set of at least k vertices that belong to an optimal feedback vertex set. For each fixed value of k, this yields a preprocessing algorithm to detect vertices that belong to an optimal solution if there is a simple certificate of their membership in an optimal solution.

In fact, Donkers and Jansen gave a more general algorithm; this is where z-antlers for $z > 1$ make an appearance. Recall that for a 1-antler decomposition (A, H, R) of a graph G, the graph $G[A \cup H]$ must contain a collection $\mathcal{C}$ of $|H|$ vertex-disjoint cycles. These cycles certify that the set H is an optimal feedback vertex set in the graph $G[A \cup H]$. In fact, the feedback vertex set H in $G[A \cup H]$ is already optimal for the subgraph $\mathcal{C} \subseteq G[A \cup H]$, and that subgraph $\mathcal{C}$ is structurally simple because each of its connected components (which is a cycle) has a feedback vertex set of size $z = 1$. This motivates the following definition of a z-antler decomposition for $z > 1$: the set H should be an optimal feedback vertex set for the subgraph $G[A \cup H]$, and moreover, there should be a subgraph $\mathcal{C}_z \subseteq G[A \cup H]$ such that (1) H is an optimal feedback vertex set in $\mathcal{C}_z$, and (2) each connected component of $\mathcal{C}_z$ has a feedback vertex set of size at most z. So for a z-antler decomposition (A, H, R) of a graph G, there is a certificate that H is part of an optimal solution in the overall graph G that consists of the decomposition together with the subgraph $\mathcal{C}_z \subseteq G[A \cup H]$ for which H is an optimal solution. The complexity of verifying this certificate scales with z: it comes down to verifying that $H \cap V(C)$ is indeed an optimal feedback vertex set of size at most z for each connected component of the subgraph $\mathcal{C}_z$. Donkers and Jansen presented an algorithm that, given integers $k \geq z \geq 0$ and a graph G that contains a z-antler decomposition whose head has size k, outputs a set of at least k vertices that belongs to an optimal feedback vertex set in time $2^{\mathcal{O}(k^5 z^2)} n^{\mathcal{O}(z)}$. For each fixed choice of k and z, this gives a reduction rule (that can potentially be applied numerous times on an instance) to reduce the search space if the preconditions are met.

3.1.0.1 Our contribution. We investigate search-space reduction algorithms for ODD CYCLE TRANSVERSAL, thereby continuing the line of research by Donkers and

Jansen [5]. We introduce the notion of *tight odd cycle cuts* to provide efficiently verifiable witnesses that a certain vertex set belongs to an optimal odd cycle transversal, and present algorithms to find vertices that belong to an optimal solution in inputs that admit such witnesses.

To be able to state our main result, we introduce the corresponding terminology. An *odd cycle cut* (OCC) in an undirected graph G is a partition of its vertex set into three parts: the bipartite part B (which is required to induce a bipartite subgraph of G), the cut part C (which is required to contain all neighbors of B), and the rest $R = V(G) \setminus (B \cup C)$. An odd cycle cut is called *tight* if the set C forms an optimal odd cycle transversal for the graph $G[B \cup C]$. In this case, it is easy to see that there is an optimal odd cycle transversal in G that contains all vertices of C, since all odd cycles through B are intersected by C. A tight OCC (B, C, R) has *order* z if there is a subgraph $\mathcal{C}_z$ of $G[B \cup C]$ for which C is an optimal odd cycle transversal, and for which each connected component of $\mathcal{C}_z$ has an odd cycle transversal of size at most z. This means that for $z = 1$, if there is such a subgraph $\mathcal{C}_z \subseteq G[B \cup C]$, then there is one consisting of $|C|$ vertex-disjoint odd cycles. We use the term z-tight OCC to refer to a tight OCC of order z. Our notion of z-tight OCCs forms an analogue of z-antler decompositions. Note that the requirement that C contains *all* neighbors of B is slightly more restrictive than in the FEEDBACK VERTEX SET case. We need this restriction for technical reasons, but discuss potential relaxations in Section 3.8.

Similarly to the setting of z-antlers for FEEDBACK VERTEX SET, assuming P≠NP there is no polynomial-time algorithm that always finds a tight OCC in a graph if one exists; not even in the case $z = 1$ (Theorem 3.36). We therefore develop algorithms that are efficient for small k and z. The following theorem captures our main result, which is an OCT-analogue of the antler-based preprocessing algorithm for FVS. The *width* of an OCC (B, C, R) is defined as $|C|$. Our theorem shows that for constant z we can efficiently find k vertices that belong to an optimal solution, if there is a z-tight OCC of width k.

Theorem 3.1. *There is a deterministic algorithm that, given a graph G and integers $k \geq z \geq 0$, runs in $2^{\mathcal{O}(k^{33}z^2)} \cdot n^{\mathcal{O}(z)}$ time and either outputs at least k vertices that belong to an optimal solution for* ODD CYCLE TRANSVERSAL, *or concludes that G does not contain a z-tight OCC of width k.*

One may wonder whether it is feasible to have more control over the output, by having

the algorithm output a z-tight OCC (B, C, R) of width k, if one exists. However, a small adaptation of a W[1]-hardness proof for antlers [5, Theorem 3.7] shows (Theorem 3.37) that the corresponding algorithmic task is W[1]-hard even for $z = 1$. This explains why the algorithm outputs a vertex set that belongs to an optimal solution, rather than a z-tight OCC.

In terms of techniques, our algorithm combines insights from the previous work on antlers [5] with ideas in the representative-set based kernelization [17] for ODD CYCLE TRANSVERSAL. The global idea behind the algorithm is to repeatedly simplify the graph, while preserving the structure of z-tight OCCs, to arrive at the following favorable situation: if there was a z-tight OCC of width k in the input, then the reduced graph has a z-tight OCC (B, C, R) of the same width that satisfies $|B| \in k^{\mathcal{O}(1)}$. At that point, we can use color coding with a set of $k^{\mathcal{O}(1)}$ colors to ensure that the structure $B \cup C$ gets colored in a way that makes it tractable to identify it. The simplification steps on the graph are inspired by the kernelization for ODD CYCLE TRANSVERSAL and involve the computation of a *cut covering set* of size $k^{\mathcal{O}(1)}$ that contains a minimum three-way $\{X, Y, Z\}$-separator for all possible choices of sets $\{X, Y, Z\}$ drawn from a terminal set T of size $k^{\mathcal{O}(1)}$. The existence of such sets follows from the matroid-based tools of Kratsch and Wahlström [17]. We can avoid the randomization incurred by their polynomial-time algorithm by computing a cut covering set in $2^{\mathcal{O}(k)} \cdot n^{\mathcal{O}(1)}$ time deterministically. Compared to the kernelization for ODD CYCLE TRANSVERSAL, a significant additional challenge we face in this setting is that the size of OCTs in the graph can be arbitrarily large in terms of the parameter k. Our algorithm is looking for a *small* region of the graph in which a vertex set exists with a simple certificate for its membership in an optimal solution; it cannot afford to learn the structure of global OCTs in the graph. This local perspective poses a challenge when repeatedly simplifying the graph: we not only have to be careful how these operations affect the total solution size in G, but also how these modifications affect the existence of simple certificates for membership in an optimal solution. This is why our reduction step works with three-way separators, rather than the two-way separators that suffice to solve or kernelize OCT.

3.1.0.2 Organization. The remainder of this work is organized as follows. The first twelve pages of the manuscript present the key statements and ideas. After presenting preliminaries on graphs in Section 3.2, we define (tight) OCCs in Section 3.3 and explore

some of their properties. In Section 3.4 we show how color coding can be used to find an OCC whose bipartite part is connected and significantly larger than its cut. Given such an OCC, we show in Section 3.5 how to simplify the graph while preserving the essential structure of odd cycles in the graph. This leads to an algorithm that finds vertices belonging to an optimal solution the presence of a tight OCC in Section 3.6. In Section 3.7 we give the hardness proofs mentioned above. Finally, we conclude in Section 3.8.

3.2 Preliminaries

Refer to Section 1.3.1 for graph-theoretic definitions and notation.

The *parity* of a path or cycle refers to the parity of its length. For a walk $W = (v_1, \ldots, v_k)$, we refer to its vertex set as $V(W) = \{v_1, \ldots, v_k\}$. Observe that if W is a closed walk of odd parity (a *closed odd walk*), then the graph $G[V(W)]$ contains a cycle of odd length (an *odd cycle*): any edge connecting two vertices of $V(W)$ that are not consecutive on W splits the walk into two closed subwalks, one of which has odd length.

For a positive integer q, a *proper q-coloring* of a graph G is a function $f\colon V(G) \to \{0, \ldots, q-1\}$ such that $f(u) \neq f(v)$ for all $uv \in E(G)$. A graph G is *bipartite* if its vertex set can be partitioned into two *partite sets* $L \dot\cup R$ such that no edge has both of its endpoints in the same partite set. It is well-known that the following three conditions are equivalent for any graph G: (1) G is bipartite, (2) G admits a proper 2-coloring, and (3) there is no cycle of odd length in G. An *odd cycle transversal* (OCT) of a graph G is a set $S \subseteq V(G)$ such that $G - S$ is bipartite. An *independent set* is a vertex set S such that $G[S]$ is edgeless. We say that a vertex set X in a graph G *separates* two (not necessarily) disjoint vertex sets S and T if no connected component of $G - X$ simultaneously contains a vertex from S and a vertex from T. For a collection $\{T_1, \ldots, T_m\}$ of (not necessarily disjoint) vertex sets in a graph G, we say that a vertex set X is an $\{T_1, \ldots, T_m\}$*-separator* if X separates all pairs (T_i, T_j) for $i \neq j$. Note that X is allowed to intersect $\bigcup_{i \in [m]} T_i$.

The following lemma captures the main idea behind the iterative compression algorithm [21] (cf. [3, §4.4]) for solving Odd Cycle Transversal. Given a (potentially suboptimal) odd cycle transversal W of a graph, it shows that the task of finding an odd cycle transversal disjoint from W whose removal leaves a bipartite graph with $W_0, W_1 \subseteq W$ in opposite partite sets of its bipartition is equivalent to separating two vertex sets derived

from a baseline bipartition of $G - W$. Our statement below is implied by Claim 1 in the work of Jansen and de Kroon [11].

Lemma 3.2 ([11, Claim 1])**.** *Let W be an OCT in graph G. For each partition of $W = W_0 \cup W_1$ into two independent sets, for each proper 2-coloring c of $G - W$, we have the following equivalence for each $X \subseteq V(G) \setminus W$: the graph $G - X$ has a proper 2-coloring with W_0 color 0 and W_1 color 1* if and only if *the set X separates A from R in the graph $G - W$, with:*

$$A = (N_G(W_0) \cap c^{-1}(0)) \cup (N_G(W_1) \cap c^{-1}(1)),$$
$$R = (N_G(W_0) \cap c^{-1}(1)) \cup (N_G(W_1) \cap c^{-1}(0)).$$

The next lemma gives a simple sufficient condition for a graph to be bipartite.

Lemma 3.3. *Let G be a graph and let $V_L \cup V_0 \cup V_R = V(G)$ be a partition of its vertices such that V_0 is a $\{V_L, V_R\}$-separator. If there exist proper 2-colorings $f_L : (V_0 \cup V_L) \to \{0,1\}$ and $f_R : (V_0 \cup V_R) \to \{0,1\}$ of $G[V_0 \cup V_L]$ and $G[V_0 \cup V_R]$ respectively such that $f_L(v_0) = f_R(v_0)$ for every $v_0 \in V_0$, then G is bipartite.*

3.2.1 Multiway Cuts

We introduce some terminology about multiway cuts. Let $\mathcal{T} = (T_1, \ldots, T_s)$ be a partition of a set $T \subseteq V(G)$ of *terminal* vertices in an undirected graph G. A *multiway cut* of $\mathcal{T}$ in G is a vertex set $X \subseteq V(G)$ such that for each pair $t_i, t_j \in T \setminus X$ that belong to different parts of partition $\mathcal{T}$, the graph $G - X$ does not contain a path from t_i to t_j. A *restricted* multiway cut of $\mathcal{T}$ is a vertex set X that is a multiway cut for $\mathcal{T}$ such that $X \cap T = \emptyset$, i.e., it does not contain any terminals. The parameterized problem of computing a restricted multiway cut is well-known to be fixed-parameter tractable parameterized by solution size [4].

Theorem 3.4 ([4, Corollary 2.6])**.** *There is an algorithm that, given an undirected n-vertex graph G, terminal set $T \subseteq V(G)$, and integer k, runs in time $2^{\mathcal{O}(k)} \cdot n^{\mathcal{O}(1)}$ and either outputs the minimum size of a vertex set $U \subseteq V(G) \setminus T$ for which all vertices of T belong to a different connected component of $G - U$, or determines that any such multiway cut has size larger than k.*

While the original formulation of the theorem is stated in terms of simply solving the decision problem (is there a solution of size at most k?), by running this algorithm for

all possible k' in the range of $\{0, \ldots, k\}$ we can identify the size of an optimal solution with only a polynomial factor overhead. Next, we show that with a small modification we can use the same algorithm to compute the size of a minimum multiway cut of an s-partition $\mathcal{T} = (T_1, \ldots, T_s)$ of the terminal set T (or determine its size is larger than k). To express the necessary modification on the graph, we need the following concept. The operation of *identifying* a vertex set X in an undirected graph G yields the graph G' obtained from G by removing X while inserting a single new vertex x with $N_{G'}(x) = N_G(X)$.

Observation 3.5. *Let G be an undirected graph with terminal set T and let $\mathcal{T} = (T_1, \ldots, T_s)$ be an s-partition of T. The following two conditions are equivalent for each vertex set $U \subseteq V(G) \setminus T$:*

- *The set U is a multiway cut of the partition $\mathcal{T}$ in G;*
- *No connected component of $G' - U$ contains more than one vertex of T', where G' is the graph obtained from G by identifying each non-empty vertex set T_i for $i \in [s]$ into a single vertex t_i, and T' is the set containing the vertices t_i resulting from these identifications. (The order in which the sets are identified does not affect the final result.)*

The forward implication is trivial; the converse follows from the fact that any path between two distinct vertices of T' contains a subpath connecting vertices belonging to two different sets of the partition $\mathcal{T}$, which must therefore be broken by any multiway cut for the partition $\mathcal{T}$ in G.

Lemma 3.6. *There is an algorithm that, given an undirected n-vertex graph G, an integer s, an s-partition $\mathcal{T} = (T_1, \ldots, T_s)$ of a terminal set $T \subseteq V(G)$, and integer k, runs in time $2^{\mathcal{O}(k)} \cdot n^{\mathcal{O}(1)}$ and either outputs the minimum size of a restricted multiway cut for $\mathcal{T}$, or outputs $\perp$ to indicate that any such restricted multiway cut has size larger than k.*

Proof. Apply the algorithm of Theorem 3.4 to the graph G' and terminal set T' obtained by identifying each set T_i into a new terminal vertex $t_i \in T'$, with a budget of k. Correctness follows from Observation 3.5. □

We will use Lemma 3.6 to construct a *deterministic* FPT-algorithm to compute a cut covering set later. For a vertex set U in an undirected graph G, the operation of *bypassing* vertex set U results in the graph $\mathsf{bypass}_U(G)$ that is obtained as follows: starting from $G -$

U, add an edge uv between every pair of distinct vertices u, v which appear together in the same connected component of $G[U]$? Hence $\mathsf{bypass}_U(G)$ is a graph on vertex set $V(G) \setminus U$ that has an edge uv whenever $uv \in E(G)$ or there is a path from u to v for which all internal vertices belong to U.

Observation 3.7. *Let G be an undirected graph with terminal set $T \subseteq V(G)$, and let $(T_1, \ldots, T_s)$ be a partition of T. For any set $R \subseteq V(G) \setminus T$ of bypassed vertices and any vertex set $U \subseteq V(G) \setminus R$, the set U is a restricted multiway cut of $(T_1, \ldots, T_s)$ in G if and only if U is a restricted multiway cut of $(T_1, \ldots, T_s)$ in $\mathsf{bypass}_R(G)$.*

This observation implies that bypassing a set of non-terminal vertices can only make it *harder* to form a multiway cut.

3.2.2 Covering Multiway Cuts for Generalized Partitions

For a positive integer s, a *generalized s-partition* of a set T is a partition $\mathcal{T}^* = (T_0, T_1, \ldots, T_s, T_X)$ of T into $s+2$ parts, some of which can be empty. The parts T_0 and T_X play a special role, which are the *free* and *deleted* part of $\mathcal{T}^*$, respectively. Let $T' = T_1 \cup \ldots \cup T_s$. A *multiway cut* of $\mathcal{T}^*$ is a (non-restricted) multiway cut in $G - T_X$ of the partition $\mathcal{T} = (T_1, \ldots, T_s)$ of T'. Hence the vertices of T_X are deleted from the graph, while no cut constraints are imposed on the vertices of T_0.

A *minimum multiway cut* of a generalized s-partition $\mathcal{T}^*$ in a graph G is a minimum-cardinality vertex set that satisfies the requirements of a multiway cut for $\mathcal{T}^*$. We denote the size of a minimum multiway cut of $\mathcal{T}^*$ in G by $\mathsf{MwCut}(G, \mathcal{T}^*)$. The following cut covering lemma by Kratsch and Wahlström will be useful for our algorithm.

Theorem 3.8 ([17, Theorem 5.14])**.** *Let G be an undirected graph on n vertices with a set $T \subseteq V(G)$ of terminal vertices, and let $s \in \mathbb{N}$ be a constant. There is a set $Z \subseteq V(G)$ with $|Z| = \mathcal{O}(|T|^{s+1})$ such that Z contains a minimum multiway cut of every generalized s-partition $\mathcal{T}^*$ of T, and we can compute such a set in randomized polynomial time with failure probability $\mathcal{O}(2^{-n})$.*

For a generalized s-partition $\mathcal{T} = (T_0, T_1, \ldots, T_s, T_X)$ of a terminal set $T \subseteq V(G)$ in an undirected graph G, we call a multiway cut X of $\mathcal{T}$ *restricted* if it satisfies $X \cap (\bigcup_{i=1}^{s} T_i) = \emptyset$. Hence a restricted multiway cut does not delete any vertex that is active as a terminal in

the generalized partition. A minimum *restricted* multiway cut of $\mathcal{T}$ is a restricted multiway cut whose size is minimum among all restricted multiway cuts. We denote the minimum size of a *restricted* multiway cut of $\mathcal{T}$ in G by $\mathsf{ResMwCut}(G, \mathcal{T})$, which we define as $+\infty$ if no such cut exists.

The following lemma shows that the randomization in the polynomial-time algorithm by Kratsch and Wahlström can be avoided by the use of a single-exponential FPT algorithm, and that the cut covering set can be adapted to work for *restricted* multiway cuts as long as we have a bound on their size.

Lemma 3.9 (⧫). *Let $s \in \mathbb{N}$ be a constant. There is a* deterministic *algorithm that, given an undirected n-vertex graph G and a set $T \subseteq V(G)$ of terminals, runs in time $2^{\mathcal{O}(|T|)} \cdot n^{\mathcal{O}(1)}$ and computes a set $Z \subseteq V(G)$ with $|Z| = \mathcal{O}(|T|^{2s+2})$ with the following guarantee: for each generalized s-partition $\mathcal{T}$ of T, if there is a restricted multiway cut for $\mathcal{T}$ of size at most $|T|$ in G, then the set Z contains a* minimum *restricted multiway cut of $\mathcal{T}$.*

3.2.3 Universal Sets: The Deterministic Version of Color Coding

For a set D of size n and integer k with $n \geq k$, an (n,k)-*universal set* for D is a family $\mathcal{U}$ of subsets of D such that for all $S \subseteq D$ of size at most k we have $\{S \cap U \mid U \in \mathcal{U}\} = 2^S$.

Theorem 3.10 ([19, Theorem 6], cf. [3, Theorem 5.20])**.** *For any set D and integers n and k with $|D| = n \geq k$, an (n,k)-universal set $\mathcal{U}$ for D with $|\mathcal{U}| = 2^{\mathcal{O}(k)} \log n$ can be created in $2^{\mathcal{O}(k)} n \log n$ time.*

We extend the notion of universal set to the following notion of *universal function family*. For sets A of size n and B of size q and an integer k with $n \geq k$, an (n,k,q)-*universal function family* is a family $\mathcal{F}$ of functions $A \to B$ such that for all $S \subseteq A$ of size at most k, the family $\{f|_S \colon f \in \mathcal{F}\}$ contains all q^k functions from A to B. Here we write $f|_S$ for the restriction of f to domain S.

Corollary 3.11. *There is an algorithm that, given sets A, B with $n = |A|$, $q = |B|$, and a positive integer $k \leq n$, constructs an (n,k,q)-universal function family $\mathcal{F}$ of size $2^{\mathcal{O}(kq')} \log^{q'} n$ in $2^{\mathcal{O}(kq')} nq' \log^{q'} n$ time, where $q' = \lceil \log_2 q \rceil$. In particular, when q is a constant, both the size and construction time are $2^{\mathcal{O}(k)} n^{\mathcal{O}(1)}$.*

Proof. Let $q' = \lceil \log_2 q \rceil$. The algorithm starts by computing an (n,k)-universal set for A in $2^{\mathcal{O}(k)} n \log n$ time via Theorem 3.10. We conceptually create q' copies of this universal set, denoted $\mathcal{U}_i$ for $i \in [q']$, where each $\mathcal{U}_i$ has size $2^{\mathcal{O}(k)} \log n$. Now consider the Cartesian product $\mathcal{U}_\times := \mathcal{U}_1 \times \ldots \times \mathcal{U}_{q'}$. For a set $X \subseteq A$ we denote its indicator function by $I_X \colon A \to \{0,1\}$, so that $I_X(x) = 1$ if $x \in X$ and $I_X(x) = 0$ if $x \notin X$. Also, let $g \colon \{0,1\}^{q'} \to [2^{q'}]$ be a function that converts binary representations of integers between 0 and $(2^{q'} - 1)$ into integers in the range $\{1, \ldots, 2^{q'}\}$; note that $2^{q'} \geq q$. Finally, define an arbitrary surjective function $h : [2^{q'}] \to B$. Now, we construct a family of functions $\mathcal{F}$ as follows: for each $(U_1, \ldots, U_{q'}) \in \mathcal{U}_\times$, create a function $f \colon A \to B$ such that $f(x) = h(g(I_{U_1}(x), \ldots, I_{U_{q'}}(x)))$. It is straightforward to see that $|\mathcal{F}| \leq |\mathcal{U}_\times| \leq |\mathcal{U}_1|^{q'} \leq 2^{\mathcal{O}(kq')} \log^{q'} n$. This construction can be done in $\mathcal{O}(|\mathcal{F}| \cdot nq')$ time because for each $f \in \mathcal{F}$, its values $f(x)$ can be computed by sequentially scanning $(U_1, \ldots, U_{q'})$ in time $\mathcal{O}(nq')$.

We complete the proof by showing that $\mathcal{F}$ is an (n,k,q)-universal function family. First, observe that for all $S \subseteq A$ of size at most k, for each $i \in [q']$, the family $\{I_{U_i}|_S \colon U_i \in \mathcal{U}_i\}$ contains all functions from S to $\{0,1\}$. Consequently, $\{g(I_{U_1}|_S, \ldots, I_{U_{q'}}|_S) \colon (U_1, \ldots, U_{q'}) \in \mathcal{U}_\times\}$ contains all functions from S to $[2^{q'}]$. Since the function h is surjective, the family $\{f|_S : f \in \mathcal{F}\}$ includes all functions from S to B. □

3.3 Odd Cycle Cuts

In order to extend the "antler" framework of [5] to ODD CYCLE TRANSVERSAL (OCT), we define a problem-specific decomposition which we term *Odd Cycle Cuts* (OCCs). Our decompositions have three parts — a bipartite induced subgraph X_B, a vertex separator X_C (which we call the *head*), and a remainder X_R.

Definition 3.12 (Odd Cycle Cut)**.** Given a graph G, a partition (X_B, X_C, X_R) of $V(G)$ is an *Odd Cycle Cut (OCC)* if (1) $G[X_B]$ is bipartite, (2) there are no edges between X_B and X_R, and (3) $X_C \cup X_B \neq \emptyset$.

We say $|X_C|$ is the *width* of an OCC, and observe that X_C hits all odd cycles in $G - X_R$. We denote the minimum size of an OCT in G by $\mathsf{oct}(G)$.

Observation 3.13. *If (X_B, X_C, X_R) is an OCC in G, then $|X_C| \geq \mathsf{oct}(G[X_C \cup X_B])$.*

Analogous to z-antlers [5], here we define a *tight OCC* as a special case of an OCC. For a graph G, a set $X_C \subseteq V(G)$ and an integer z, an X_C-*certificate* of order z is a subgraph H of G such that X_C is an optimal OCT of H, and for each component H' of H we have $|X_C \cap V(H')| \leq z$. Throughout the chapter, and starting with the following definition, we will use the convention of referring to a tight OCC as (A_B, A_C, A_R) to emphasize its stronger guarantees compared to an arbitrary OCC (X_B, X_C, X_R).

Definition 3.14 ((z-)tight OCC)**.** An OCC (A_B, A_C, A_R) of a graph G is *tight* when $|A_C| = \mathsf{oct}(G[A_C \cup A_B])$. Furthermore, (A_B, A_C, A_R) is a tight OCC of order z (equivalently, z-*tight OCC*) if $G[A_C \cup A_B]$ contains an A_C-certificate of order z.

Note this definition naturally implies $\mathsf{oct}(G) = |A_C| + \mathsf{oct}(G[A_R])$: the union of A_C with a minimum OCT in $G[A_R]$ forms an OCT for G (since A_C separates A_B from A_R) for which the requirement $|A_C| = \mathsf{oct}(G[A_C \cup A_B])$ guarantees optimality.

The following lemma characterizes an intersection of an OCC and a tight OCC.

Lemma 3.15. *Let (X_B, X_C, X_R) be a (not necessarily tight) OCC in the graph G and let (A_B, A_C, A_R) be a tight OCC in G. Then $|A_C \cap X_B| \leq |X_C|$.*

Proof. Suppose for contradiction that $|A_C \cap X_B| > |S|$. Then, $A'_C := (A_C \setminus X_B) \cup (X_C \cap (A_B \cup A_C))$ is a subset of $A_B \cup A_C$ that is strictly smaller than A_C. Now, showing that A'_C is an OCT of $G[A_B \cup A_C]$ contradicts the assumption that A_C is a smallest such OCT by virtue of (A_B, A_C, A_R) being a tight OCC.

To show that A'_C is an OCT of $G[A_B \cup A_C]$, we let F be an arbitrary odd cycle in this graph and show that it intersects A'_C. First, if F intersects X_C, it intersects A'_C in particular, since $X_C \cap (A_B \cup A_C) \subseteq A'_C$.

Otherwise, since X_C separates X_B and X_R in G, F is completely contained in either $G[X_B]$ or $G[X_R]$. The former is not possible, since $G[X_B]$ is bipartite by assumption, so F lives in $G[X_R]$. Furthermore, since F was assumed to live in $G[A_B \cup A_C]$ and $G[A_B]$ is bipartite, F intersects A_C. In particular, as we found F to live in $G[X_R]$, it intersects $A_C \cap X_R$ which is a subset of A'_C by construction. Hence, F intersects A'_C in any case. □

The main result of this section is that assuming a graph G has a z-tight OCC, there exists a z-tight OCC (A_B, A_C, A_R) such that the number of components in $G[A_B]$ is bounded in

terms of z and $|A_C|$. To facilitate a proof, here we characterize the components in the bipartite part of an X_C-certificate that are safe to remove.

Lemma 3.16. *For a graph G and its vertex set $X_C \subseteq V(G)$, let H be an X_C-certificate of order z in G. Suppose there exists a connected component D of $H - X_C$ that satisfies the following two criteria, where H' is a connected component of H that contains D, and $X'_C := X_C \cap V(H')$:*

(1) For each vertex $x \in X'_C$ for which $H[\{x\} \cup V(D)]$ contains an odd cycle, there are at least z connected components $D' \neq D$ of $H - X_C$ for which $H[\{x\} \cup V(D')]$ also contains an odd cycle, and

(2) For each pair of distinct vertices $x, y \in X'_C$, for each parity $p \in \{0, 1\}$, if the graph $H[\{x, y\} \cup V(D)]$ contains an (x, y)-path of parity p, then there are at least z connected components $D' \neq D$ of $H - X_C$ for which $H[\{x, y\} \cup V(D')]$ also contains an (x, y)-path of parity p.

Then $\mathsf{oct}(H) = \mathsf{oct}(H - V(D))$.

Proof. Let us suppose that D satisfies these criteria. Assume for a contradiction that $\mathsf{oct}(H - V(D)) < \mathsf{oct}(H)$. Then $\mathsf{oct}(H' - V(D)) < \mathsf{oct}(H') \leq z$, where the last inequality comes from the assumption that H is an order-z X_C-certificate. Let X be an optimal OCT for $\mathsf{oct}(H' - V(D))$, whose size is less than $\mathsf{oct}(H') \leq z$. It follows that $H' - X$ has an odd cycle. Out of all odd cycles of $H' - X$, let A be an odd cycle that minimizes the quantity $t := |V(A) \cap V(D)|$. Note that $t > 0$, otherwise A would also be an odd cycle in $H' - V(D)$ which contradicts X being an OCT for that subgraph. Since X_C is an OCT in H by the definition of X_C-certificate, the component D of $H - X_C$ is bipartite. Hence the odd cycle A is not fully contained in $V(D)$, but contains at least one vertex d of D since $t \geq 1$. Orient the odd cycle A in an arbitrary direction and let x be the first vertex after d on A that does not belong to $V(D)$. Similarly, let y be the first vertex before d on A that does not belong to $V(D)$. We distinguish two cases, depending on whether $x = y$.

Case 1: If $x = y$. In this case, A is an odd cycle in $H[\{x\} \cup V(D)]$. Since component D satisfied criterion (1), there are at least z connected components $D' \neq D$ of $H - X_C$ for which $H[\{x\} \cup V(D')]$ contains an odd cycle. Since $|X| < z$, at least one of these components D^* does not contain any vertex of X. Then, $H[\{x\} \cup V(D^*)]$ contains an odd

cycle A', but we have $V(A') \cap V(D) = \emptyset$. This contradicts our choice of A as an odd cycle minimizing $V(A) \cap V(D)$.

Case 2: If $x \neq y$. Observe that the subpath of C from x to y through d forms an (x, y)-path P in the graph $H[\{x, y\} \cup V(D)]$. Let p be the parity of path P. Since component D satisfied criterion (2), there are at least z connected components $D' \neq D$ of $H - X_C$ that provide an (x, y)-path of parity p. Since $|X| < z$, at least one of these components D^* does not contain any vertex of X. Consider the closed walk A' that is obtained by replacing the subpath P of A by an (x, y)-path P' of the same parity in the graph $H[\{x, y\} \cup V(D^*)]$. Since the replacement preserves the total parity and provides an alternative connection between the two vertices x, y that appeared on A, it follows that A' is a closed odd walk. Since $V(D^*) \cap X = \emptyset$, it follows that A' is a closed odd walk in $H' - X$, which implies that $H' - X$ contains an odd cycle whose vertex set is a subset of $V(A')$. But note that A' uses strictly fewer vertices from $V(D)$ than A does, since vertex $d \in V(D)$ does not appear in $V(A')$. This contradicts our choice of A as an odd cycle minimizing $|V(A) \cap V(D)|$ and proves the claim. □

The following lemma states that we can bound the number of components in an X_C-certificate. This is inspired by and a variation of [5, Lemma 4.5].

Lemma 3.17. *Let G be a graph. For a set $X_C \subseteq V(G)$, let H be an X_C-certificate of order z in G. Then H contains an X_C-certificate $\hat{H}$ of order z in G such that $\hat{H} - X_C$ has at most $z^2|X_C|$ components.*

Proof. If there is a connected component D of $H - X_C$ that satisfies the two criteria of Lemma 3.16, then, update H to $H - V(D)$. From Lemma 3.16, we have $\mathsf{oct}(H) = \mathsf{oct}(H - V(D)) = |X_C|$, and since we never remove vertices in X_C, $H - V(D)$ remains an z-tight OCC. We repeat this process until the criteria become unsatisfied, and let $\hat{H}$ be the final graph.

Now, we count the number of components in $\hat{H} - X_C$. For each vertex $x \in X_C$, there are at most z components in $\hat{H} - X_C$ that violates criterion (1) because otherwise, one of those components will satisfy criterion (1). This results in that there exist at most $z \cdot |X_C|$ components in $\hat{H} - X_C$ violating criterion (1).

For each component H' of H, for each distinct vertex pair $x, y \in X_C \cap H'$, and for each parity $p \in \{0,1\}$, there are at most z components in $\hat{H} - X_C$ that violates criterion (2) because otherwise, one of those components will satisfy criterion (2). This results in that there exist at most $\sum_{H'} \binom{|X_C \cap V(H')|}{2} \cdot 2z = z \cdot \sum_{H'}(|H_C \cap V(H')|)(|H_C \cap V(H')| - 1) \leq z \cdot (z-1)|X_C|$ components in $\hat{H} - X_C$ violating criterion (2).

Hence, the number of components in $\hat{H} - X_C$ is upper-bounded by $z \cdot |X_C| + z \cdot (z-1)|X_C| = z^2|X_C|$, as desired. □

Lastly, we state the following lemma. This is an extension of [5, Lemma 4.6].

Lemma 3.18. *Let (A_B, A_C, A_R) be a z-tight OCC in a graph G for some $z \geq 0$. There exists a set $A'_B \subseteq A_B$ such that $(A'_B, A_C, A_R \cup A_B \setminus A'_B)$ is a z-tight OCC in G and $G[A'_B]$ has at most $z^2|A_C|$ components.*

Proof. Since (A_B, A_C, A_R) is a z-tight OCC, the graph $G[A_B \cup A_C]$ contains an A_C-certificate H of order z. From Lemma 3.17 we know that H contains an A_C-certificate $\hat{H}$ of order z such that $\hat{H} - A_C$ has at most $z^2|A_C|$ components.

Let A'_B be the union of the vertices in the components B in $G[A_B]$ such that B includes at least one vertex from $\hat{H}$. By definition, $(A'_B, A_C, A_R \cup A_B \setminus A'_B)$ is a z-tight OCC such that $G[A'_B]$ has at most $z^2|A_C|$ components. □

Finally, we introduce the notion of an *imposed separation problem* whose solutions naturally correspond to odd cycle transversals of specific subgraphs.

Definition 3.19. Let (X_B, X_C, X_R) be an OCC of G, and let $f_B \colon X_B \to \{0,1\}$ be a proper 2-coloring of $G[X_B]$. Let $C_1, C_2 \subseteq X_C$ be two disjoint subsets of X_C and let $f_C \colon C_1 \to \{0,1\}$ be a (not necessarily proper) 2-coloring of the vertices in C_1. Based on this 4-tuple of objects (C_1, C_2, f_C, f_B), we define three (potentially overlapping) subsets $A, R, N \subseteq X_B$.

1. Let A be the set of vertices $v_b \in X_B$ with a neighbor $v_c \in C_1$ such that $f_B(v_b) = f_C(v_c)$.
2. Let R be the set of vertices $v_b \in X_B$ with a neighbor $v_c \in C_1$ such that $f_B(v_b) \neq f_C(v_c)$.
3. Finally, let $N := N_G(C_2) \cap X_B$.

We refer to the problem of finding a smallest $\{A, R, N\}$-separator in $G[X_B]$ as *the* $\{A, R, N\}$-*separation problem imposed onto* $G[X_B]$ *by* (C_1, C_2, f_C, f_B).

To see the connection between solutions and OCTs, one may let C_1 and f_B in this definition correspond to W and c respectively in Lemma 3.2, while the color classes of f_C correspond to the sets W_0 and W_1 respectively. As shown below in Lemma 3.20, we can recognize parts of tight OCCs as optimal solutions to specific imposed separation problems.

Although Definition 3.19 requires f_B and f_C to be colorings of X_B and C_1 respectively, we sometimes abuse the notation by providing colorings whose domains are supersets of these intended domains. In these cases, one may interpret the definition of the imposed separation problem as if given the restrictions of these colorings to their respective intended domains.

One important role of these separation problems is to allow us to characterize intersections of two OCCs when at least one is tight. Specifically, in Lemma 3.20, we show that the intersection of one OCC's head with the other OCC's bipartite part forms an optimal solution to a specific 3-way separation problem, which is even optimal for a corresponding 2-way problem.

Lemma 3.20. *[♦] Let (X_B, X_C, X_R) be a (not necessarily tight) OCC in the graph G and let (A_B, A_C, A_R) be a tight OCC in G. Let $f_X \colon X_B \to \{0,1\}$ and $f_A \colon A_B \to \{0,1\}$ be proper 2-colorings of $G[X_B]$ and $G[A_B]$ respectively. Let A, R and N be the three sets to be separated in the separation problem imposed onto $G[X_B]$ by $(X_C \cap A_B, X_C \cap A_R, f_A, f_B)$ and let their names correspond to their roles as defined in Definition 3.19. Then, $A_C \cap X_B$ is both a minimum-size $\{A, R\}$-separator and a minimum-size $\{A, R, N\}$-separator in $G[X_B]$.*

This will prove to be a useful property in Section 3.5 by which we are able to recognize part of a tight OCC (A_B, A_C, A_R) in an arbitrary graph. In Lemma 3.15, we show that for any other OCC (X_B, X_C, X_R) the intersection $A_C \cap X_B$ is at most as large as X_C.

3.4 Finding Odd Cycle Cuts

Our ultimate goal is to show that if the graph contains any tight OCC (X_B, X_C, X_R) with $|X_C| \leq k$, then we can produce a tight OCC with $|X_C| \leq k$ *and* $|X_B|$ upper-bounded

by some function of k. To achieve this, we first show that we can efficiently find some OCC where $|X_B|$ is large enough, and then (in Section 3.5) that we can reduce any such cut so that $|X_B|$ is small without destroying any essential structure of the input graph.

Specifically, we say an OCC (X_B, X_C, X_R) is *reducible* with respect to some function g_r if $|X_B| > g_r(|X_C|)$. Our results all hold for a specific polynomial $g_r(x)$ in $\Theta(x^{16})$, which we specify in the main paper. We say an OCC (X_B, X_C, X_R) is a *single-component* OCC if $G[X_B]$ is connected.

Given a graph G, our goal is to output a reducible OCC efficiently assuming that G contains a single-component OCC (X_B, X_C, X_R) with $|X_B| > g_r(2|X_C|)$ and $|X_C| \leq k$. We achieve this by color coding of the vertices in G (see definitions in Section 3.2.3 for details). Consider a coloring $\chi\colon V(G) \to \{\dot{\mathsf{B}}, \dot{\mathsf{C}}\}$. For an integer ℓ, an OCC (X_B, X_C, X_R) with $|X_B| \geq \ell$ is *ℓ-properly colored* by χ if $X_C \subseteq \chi^{-1}(\dot{\mathsf{C}})$ and there is a set of ℓ vertices of X_B that are colored $\dot{\mathsf{B}}$ and induce a connected subgraph of G.

Our result is based on bipartite separations, as introduced by Jansen *et al.* [13]. Specifically, define a *$(\mathcal{H}, k)$-separation* to be a pair (C, S) of disjoint vertex sets for a graph class $\mathcal{H}$ and an integer k such that $G[C] \in \mathcal{H}$, $|S| \leq k$ and $N_G(C) \subseteq S$. For $\mathcal{H} = \mathsf{bip}$, where bip denotes the class of bipartite graphs, Jansen *et al.* [13] gave a 2-approximation for the problem of computing a (bip, k)-separation covering a given connected vertex set Z.

Corollary 3.21 ([13, Lemma 4.24] (rephrased))**.** *There is a polynomial-time algorithm that, given a graph G, an integer k, and a non-empty vertex set Z such that $G[Z]$ is connected, either:*

- *returns a $(\mathsf{bip}, 2k)$-separation (C, S) with $Z \subseteq C$, or*
- *concludes that G does not have a (bip, k)-separation (C', S') with $Z \subseteq C'$.*

The original formulation of the corresponding result is in terms of a problem called $(\mathcal{H}, k)$-Separation Finding. In the definition of this problem, the input graph is required to have a bound on its $\mathcal{H}$-treewidth and the output separation is only guaranteed to *weakly* cover Z, meaning that $Z \subseteq C \cup S$ rather than $Z \subseteq C$. Jansen *et al.* [13] used this formalism to treat a variety of different graph classes $\mathcal{H}$. For the particular case of bipartite graphs that is of interest here, the assumption on the $\mathcal{H}$-treewidth of the input graph is never used (as is acknowledged in their paper), and from the specification of the algorithm it can be

seen that $Z \cap S$ is empty: the set S is constructed as a separator in an auxiliary graph, separating two copies of Z without intersecting these sets. Hence our formulation follows directly from their results.

First, we show how to construct an OCC with large X_B from a proper coloring.

Lemma 3.22. *Given a graph G, integers k, ℓ, and a coloring $\chi\colon V(G) \to \{\dot{\mathsf{B}}, \dot{\mathsf{C}}\}$ of $V(G)$ that ℓ-properly colors a single-component OCC (X_B, X_C, X_R) with $|X_C| \leq k$, an OCC (X'_B, X'_C, X'_R) such that $|X'_B| \geq \ell$ and $|X'_C| \leq 2k$ can be found in polynomial time.*

Proof. Consider the following algorithm. Let Z be the vertices in a connected component of $G[\chi^{-1}(\dot{\mathsf{B}})]$. We iterate over all components. If $|Z| < \ell$ or $G[Z]$ is not bipartite, continue to the next component. Otherwise, we invoke Corollary 3.21 to find a $(\mathsf{bip}, 2k)$-separation (B, S) such that $Z \subseteq B$. If such a separation exists, output $(B, S, V(G) \setminus (B \cup S))$. Otherwise, continue to the next component.

(Correctness) Let $G' = G[\chi^{-1}(\dot{\mathsf{B}})]$. By assumption, the single-component OCC (X_B, X_C, X_R) is ℓ-properly colored by χ. Hence $V(G') \cap X_C = \emptyset$ and G' contains a connected component $G[Z]$ such that $Z \subseteq X_B$ and $|Z| \geq \ell$. Also, because (X_B, X_C) is a $(\mathsf{bip}, |X_C|)$-separation in G such that $Z \subseteq X_B$, the algorithm must output some solution $(B, S, V(G) \setminus (B \cup S))$ such that $Z \subseteq B$, which is by definition an OCC, and we know $|B| \geq |Z| \geq \ell$ and $|S| \leq 2|X_C| \leq 2k$.

(Running time) From Corollary 3.21, we can, given a component Z, find a $(\mathsf{bip}, 2k)$-separation (B, S) such that $Z \subseteq B$ in polynomial time. There are $\mathcal{O}(n)$ components in G', so the overall runtime is also polynomial. □

Now, we use this coloring scheme to find a reducible OCC, assuming that a graph G has a single-component OCC (X_B, X_C, X_R) with large X_B.

Lemma 3.23. *There exists a $2^{\mathcal{O}(k^{16})} n^{\mathcal{O}(1)}$-time algorithm that, given a graph G and an integer k, either determines that G does not contain a single-component OCC (X_B, X_C, X_R) of width at most k with $|X_B| > g_r(2k)$ or outputs a reducible OCC in G.*

Proof. We will invoke the algorithm from Lemma 3.22 multiple times for $\ell = g_r(2k) + 1$. If we supply a coloring that ℓ-properly colors (X_B, X_C, X_R), then the algorithm is guaranteed to find an OCC (X'_B, X'_C, X'_R) such that $|X'_B| > g_r(2k)$ and $|X'_C| \leq 2k$, which is reducible

as $|X'_B| > g_r(2k) \geq g_r(|X'_C|)$. If all relevant colorings fail to find such a reducible OCC, then we can conclude that G does not contain a single-component OCC (X_B, X_C, X_R) with $|X_C| \leq k$ and $|X_B| \geq \ell > g_r(2k)$.

Let $X'_B \subseteq X_B$ be an arbitrary vertex set of size ℓ that induces a connected subgraph of G. Since $G[X_B]$ is connected, such X'_B must exist. Observe that we obtain an ℓ-proper coloring if $X_C \cup X'_B$ are colored correctly. Let $s = |X_C \cup X'_B| = k + g_r(2k) + 1 = \mathcal{O}(k^{16})$.

Using an (n, k)-universal set, which is a well-known pseudorandom object [19, 3] used to derandomize applications of color coding (see Theorem 3.10), we can construct a family of $2^{\mathcal{O}(s)} \log n$ many subsets $A_1, \ldots, A_{2^{\mathcal{O}(s)} \log n}$ with the guarantee that for each set $S \subseteq V(G)$ of size s, for each subset S' of S, there exists a set in the family with $A_i \cap S = S'$. This can be done in $2^{\mathcal{O}(s)} n \log n = 2^{\mathcal{O}(k^{16})} n \log n$ time. From this family, we can construct a family of colorings that is guaranteed to include one that ℓ-properly colors a suitable OCC (X_B, X_C, X_R) if one exists. To derive a coloring χ_i from a member $A_i \subseteq V(G)$ of the (n, s)-universal set, it suffices to pick $\chi(a \in A) = \dot{\mathsf{R}}$ and $\chi(a \notin A) = \dot{\mathsf{B}}$.

We run the $n^{\mathcal{O}(1)}$-time algorithm from Lemma 3.22 for each coloring, which results in the overall runtime $2^{\mathcal{O}(k^{16})} n^{\mathcal{O}(1)}$. □

3.5 Reducing Odd Cycle Cuts

Given an OCC (X_B, X_C, X_R) of G with $|X_B| > g_r(|X_C|)$, the next step is to "shrink" X_B in a way that preserves some of the structure of the input graph. In this section, we give a reduction to do this and prove that it preserves the general structure of minimum-size OCTs and of tight OCCs in the graph. The reduction starts with a marking scheme that is discussed separately in Section 3.5.1. We give the full reduction, which includes this marking scheme as a subroutine, in Section 3.5.2. The reduction will only affect $G[X_B]$ and the edge set between X_B and X_C, which already ensures that an important part of the input graph is maintained.

3.5.1 A Marking Scheme for the Reduction

The goal of the marking scheme is to mark a set $B^* \subseteq X_B$ of size $|X_C|^{\mathcal{O}(1)}$ as "interesting" vertices that the reduction should not remove or modify. Intuitively, we want this set to contain vertices which we expect might be part of the cut part of a tight OCC in G. More

precisely, we guarantee that for every tight OCC in G there is a (possibly different) tight OCC (A_B, A_C, A_R) such that $A_C \cap X_B$ is contained in the marked set B^*.

As seen in Lemma 3.20, for every tight OCC (A_B, A_C, A_R) in the graph, the intersection $A_C \cap X_B$ forms an optimal solution to a specific imposed separation problem (Definition 3.19). As such, it suffices if B^* is a *cut covering set* for these imposed separation problems.

Indeed, the key ingredient of the algorithm presented below is the computation of such a cut covering set. Preceding this computation is a graph reduction ensuring that the computed set covers precisely the imposed separation problems. In Lemma 3.9, we will show that a cut covering set can be computed in deterministic FPT time parameterized by the size of the terminal set, which leads to a total running time of $2^{\mathcal{O}(|X_C|)}n^{\mathcal{O}(1)}$ time for the marking step.

3.5.1.1 Marking step. Consider the following algorithm.

Input: A graph G and an OCC (X_B, X_C, X_R) of G.

Output: Marked vertices $B^* \subseteq X_B$.

1. Find a proper 2-coloring $f_X \colon X_B \to \{0, 1\}$ of $G[X_B]$.

2. Construct an auxiliary (undirected) graph G', initialized to a copy of $G[X_B]$. For each $v \in X_C$, do the following:

   - Add vertices $v^{(0)}$ and $v^{(1)}$ to G'.
   - For each neighbor $u \in N_G(v) \cap X_B$, add an edge $v^{(f_X(u))}u$.

   Let T be the set $\{v^{(i)} \mid v \in X_C, i \in \{0, 1\}\}$. Note that $|T| = 2|X_C|$.

3. Compute a cut covering set $B^* \subseteq V(G')$ via Lemma 3.9 such that for every partition $\mathcal{T}^* = (T_0, T_1, T_2, T_3, T_X)$ of T, the set B^* contains a minimum-size solution to the following problem:

   - find a vertex set $S \subseteq V(G') \setminus (T_1 \cup T_2 \cup T_3)$ such that S separates T_i and T_j in the graph $G' - T_X$ for all $1 \leq i < j \leq 3$,

   as long as this problem has a solution of size at most $|T|$.

Lemma 3.24. *[♦] Let B^* be constructed as in the Marking step when given the graph G and an OCC (X_B, X_C, X_R) of G as input. If there exists a z-tight OCC (A_B, A_C, A_R) in G, then there exists a z-tight OCC (A_B^*, A_C^*, A_R^*) in G with $|A_C^*| = |A_C|$ and with $A_C^* \cap X_B \subseteq B^*$.*

Proof sketch. Let $f_X \colon X_B \to \{0,1\}$ be the 2-coloring obtained in step 1 of the Marking step, let $f_A \colon A_B \to \{0,1\}$ be a proper 2-coloring of $G[A_B]$ and consider the separation problem imposed onto $G[X_B]$ by $(X_C \cap A_B, X_C \cap A_R, f_A, f_B)$. Let A, R and N be the three sets to be separated in this problem with their names corresponding to their roles as in Definition 3.19.

By putting the correct copy of each vertex from $A_B \cap X_C$ into T_1 and T_2 respectively, putting both copies of vertices from $A_R \cap X_C$ into T_3 and putting both copies of vertices from $A_C \cap X_C$ into T_X, we obtain a partition $(\emptyset, T_1, T_2, T_3, T_X)$ of the set T defined in step 2, such that the corresponding separation problem has the same solution space as the $\{A, R, N\}$-separation problem imposed onto $G[X_B]$. By construction of B^* in step 3, there is a set $S \subseteq B^*$ (possibly different from $A_C \cap X_B$) that is an optimal $\{A, R, N\}$-separator in $G[X_B]$. To construct the tight OCC (A_B^*, A_C^*, A_R^*), we use this set S as replacement for $A_C \cap X_B$, which is also a minimum-size $\{A, R, N\}$-separator in $G[X_B]$ by Lemma 3.20.

As such, we define $A_C^* := (A_C \setminus X_B) \cup S$. To define A_B^*, let U be the set of vertices from $X_B \setminus S$ that are not reachable from N in $G[X_B] - S$. Now, we define $A_B^* := (A_B \setminus X_B) \cup U$. Finally, we define $A_R^* := V(G) \setminus (A_B^* \cup A_C^*)$. Clearly, this 3-partition of $V(G)$ satisfies the constraints $|A_C^*| = |A_C|$ and $A_C^* \cap X_B \subseteq B^*$. We proceed by showing that it satisfies the three additional properties required to be a z-tight OCC.

First, to see that $G[A_B^*]$ is bipartite, we note that A_B^* only contains vertices from A_B and $X_B \setminus S$. Both are vertex sets that induce a bipartite subgraph. Then, noting the correspondence between the sets A and R obtained from the separation problem and the sets A and R as in Lemma 3.2, we invoke this lemma on $G[(X_C \cap A_B) \cup X_B]$ with $c = f_A$ and with W_0 and W_1 being the two color classes of this coloring restricted to $X_C \cap A_B$. It follows that the vertices from A_B^* in $X_B \setminus S$ can be properly 2-colored by a coloring f that agrees with f_A on the vertex set $X_C \cap A_B$ that separates $A_B^* \cap X_R$ and $A_B^* \cap X_B$. As these two vertex sets are properly colored by f_A and f respectively, these colorings combine to a proper 2-coloring of the entire graph $G[A_B^*]$ (see Lemma 3.3).

Secondly, a case distinction shows that there are no edges between A_B^* and A_R^*. It combines the fact that $A_C \cap X_B$ is an $\{A, R, N\}$-separator in $G[X_B]$ — thereby in particular separating $A_B \cap X_B$ from N in $G[X_B]$ — and the fact that A_B^* only contains vertices that already belonged to A_B and vertices from X_B that are not reachable from N in $G[X_B] - A_C^*$.

Finally, it remains to show that (A_B^*, A_C^*, A_R^*) has an A_C^*-certificate of order z. To prove this, we show that the order-z certificate D of the original OCC (A_B, A_C, A_R) is also an order-z certificate in (A_B^*, A_C^*, A_R^*). The main effort here is to prove that D even lives in $A_B^* \cup A_C^*$, after which it is easy to see that it is also an order-z certificate for our new OCC.

As Lemma 3.20 guarantees that $A_C \cap X_B$ is not only an optimal $\{A, R, N\}$-separator in $G[X_B]$ but even an optimal $\{A, R\}$-separator in this graph, it contains exactly one vertex from every path of a maximum packing $\mathcal{P}$ of pairwise vertex-disjoint (A, R)-paths in $G[X_B]$, due to Menger's theorem [22, Theorem 9.1]. Likewise, $A_C^* \cap X_B = S$ is also an optimal $\{A, R\}$-separator in $G[X_B]$ and hence also contains exactly one vertex from every path of $\mathcal{P}$.

Intuitively, for any path $P \in \mathcal{P}$, a vertex on this path that stops being reachable from one endpoint of P when sliding the picked vertex along the path, starts becoming reachable from the other endpoint of P. As both endpoints of P belong to $A \cup R$ and S only differs from $A_C \cap X_B$ by which vertex is picked from each path in $\mathcal{P}$, it cannot drastically alter which vertices are reachable from $A \cup R$, which in turn are all vertices that end up in A_B^*.

Using the observation that A_B and A_B^* are separated from N by A_C and A_C^* respectively, we see that all vertices that *are* disconnected from $A \cup R$ by substituting $A_C \cap X_B$ for S are in particular also disconnected from N. Thereby, these vertices end up in A_B^*. This shows that $(A_B \cup A_C) \subseteq (A_B^* \cup A_C^*)$, which implies that the certificate D also lives in the latter. □

Now, we cover the various guarantees that were promised for the Marking step. A visual example of the graph reduction that is part of it may be found in Figure 3.2.

First, we briefly argue the following basic guarantees of the Marking step.

Lemma 3.25. *Let G be a graph and let (X_B, X_C, X_R) be an OCC in it. Then the Marking step can be executed with these input parameters in $2^{\mathcal{O}(|X_C|)} n^{\mathcal{O}(1)}$ time and outputs a set B^* of size $|X_C|^{\mathcal{O}(1)}$.*

Proof. First we note that indeed $|B^*| = |X_C|^{\mathcal{O}(1)}$, since B^* is constructed as a set of at most

$c \cdot (2|X_C|)^8$ vertices. Next, it is clear to see that steps 1 and 2 can be done in polynomial time, and step 3 takes $2^{\mathcal{O}(|T|)}n^{\mathcal{O}(1)} = 2^{\mathcal{O}(|X_C|)}n^{\mathcal{O}(1)}$ time. □

Next, we formalize the main property we wished to hold for the Marking step and proof that it is indeed satisfied.

Lemma 3.26. *Let G be a graph, let (X_B, X_C, X_R) be an OCC in it and let $B^* \subseteq X_B$ be the result of executing the Marking step with these input parameters. Let $f_X : X_B \to \{0,1\}$ be the proper 2-coloring of $G[X_B]$ computed in the first step of the Marking step. Let $C_1, C_2 \subseteq X_C$ be two arbitrary disjoint subsets of X_C and let $f_C : C_1 \to \{0,1\}$ be a (not necessarily proper) 2-coloring of C_1. If the $\{A, R, N\}$-separation problem imposed onto $G[X_B]$ by (C_1, C_2, f_C, f_X) has a solution of size at most $|X_C|$, then the set B^* contains an optimal solution to this separation problem.*

Proof. Recall that the auxiliary graph G' consists of a copy of $G[X_B]$ and terminals $T = \{v^{(i)} \mid v \in X_C, i = 0,1\}$. Given C_1, C_2 and f_C, consider the following generalized 3-partition $\mathcal{T}$ of T into $(\emptyset, A', R', N', T_X)$, where T_X represents deleted vertices.

- If $v \in C_1$ and $f_C(v) = 0$, then let $v^{(0)} \in A'$ and $v^{(1)} \in R'$.
- If $v \in C_1$ and $f_C(v) = 1$, then let $v^{(0)} \in R'$ and $v^{(1)} \in A'$.
- If $v \in C_2$, then let $v^{(0)}, v^{(1)} \in N'$.
- If $v \in X_C \setminus (C_1 \cup C_2)$, then let $v^{(0)}, v^{(1)} \in T_X$.

Figure 3.2 illustrates this construction. Note that for $v \in C_1$ we have $v^{(i)} \in A'$ ($v^{(i)} \in R'$, resp.) if and only if $f_C(v) = i$ ($f_C(v) = 1 - i$). We shall show the equivalence between restricted 3-way cuts for $\{A', R', N'\}$ in $G' - T_X$ and $\{A, R, N\}$-separators in $G[X_B]$.

First, let S' be a restricted 3-way cut for $\{A', R', N'\}$ in $G' - T_X$, and assume that there is a path sPt in $G[X_B] - S'$ where s and t belong to different sets in $\{A, R, N\}$.

(1) If $s \in A$ and $t \in R$, then there exist vertices $v_s \in N_G(s) \cap C_1$ and $v_t \in N_G(t) \cap C_1$ such that $f_C(v_s) = f_X(s)$ and $f_C(v_t) \neq f_X(t)$. This implies $v_s^{(f_X(s))}s, v_t^{(f_X(t))}t \in E(G')$. By construction, $v_s^{(f_X(s))} = v_s^{(f_C(v_s))} \in A'$ and $v_t^{(f_X(t))} = v_t^{(1-f_C(v_t))} \in R'$, and these vertices are present in $G' - T_X$. Hence, $v_s^{(f_X(s))}sPtv_t^{(f_X(t))}$ is a path between A' and R' in $G' - T_X - S'$, a contradiction.

(2) If $s \in A$ and $t \in N$, then there exist vertices $v_s \in N_G(s) \cap C_1$ and $v_t \in N_G(t) \cap C_2$ such that $f_C(v_s) = f_X(s)$. This implies $v_s^{(f_X(s))}s, v_t^{(f_X(t))}t \in E(G')$. By construction, $v_s^{(f_X(s))} = v_s^{(f_C(v_s))} \in A'$ and $v_t^{(f_X(t))} \in N'$. Hence, $v_s^{(f_X(s))}sPtv_t^{(f_X(t))}$ is a path between A' and N' in $G' - T_X - S'$, again a contradiction.

Other cases are proven symmetrically.

Conversely, let S be an $\{A, R, N\}$-separator in $G[X_B]$, and assume that S is not a restricted 3-way cut for $\{A', R', N'\}$ in $G' - T_X$. Then, since terminals in G' are independent, there is a path $s^{(i)}Pt^{(j)}$ in $G' - T_X - S$ such that $s^{(i)}$ and $t^{(j)}$ belong to different sets in $\{A', R', N'\}$ and P is a non-empty path in $G'[X_B]$. Let s' and t' be the first and the last vertices in P respectively (s' and t' can be the same vertex). We will show that s' and t' belong to different sets in $\{A, R, N\}$ in $G[X_B]$, which leads to a contradiction.

(1) If $s^{(i)} \in A'$ and $t^{(j)} \in R'$, then $s, t \in C_1$ and $f_C(s) = i$, $f_C(t) = 1 - j$. Also, knowing that $f_X(s') = i$, $f_X(t') = j$, we have $s' \in A$ and $t' \in R$ in $G[X_B]$.

(2) If $s^{(i)} \in A'$ and $t^{(j)} \in N'$, then $s \in C_1$, $t \in C_2$, and $f_C(s) = i$. Since $f_X(s') = i$, we have $s' \in A$. From $t't^{(j)} \in E(G')$, we have $t' \in N$.

Other cases are symmetrical to these.

Now, suppose there exists a smallest $\{A, R, N\}$-separator of size at most $|X_C|$ in $G[X_B]$. Then, there exists a restricted 3-way cut of size at most $|X_C|$ for $\{A', R', N'\}$ in $G' - T_X$. From Lemma 3.9, we know that B^* contains a minimum restricted 3-way cut Z of size at most $|X_C| \leq |T|$ for $\{A', R', N'\}$ in $G' - T_X$. Finally, Z is indeed an optimal solution to the $\{A, R, N\}$-separation problem. □

3.5.2 Simplifying the Graph

Our eventual reduction starts with the Marking step from the previous section, after which the graph is modified in a way that leaves marked vertices untouched. We want the reduction to preserve the general structure of optimal OCTs and tight OCCs in the input graph. As this is governed by the locations and interactions of odd cycles in the graph, we encode this information in a more space-efficient manner using the following reduction.

3.5.2.1 Reduction step. Given a graph G and an OCC (X_B, X_C, X_R) of it, we construct a graph G' as follows.

1. Use the Marking step with input G and (X_B, X_C, X_R) to obtain the set $B^* \subseteq X_B$.

2. Initialize G' as a copy of $G - (X_B \setminus B^*)$.

3. For every $u, v \in X_C \cup B^*$ and for every parity $p \in \{\text{even}, \text{odd}\}$, check if the subgraph $G[X_B \setminus B^*]$ contains the internal vertex of a (u, v)-path with parity p. If so, then:

   - if $p =$ even, add two new vertices x and x' to G and connect both of them to u and v.
   - if $p =$ odd, add four new vertices x, y, x' and y' to G and add the edges $\{u, x\}$, $\{x, y\}$, $\{y, v\}$, $\{u, x'\}$, $\{x', y'\}$ and $\{y', v\}$.

   Note that we explicitly allow $u = v$ in this step.

Effectively, this reduction deletes the vertices $X_B \setminus B^*$ from the graph. For each pair of neighbors u, v from that set, if the deleted vertices provided an odd (resp. even) path between them, then we insert two vertex-disjoint odd (resp. even) paths between u and v. Hence we shrink the graph while preserving the parity of paths provided by the removed vertices.

As we will prove in Lemma 3.29 and Lemma 3.30 respectively, the reduction can be performed in $2^{\mathcal{O}(|X_C|)} \cdot n^{\mathcal{O}(1)}$ time and it is guaranteed to output a strictly smaller graph than its input graph whenever it receives an OCC that is reducible with respect to the function g_r as in Section 3.4. To show that the reduction also preserves OCT and OCC structures, we prove that it satisfies two safety properties formalized below in Lemmas 3.27 and 3.28.

Lemma 3.27. *[⧫] Let G be a graph, let (X_B, X_C, X_R) be an OCC in G and let G' be the graph obtained by running the Reduction step with these input parameters. For all $z \geq 0$, if there exists a z-tight OCC (A_B, A_C, A_R) in G, then there exists a z-tight OCC (A'_B, A'_C, A'_R) in G' with $|A'_C| = |A_C|$.*

The proof of the lemma above uses Lemma 3.24 to infer that, for any z-tight OCC (A_B, A_C, A_R) of G, the graph G also contains a z-tight OCC (A^*_B, A^*_C, A^*_R) of the same width such that $A^*_C \subseteq V(G) \cap V(G')$. This allows for the construction of an OCC (A'_B, A'_C, A'_R) in G' with $A'_C = A^*_C$. Then, A'_B can be defined as the union of $A^*_B \cap V(G) \cap V(G')$ and the set

of vertices that were added during the reduction to provide a replacement connection between any two vertices from $A_C^* \cup (A_B^* \cap V(G) \cap V(G'))$. Finally, $A'_R := V(G') \setminus (A'_B \cup A'_C)$.

The proof proceeds to show that the resulting partition (A'_B, A'_C, A'_R) is a z-tight OCC of G'. The two main insights used to prove this are the facts that:

- optimal OCTs of G' are disjoint from the set of newly added vertices $V(G') \setminus V(G)$, and
- odd cycles in G can be translated to very similar odd cycles in G' and vice versa.

These insights are also covered in the proof of the second safety property below.

Lemma 3.28. *[⧫] Let G be a graph, let (X_B, X_C, X_R) be an OCC in G and let G' be the graph obtained by running the Reduction step with these input parameters. If S' is a minimum-size OCT of G', then $S' \subseteq V(G) \cap V(G')$ and S' is a minimum-size OCT of G.*

We prove that the Reduction step can be executed in FPT time parameterized by $|X_C|$.

Lemma 3.29 (⧫). *The Reduction step can be performed in $2^{\mathcal{O}(|X_C|)} \cdot n^{\mathcal{O}(1)}$ time.*

Next, we show that the Reduction step outputs a strictly smaller graph than its input graph when the OCC it is given is reducible. Recall that we call an OCC (X_B, X_C, X_R) reducible if $|X_B| > g_r(|X_C|)$ for some polynomial $g_r(x)$. The exact definition of this polynomial had been postponed so far. Having reached the statement whose proof relies on this definition, we now specify this exact polynomial.

Its definition is based on Lemma 3.9 in which sets Z and T are specified. Setting the value of s in this lemma to 3 yields the existence of a constant $c \in \mathbb{N}$ such that $|Z| \leq c \cdot |T|^8$ for large enough $|T|$. Given this constant c, we define $g_r \colon \mathbb{N} \to \mathbb{N}$ as $g_r(x) = (6(2^8c + 1)^2 + 2^8c) \cdot x^{16}$. With this definition, we can prove the following result.

Lemma 3.30 (⧫). *Let G be a graph and let (X_B, X_C, X_R) be an OCC of it. If (X_B, X_C, X_R) is reducible, then running the Reduction step with these input parameters yields an output graph G' with $|V(G')| < |V(G)|$.*

3.6 Finding and Removing Tight OCCs

Now we find tight OCCs by the same color coding technique used in prior work [5]. Consider a coloring $\chi\colon V(G) \cup E(G) \to \{\dot{\mathsf{B}}, \dot{\mathsf{C}}, \dot{\mathsf{R}}\}$ of the vertices and edges of a graph G. For every color $c \in \{\dot{\mathsf{B}}, \dot{\mathsf{C}}, \dot{\mathsf{R}}\}$, let $\chi_V^{-1}(c) = \chi^{-1}(c) \cap V(G)$. For any integer $z \geq 0$, a z-tight OCC (A_B, A_C, A_R) is *z-properly colored* by a coloring χ if all the following hold: (i) $A_C \subseteq \chi_V^{-1}(\dot{\mathsf{C}})$, (ii) $A_B \subseteq \chi_V^{-1}(\dot{\mathsf{B}})$, and (iii) for each component H of $G' = G[A_B \cup A_C] - \chi^{-1}(\dot{\mathsf{R}})$ we have $\mathsf{oct}(H) = |A_C \cap V(H)|$ and $|A_C \cap V(H)| \leq z$. Note that $\chi^{-1}(\dot{\mathsf{R}})$ may include both vertices and edges, so that the process of obtaining G' involves removing both the vertices and edges colored $\dot{\mathsf{R}}$.

The algorithm to extract a z-tight OCC from a given coloring χ of the vertices and edges of G, is inspired by a corresponding step in previous work [5, Lemma 6.2]. The main idea is to iteratively refine the coloring, by changing the colors of vertices and edges into $\dot{\mathsf{R}}$ when the current coloring does not justify their membership in a z-tight OCC. It turns out that after exhaustively refining the coloring in this way, we can ensure the vertices colored $\dot{\mathsf{B}}$ and $\dot{\mathsf{C}}$ actually form the bipartite part and head of a z-tight OCC (assuming their union is nonempty). The most computationally expensive step of the algorithm comes from verifying the requirement that the coloring highlights an A_C-certificate, consisting of connected components that each have an OCT of size at most z. To verify this property, the algorithm will iterate over all sets C of at most z vertices colored $\dot{\mathsf{C}}$ and test whether they form an optimal OCT for the subgraph induced by C together with the vertex sets of connected components colored $\dot{\mathsf{B}}$ whose neighborhood is a subset of $\dot{\mathsf{C}}$. This leads to the $n^{\mathcal{O}(z)}$ term in the running time.

Lemma 3.31. *There is an $n^{\mathcal{O}(z)}$ time algorithm taking as input an integer $z \geq 0$, a graph G, and a coloring $\chi\colon V(G) \cup E(G) \to \{\dot{\mathsf{B}}, \dot{\mathsf{C}}, \dot{\mathsf{R}}\}$ that either determines that χ does not z-properly color any z-tight OCC, or outputs a z-tight OCC (A_B, A_C, A_R) in G such that for each OCC $(\hat{A}_B, \hat{A}_C, \hat{A}_R)$ that is z-properly colored by χ, we have $\hat{A}_B \subseteq A_B$ and $\hat{A}_C \subseteq A_C$.*

Proof. Let $G_\chi := G - \chi^{-1}(\dot{\mathsf{R}})$. Note that the value of this expression will change when the algorithm updates the coloring χ below. We define a function $W_\chi\colon 2^{\chi_V^{-1}(\dot{\mathsf{C}})} \to 2^{\chi_V^{-1}(\dot{\mathsf{B}})}$ as follows: for any $C \subseteq \chi_V^{-1}(\dot{\mathsf{C}})$ let $W_\chi(C)$ denote the set of all vertices that are in a component B of $G_\chi[\chi_V^{-1}(\dot{\mathsf{B}})]$ for which $N_{G_\chi}(B) \subseteq C$. Intuitively, the function W_χ maps $\dot{\mathsf{C}}$-colored

vertices C to $\dot{\mathsf{B}}$-colored vertices B such that B could be used to build a C-certificate. The following algorithm updates the coloring χ and recolors any vertex or edge that is not part of a z-tight OCC to color $\dot{\mathsf{R}}$.

1. Recolor all edges uv to color $\dot{\mathsf{R}}$ if $\chi(u) = \dot{\mathsf{R}}$ or $\chi(v) = \dot{\mathsf{R}}$.
2. For each component B of $G[\chi_V^{-1}(\dot{\mathsf{B}})]$, recolor all vertices of B and their incident edges to $\dot{\mathsf{R}}$ if $G[B]$ is not bipartite or $N_G(B) \not\subseteq \chi_V^{-1}(\dot{\mathsf{C}})$.
3. For each subset $C \subseteq \chi_V^{-1}(\dot{\mathsf{C}})$ of size at most z, mark all vertices in C if $\mathsf{oct}(G_\chi[C \cup W_\chi(C)]) = |C|$.
4. If $\chi_V^{-1}(\dot{\mathsf{C}})$ contains unmarked vertices we recolor them to $\dot{\mathsf{R}}$, clear markings made in Step 3 and repeat from Step 1.
5. At this point, all vertices in $\chi_V^{-1}(\dot{\mathsf{C}})$ are marked in Step 3. If $\chi_V^{-1}(\dot{\mathsf{B}}) \cup \chi_V^{-1}(\dot{\mathsf{C}}) \neq \emptyset$, then return $(\chi_V^{-1}(\dot{\mathsf{B}}), \chi_V^{-1}(\dot{\mathsf{C}}), \chi_V^{-1}(\dot{\mathsf{R}}))$ as a z-tight OCC.
6. Otherwise, report that χ does not z-properly color any z-tight OCC.

(Running time) There will be at most n iterations (Steps 1-4) since in every iteration the number of vertices in $\chi_V^{-1}(\dot{\mathsf{R}})$ increases. Steps 1, 2, 4, 5, and 6 can be performed in no more than $\mathcal{O}(n^2)$ time. For Step 3 we solve ODD CYCLE TRANSVERSAL in time $3^z n^{\mathcal{O}(1)}$ [21] (see [3, Thm. 4.17]) for all $\mathcal{O}(n^z)$ subsets $C \subseteq \chi_V^{-1}(\dot{\mathsf{C}})$ of size at most z. Hence, the overall runtime is $n^{\mathcal{O}(z)}$.

(Correctness) We first show that the algorithm preserves the properness of recoloring, that is, if an arbitrary z-tight OCC is z-properly colored prior to the recoloring, it is also z-properly colored after the recoloring. Second, we show that output in Step 5 is necessarily a z-tight OCC in G. Under these assumptions, if χ z-properly colors a z-tight OCC, then the algorithm must output a z-tight OCC (A_B, A_C, A_R) in G such that for every z-properly colored OCC $(\hat{A}_B, \hat{A}_C, \hat{A}_R)$, we have $\hat{A}_B \subseteq A_B$ and $\hat{A}_C \subseteq A_C$ as it does not exclude any z-properly colored z-tight OCCs. And if χ does not z-properly color any z-tight OCCs, the algorithm should correctly report the absence of z-properly colored z-tight OCCs in Step 6.

Claim 3.32. All z-tight OCCs $(\hat{A}_B, \hat{A}_C, \hat{A}_R)$ that are z-properly colored by χ prior to executing the algorithm are also z-properly colored by χ after termination of the algorithm.

Proof. Suppose an arbitrary z-tight OCC $(\hat{A}_B, \hat{A}_C, \hat{A}_R)$ is z-properly colored by χ. Then, $\hat{A}_C \subseteq \chi_V^{-1}(\dot{\mathsf{C}})$, $\hat{A}_B \subseteq \chi_V^{-1}(\dot{\mathsf{B}})$, and $G' = G[\hat{A}_B \cup \hat{A}_C] - \chi^{-1}(\dot{\mathsf{R}})$ is an $\hat{A}_C$-certificate of order z. We will show that (1) if a vertex v is recolored, then $v \notin \hat{A}_B \cup \hat{A}_C$ and (2) edges in $G[\hat{A}_B \cup \hat{A}_C]$ are never recolored. With these conditions, we see that $(\hat{A}_B, \hat{A}_C, \hat{A}_R)$ is z-properly colored by χ at any time during the algorithm.

To show (1), analyze each step where a vertex is recolored. Suppose a vertex v is recolored in Step 2. Since χ z-properly colors $(\hat{A}_B, \hat{A}_C, \hat{A}_R)$, we have $N(\hat{A}_B) \subseteq \hat{A}_C \subseteq \chi_V^{-1}(\dot{\mathsf{C}})$. We know that $\chi(v) = \dot{\mathsf{B}}$, so if $v \in \hat{A}_B$, then the component B of $G[\chi_V^{-1}(\dot{\mathsf{B}})]$ with $v \in B$ must be entirely included in $\hat{A}_B$. This implies that $G[B]$ is bipartite and $N(B) \subseteq N(\hat{A}_B) \subseteq \chi_V^{-1}(\dot{\mathsf{C}})$, contradicting the recoloring condition in Step 2.

Next, suppose a vertex v is recolored in Step 4. We know that $\chi(v) = \dot{\mathsf{C}}$, and v was not marked during Step 3. Assume for the contradiction that $v \in \hat{A}_C$. Since G' is an $\hat{A}_C$-certificate of order z, there exists a component H of G' such that $v \in H$ and $\mathsf{oct}(H) = |\hat{A}_C \cap V(H)| \leq z$. From $\hat{A}_C \cap V(H) \subseteq \hat{A}_C \subseteq \chi_V^{-1}(\dot{\mathsf{C}})$, we know that in some iteration in Step 3 we have $C = \hat{A}_C \cap V(H)$. Vertex v was marked if $\mathsf{oct}(G_\chi[C \cup W_\chi(C)]) = |C|$, which we want to show for a contradiction. First, notice that $W_\chi(C) \subseteq \chi^{-1}(\dot{\mathsf{B}})$, and at this point $\chi^{-1}(\dot{\mathsf{B}})$ is bipartite. Hence, $\mathsf{oct}(G_\chi[C \cup W_\chi(C)]) \leq |C|$. Next, we show that H is a subgraph of $G_\chi[C \cup W_\chi(C)]$. It is clear to see that a component H' in $H - C$ is also a component in $G_\chi[\chi_V^{-1}(\dot{\mathsf{B}})]$. Since H is a component of G', $N_{G'}(H') \subseteq \hat{A}_C \cap V(H) = C$, and hence $N_{G'}(V(H - C)) \subseteq C$. Also, because H is connected in G', we have $V(H - C) \subseteq W_\chi(C)$, implying that H is a subgraph of $G_\chi[C \cup W_\chi(C)]$. We have $\mathsf{oct}(G_\chi[C \cup W_\chi(C)]) \geq \mathsf{oct}(H) = |C|$.

For (2), edge recoloring takes place only when the edge is incident to a vertex colored $\dot{\mathsf{R}}$. Every edge e in $G[\hat{A}_B \cup \hat{A}_C]$ is not incident to any vertex in $\chi_V^{-1}(\dot{\mathsf{R}})$, so e cannot be recolored. ◁

Claim 3.33. In Step 5, $(\chi_V^{-1}(\dot{\mathsf{B}}), \chi_V^{-1}(\dot{\mathsf{C}}), \chi_V^{-1}(\dot{\mathsf{R}}))$ is a z-tight OCC in G.

Proof. In this proof, for any family of sets $X_1, \ldots, X_\ell$ indexed by $\{1, \ldots, \ell\}$ we define the following for all $1 \leq i \leq \ell$: $X_{<i} := \bigcup_{1 \leq j < i} X_j$ and $X_{\leq i} := \bigcup 1 \leq j \leq i X_j$.

From Step 2, we know that $G[\chi_V^{-1}(\dot{B})]$ is bipartite and $\chi_V^{-1}(\dot{B}) \subseteq \chi_V^{-1}(\dot{C})$. Also, Step 5 guarantees that $\chi_V^{-1}(\dot{B}) \cup \chi_V^{-1}(\dot{C}) \neq \emptyset$, so $(\chi_V^{-1}(\dot{B}), \chi_V^{-1}(\dot{C}), \chi_V^{-1}(\dot{R}))$ is an OCC in G. It remains to show that $G[\chi_V^{-1}(\dot{B}) \cup \chi_V^{-1}(\dot{C})]$ contains a $\chi_V^{-1}(\dot{C})$-certificate of order z.

Let $\mathcal{C} \subseteq 2^{\chi_V^{-1}(\dot{C})}$ be the family of all subsets $C \subseteq \chi_V^{-1}(\dot{C})$ that have been considered and marked in Step 3, i.e., $\mathsf{oct}(G_\chi[C \cup W_\chi(C)]) = |C| \leq z$ for every $C \in \mathcal{C}$. Let $C_1, \ldots, C_{|\mathcal{C}|}$ be the sets in $\mathcal{C}$ in an arbitrary order and define $D_i := C_i \setminus C_{<i}$ for all $1 \leq i \leq |\mathcal{C}|$. Also define vertex-disjoint subgraphs $G_i := G_\chi[D_i \cup (W_\chi(D_{\leq i})) \setminus W_\chi(D_{<i})]$ for all $1 \leq i \leq |\mathcal{C}|$. For any $i < j$, we have $D_i \cap D_j = \emptyset$ by definition, and $(W_\chi(D_{\leq i}) \setminus W_\chi(D_{<i})) \cap (W_\chi(D_{\leq j}) \setminus W_\chi(D_{<j})) = \emptyset$ because $W_\chi(D_{\leq i}) \subseteq W_\chi(D_{<j})$. Hence, $V(G_i) \cap V(G_j) = \emptyset$.

Because we have $\mathsf{oct}(G_\chi[C_i \cup W_\chi(C_i)]) = |C_i|$, then $\mathsf{oct}(G_\chi[C_i \cup W_\chi(C_i)] - (C_i \setminus D_i)) = \mathsf{oct}(G_\chi[D_i \cup W_\chi(C_i)]) = |D_i|$. Furthermore, in $G_\chi[D_i \cup W_\chi(C_i)]$, the vertices $W_\chi(C_i) \cap W_\chi(C_{<i})$ induce a bipartite graph and are disconnected from $D_i \cup W_\chi(C_i) \setminus W_\chi(C_{<i})$; by construction, there are no edges between D_i and $W_\chi(C_{<i})$ because a vertex $v \in C$ adjacent to $W_\chi(C_{<i})$ must belong to $C_{<i}$, and there are no edges between $W_\chi(C_i)$ and $W_\chi(C_{<i})$ as they form different components in $G\chi[\chi_V^{-1}(\dot{B})]$. Hence, we have $\mathsf{oct}(G_\chi[D_i \cup W_\chi(C_i)]) = \mathsf{oct}(G_\chi[D_i \cup W_\chi(C_i)] - (W_\chi(C_{<i}))) = \mathsf{oct}(G_i) = |D_i| \leq |C_i| \leq z$. The disjoint union of G_i contains a $(\bigcup_i D_i = \chi_V^{-1}(\dot{C}))$-certificate of order z. ◁

This concludes the proof of Lemma 3.31. □

Combining all ingredients in the previous sections leads to a proof of the main theorem.

Theorem 3.1. *There is a deterministic algorithm that, given a graph G and integers $k \geq z \geq 0$, runs in $2^{\mathcal{O}(k^{33}z^2)} \cdot n^{\mathcal{O}(z)}$ time and either outputs at least k vertices that belong to an optimal solution for* Odd Cycle Transversal, *or concludes that G does not contain a z-tight OCC of width k.*

Proof. Consider the following algorithm.

1. Use the algorithm from Lemma 3.23 to either obtain a reducible OCC (X_B, X_C, X_R) of width at most k or determine if there is no single-component OCC of width at most k with $|X_B| > g_r(2k)$ in $2^{\mathcal{O}(k^{16})} n^{\mathcal{O}(1)}$ time.

2. If we obtain a reducible OCC in Step 1, then apply reductions in $2^{\mathcal{O}(k)} n^{\mathcal{O}(1)}$ time as described in Lemma 3.9, and continue to Step 1.

3. Let G' be the reduced graph. Create a $(|V(G')| + |E(G')|, g(k,z), 3)$-universal function family $\mathcal{F}$ for $V(G') \cup E(G') \to \{\dot{\mathsf{B}}, \dot{\mathsf{C}}, \dot{\mathsf{R}}\}$ using Corollary 3.11, where we set $g(k,z) = 2kz^2(g_r(2k))^2$. Each function $\chi \in \mathcal{F}$ represents a coloring of the vertices and edges in G'.

4. Iterate over all colorings $\chi \in \mathcal{F}$. For each χ, call the algorithm from Lemma 3.31 as a subroutine. If it outputs a z-tight OCC (A_B, A_C, A_R) with $|A_C| \geq k$, then output A_C as the final result.

5. If all colorings result in a z-tight OCC (A_B, A_C, A_R) such that $|A_C| < k$, or all colorings are concluded as having no z-tight OCCs, then report that G does not contain a z-tight OCC of width k.

(Running time) The reduction steps (Steps 1-2) take $2^{\mathcal{O}(k^{16})}n^{\mathcal{O}(1)}$ time because whenever we find a reducible OCC, the number of vertices in G decreases, resulting in at most n iterations. For Step 3, from Corollary 3.11 we can construct a family of colorings χ that cover all colorings when restricted to $g(k,z)$ elements. Since the number of vertices never increases by the reduction steps, it holds that $|V(G')| \leq n$. Steps 4-5 take $n^{\mathcal{O}(z)}$ time for each coloring, and there are $2^{\mathcal{O}(g(k,z))}\log^2 n$ colorings. The overall runtime is $2^{\mathcal{O}(g(k,z))}n^{\mathcal{O}(z)} = 2^{\mathcal{O}(k^{33}z^2)}n^{\mathcal{O}(z)}$.

(Correctness) The algorithm first reduces all reducible OCCs in Steps 1-2. Let G' be the reduced graph when the algorithm enters Step 3.

We start by arguing that the output of the algorithm is correct when it outputs the set A_C in Step 4 after having found a z-tight OCC (A_B, A_C, A_R) in G'. Observe that by definition of tight OCC, the set A_C is a subset of an optimal OCT S for G'. Then, from the backward safety (Lemma 3.28) and its transitivity, we have that $S \subseteq V(G) \cap V(G')$ and S is an optimal OCT of G. So the final result A_C in Step 4 belongs to an optimal solution for ODD CYCLE TRANSVERSAL in G.

Suppose now that G contains a z-tight OCC of width k; we argue that the algorithm will produce a suitable output in Step 4. From this, it will follow that the algorithm is correct in reporting that G does not have a z-tight OCC of width k if it terminates in Step 5. By invoking forward safety (Lemma 3.27) and its transitivity, we know that if there exists a z-tight OCC (A_B, A_C, A_R) of width k in G, then there exists a z-tight OCC (A'_B, A'_C, A'_R)

of width k in G'. By applying Lemma 3.18, we may assume without loss of generality that $G'[A'_B]$ has at most $z^2|A'_C|$ components.

For each component B' in $G'[A'_B]$, the structure $(V(B'), A'_C, A'_R \cup (A'_B \setminus V(B')))$ is a single-component OCC in G' of width k. Since the reduction process stabilized with G', we have $|V(B')| \leq g_r(2k)$. Hence each of the at most $z^2|A'_C|$ components of $G'[A'_B]$ has at most $g_r(2k)$ vertices, so that $|A'_B| \leq kz^2 \cdot g_r(2k)$.

Next, we will show that there exists a coloring of $V(G') \cup E(G')$ that properly colors (A'_B, A'_C, A'_R). To see this, consider an A'_C-certificate D' of order z in $G'[A'_C \cup A'_B]$, which exists by Definition 3.14; hence D' is a subgraph of $G'[A'_C \cup A'_B]$ for which A'_C is an optimal OCT, while each component of D' contains at most z vertices from A'_C. Then any coloring χ that assigns A'_C color $\dot{\mathsf{C}}$, assigns A'_B color $\dot{\mathsf{B}}$, assigns all edges that belong to D' color $\dot{\mathsf{B}}$ and all remaining edges of $G'[A'_B \cup A'_C]$ color $\dot{\mathsf{R}}$, will z-properly color (A'_B, A'_C, A'_R). To obtain a z-proper coloring, it therefore suffices for the coloring to act as prescribed on the vertex set $A'_B \cup A'_C$ and on the edges of the subgraph $G'[A'_B \cup A'_C]$.

Notice that a bipartite graph of n vertices may have at most $n^2/4$ edges. Hence:

$$\begin{aligned}|E(G'[A'_B \cup A'_C])| &= |E(G'[A'_B])| + |E_{G'}(A'_B, A'_C)| + |E(G'[A'_C])| \\ &\leq kz^2(g_r(2k))^2/4 + kz^2 g_r(2k) \cdot k + \binom{k}{2} \\ &\leq kz^2(g_r(2k))^2,\end{aligned}$$

where we use the fact that $|A'_C| \leq k$ and use $E_{G'}(A'_B, A'_C)$ to denote the edges of G' that have one endpoint in A'_B and the other in A'_C. This allows us to bound the number of elements whose color determines whether or not a coloring is z-proper by:

$$\begin{aligned}|A'_B \cup A'_C \cup E(G'[A'_B \cup A'_C])| &\leq kz^2 \cdot g_r(2k) + k + kz^2(g_r(2k))^2 \\ &\leq 2kz^2(g_r(2k))^2 = g(k,z).\end{aligned}$$

Since the algorithm constructs a universal function family with parameters $(|V(G')| + |E(G')|, g(k,z), 3)$, it follows that the algorithm encounters at least one z-proper coloring of (A'_B, A'_C, A'_R) if it exists. Hence the algorithm is correct if it concludes there is no z-tight OCC in G of width k. □

3.7 Hardness Results

First, we consider the NP-hardness of the following problem.

1-TIGHT OCC DETECTION

Input: A graph G.

Problem: Output a non-empty 1-tight OCC (X_B, X_C, X_R) in G or conclude that no non-empty 1-tight OCC exists.

To prove that this problem is NP-hard under Turing reductions, we consider the following version of OCT.

LOWER-BOUNDED OCT

Input: A graph G and k vertex-disjoint odd cycles $\{C_i\}$ in G.

Problem: Is there a set $S \subseteq V(G)$ of size at most k such that $G - S$ is bipartite?

We claim that 1-TIGHT OCC DETECTION is as hard as LOWER-BOUNDED OCT.

Lemma 3.34. *Assuming* $\mathsf{P} \neq \mathsf{NP}$*, if there is no polynomial-time algorithm for solving* LOWER-BOUNDED OCT*, then there is no polynomial-time algorithm for* 1-TIGHT OCC DETECTION*.*

Proof. Assume 1-TIGHT OCC DETECTION is solvable in polynomial time. Then, consider the following algorithm for LOWER-BOUNDED OCT.

1. Let $C \leftarrow \{C_i\}$. Initially, $|C| = k$.
2. Return Yes if $C = \emptyset$.
3. Run an algorithm for 1-TIGHT OCC DETECTION on G. If it finds a non-empty 1-tight OCC (X_B, X_C, X_R), then let $G \leftarrow X_R$ and $C \leftarrow \{C_i \in C \mid X_C \cap V(C_i) \neq \emptyset\}$, i.e., remove odd cycles in C that hit any vertex in X_C, and then continue to Step 2. Otherwise, return No.

Let S be the union of X_C found in Step 3. If the algorithm returns Yes, then it is clear to see that $|S| = k$ and $G - S$ is bipartite. $(G, \{C_i\})$ is a yes-instance for LOWER-BOUNDED OCT. If the algorithm returns No, then there exists an induced subgraph G' of G such that for any OCT S of G', there exists a cycle C_i such that $|V(C_i) \cap S| > 1$. Hence, $\mathsf{oct}(G) > k$,

and $(G, \{C_i\})$ is a no-instance. If 1-TIGHT OCC DETECTION is poly-time solvable, then so is LOWER-BOUNDED OCT. □

Now, let us show that LOWER-BOUNDED OCT is actually NP-hard. We adopt an argument from [6, Lemma 16] but with simpler gadgets.

Lemma 3.35. LOWER-BOUNDED OCT *is NP-hard.*

Proof. We reduce from 3-SAT, a version of SATISFIABILITY (SAT) where every clause includes exactly 3 literals, to LOWER-BOUNDED OCT. Let $X = x_1, \ldots, x_n$ be the variables and $\Phi = \phi_1, \ldots, \phi_m$ be the clauses appearing in a SAT instance.

We construct a graph G as follows:

- For each variable x_i, create a new triangle and let v_i and $\overline{v_i}$ be two of the triangle's vertices. These vertices represent SAT literals x_i and $\overline{x_i}$ as shown in Figure 3.3 (left).
- For each clause $\phi_j = \ell_{j1} \vee \ell_{j2} \vee \ell_{j3}$, introduce a gadget illustrated in Figure 3.3 (right). We then identify vertices s_1, s_2, s_3 and the vertices representing ℓ_{j1}, ℓ_{j2}, ℓ_{j3}, respectively.

It is clear to see that this construction can be done in polynomial time. Also observe that there are n disjoint triangles for SAT variables and $2m$ disjoint triangles for clause gadgets. By letting $k = n + 2m$, we have that G contains k vertex-disjoint triangles, which are odd cycles. We claim that (X, Φ) is satisfiable if and only if G contains an OCT of size at most k.

First, assume (X, Φ) is satisfiable. Let $U \subseteq V(G)$ be the vertices that correspond to a certificate, and consider all odd cycles present in $G - U$. For each variable gadget, we know that either x_i or $\overline{x_i}$ is in U, so no triangles remain in $G - U$. For each clause gadget, we know that at least one of s_1, s_2, s_3 is in U. In any cases, we can choose 2 vertices hitting all remaining triangles. Hence, $\mathsf{oct}(G - U) \leq 2m$ and $\mathsf{oct}(G) \leq n + 2m = k$.

Conversely, assume G has an OCT $S \subseteq V(G)$ of size at most k. Since G has k vertex-disjoint triangles, S contains exactly one vertex from each of them. For every variable gadget, exactly one vertex is in S. For every clause gadget excluding s_1, s_2, s_3, exactly 2 vertices must be in S, which leads to the fact that at least one of s_1, s_2, s_3 is in S. Finally,

we construct a solution for 3-SAT. For each $1 \leq i \leq n$, we set x_i to true if $v_i \in S$, and false otherwise. This by construction forms a certificate, and (X, Φ) is satisfiable. □

Combining Lemma 3.34 and Lemma 3.35, we obtain the following result.

Theorem 3.36. 1-TIGHT OCC DETECTION *is NP-hard under Turing reductions.*

Next, we show that finding a non-empty 1-tight OCC of bounded width is W[1]-hard, parameterized by the width.

BOUNDED-WIDTH 1-TIGHT OCC DETECTION

Input: A graph G and an integer k.

Parameter: k.

Problem: Does G admit a non-empty 1-tight OCC (X_B, X_C, X_R) with $|X_C| \leq k$?

We will show the W[1]-hardness of this problem by a parameterized reduction from the well-known MULTICOLORED CLIQUE problem.

MULTICOLORED CLIQUE

Input: A graph G, an integer k, and a partition of $V(G)$ into sets $V_1, \ldots, V_k$.

Parameter: k.

Problem: Does G include as a subgraph such a clique S such that for each $1 \leq i \leq k$ we have $|S \cap V_i| = 1$?

Theorem 3.37. BOUNDED-WIDTH 1-TIGHT OCC DETECTION *is W[1]-hard.*

Proof. Consider an instance (G, χ, k) of MULTICOLORED CLIQUE, where $\chi\colon V(G) \to [k]$ denotes the *color* of each vertex such that $\chi(v) = \ell$ if and only if $v \in V_\ell$. Without loss of generality, we assume $k \geq 2$ and $n > k + 2$. Then, we construct an input (G', k') for BOUNDED-WIDTH 1-TIGHT OCC DETECTION as follows (see Figure 3.4 for an illustration).

1. For each vertex $i \in V(G)$, introduce $(k-1)$ vertices $U_i := \{u_{i\ell} \mid \ell \in [k] \setminus \{\chi(i)\}\}$ to G'.

2. For each edge $ij \in E(G)$, add 4 vertices x_{ij}, x'_{ij}, y_{ij}, and y'_{ij} to G'.

3. Insert edges to turn $\bigcup_{i \in V(G)} U_i$ into a clique in G'.

4. For each edge $ij \in E(G)$, add edges from $U_i \cup U_j$ to x_{ij}, x_{ij} to x'_{ij}, and x'_{ij} to $u_{i\chi(j)}$. Similarly, add edges from $U_i \cup U_j$ to y_{ij}, y_{ij} to y'_{ij}, and y'_{ij} to $u_{j\chi(i)}$.

Let $k' = k(k-1)$, and we claim that (G', k') for BOUNDED-WIDTH 1-TIGHT OCC DETECTION MULTICOLORED CLIQUE is an equivalent instance to (G, χ, k) for MULTICOLORED CLIQUE.

Claim 3.38. If G has a multicolored clique of size k, then G' has a non-empty 1-tight OCC of width k'.

Proof. Suppose S is a multicolored clique of size k in G. Define a 1-tight OCC (X_B, X_C, X_R) as follows:

- $X_B = \bigcup_{i,j \in S: i \neq j} \{x_{ij}, x'_{ij}, y_{ij}, y'_{ij}\}$
- $X_C = \bigcup_{i \in S} U_i$
- $X_R = V(G') \setminus (X_B \cup X_C)$

It is clear to see that $|X_C| = k(k-1) = k'$. Also, by construction, we have $N(X_C) = \bigcup_{i,j \in S | i \neq j} (U_i \cup U_j \cup \{u_{i\chi(j)}, u_{j\chi(i)}\}) = \bigcup_{i \in S} U_i = X_C$, which remains us to show that $G'[X_B \cup X_C]$ contains k' vertex-disjoint odd cycles. Notice that $G'[X_B \cup X_C]$ contains the set of triangles $T = \bigcup_{i,j \in S | i \neq j} \{u_{i\chi(j)} x_{ij} x'_{ij}, u_{j\chi(i)} y_{ij} y'_{ij}\}$. Since $|T| = \binom{k}{2} \cdot 2 = k'$ and $x_{ij}, x'_{ij}, y_{ij}, y'_{ij}$ are unique in T, it suffices to show that $\left|\bigcup_{i,j \in S | i \neq j} \{u_{i\chi(j)}, u_{j\chi(i)}\}\right| = k'$. Since S is a multicolored clique, for every pairwise distinct $i, j \in S$, we have $\chi(i) \neq \chi(j)$. Therefore, $\bigcup_{i,j \in S | i \neq j} \{u_{i\chi(j)}, u_{j\chi(i)}\} \supseteq \bigcup_{i \in S} \left(\bigcup_{j \in S \setminus \{i\}} u_{i\chi(j)}\right) = \bigcup_{i \in S} \bigcup_{\ell \in [k] \setminus \{\chi(i)\}} u_{i\ell} = \bigcup_{i \in S} U_i$. T indeed consists of $|\bigcup_{i \in S} U_i| = k(k-1) = k'$ vertex-disjoint triangles, and (X_B, X_C, X_R) is a 1-tight OCC. ◁

Finally, we show the backward implication.

Claim 3.39. If G' has a non-empty 1-tight OCC of size k', then G has a multicolored clique of size k.

Proof. Let (X_B, X_C, X_R) be a non-empty 1-tight OCC of size k' in G'. We write U for $\bigcup_{i \in S} U_i$ and W for $\bigcup_{e \in E(G)} \{x_e, x'_e, y_e, y'_e\}$.

First, we show that $X_B \subseteq W$. Assume not. Then, there exists a vertex $u \in X_B \cap U$. Since X_B is bipartite and U forms a clique in G', we have $|X_B \cap U| \leq 2$. We know that $N_{G'}(u) \subseteq X_B \cup X_C$, and thus $|N_{G'}(u) \setminus X_B| \leq |X_C|$. We have: $|N_{G'}(u) \setminus X_B| \geq |N_{G'}(u)| - 1 \geq (|U| - 1) - 1 = n(k-1) - 2 > ((k+2)(k-1) - 2) = k^2 + k - 4 \geq k^2 - k = k'$. This implies $|X_C| > k'$, a contradiction.

Next, we show that $X_C \subseteq U$. For the sake of contradiction, assume there exists a vertex $x \in X_C \cap W$. Then, there must be an odd cycle in $G'[\{x\} \cup X_B]$, but because $\{x\} \cup X_B \subseteq W$ is bipartite, we reach a contradiction.

Now that we know $X_B \subseteq W$ and $X_C \subseteq U$, for each $u \in X_B$, there is an odd cycle in $G'[\{u\} \cup X_B]$. This happens only when $ux_e x'_e$ or $uy_e y'_e$ forms a triangle for some $e \in E(G)$, which leads to $|X_B| \geq 2|X_C|$.

Let $E' = \{e \in E(G) \mid \{x_e, y_e\} \cap X_B \neq \emptyset\}$. If $ij \in E'$, then all the vertices in $C_i \cup C_j$ are in X_C, so we can write $X_C = \bigcup_{i \in V'} C_i$ for some $V' \subseteq V(G)$. Furthermore, from $|X_C| = k' = k(k-1)$, we know $|V'| = k$. From $E' \subseteq \binom{V'}{2}$, we have $|X_B| \leq 4|E'| \leq 2|V'|(|V'| - 1) = 2k'$. On the other hand, $|X_B| \geq 2|X_C| = 2k'$. These inequalities hold only when $E' = \binom{V'}{2}$.

We claim that V' is a multicolored clique in G. Let ij be an edge in E'. By construction, triangles $u_{i\chi(j)} x_{ij} x'_{ij}$ and $u_{j\chi(i)} y_{ij} y'_{ij}$ are in $G'[X_B \cup X_C]$, and thus $\chi(i) \neq \chi(j)$. Since V' forms a clique in G such that any vertex pair has different colors, V' is a multicolored clique. $\triangleleft$

From Claims 3.38 and 3.39, we conclude that (G, χ, k) and (G', k') are equivalent, and BOUNDED-WIDTH 1-TIGHT OCC DETECTION is W[1]-hard. $\square$

3.8 Conclusion

Inspired by crown decompositions for VERTEX COVER and antler decompositions for FEEDBACK VERTEX SET, we introduced the notion of (tight) odd cycle cuts to capture local regions of a graph in which a simple certificate exists for the membership of certain vertices in an optimal solution to ODD CYCLE TRANSVERSAL. In addition, we developed a fixed-parameter tractable algorithm to find a non-empty subset of vertices that belong to an optimal odd cycle transversal in input graphs admitting a tight odd cycle cut; the parameter k we employed is the *width* of the tight OCC. Finding tight odd cycle cuts and removing the vertices certified to be in an optimal solution leads to search-space reduction for the natural parameterization of ODD CYCLE TRANSVERSAL. To obtain our

results, one of the main technical ideas was to replace the use of minimum two-way separators that arise naturally when solving ODD CYCLE TRANSVERSAL, by minimum three-way separators that simultaneously handle breaking the odd cycles in a subgraph *and* separating the resulting local bipartite subgraph from the remainder of the graph.

3.8.0.1 Theoretical challenges. There are several interesting directions for follow-up work. We first discuss the theoretical challenges. The algorithm we presented runs in time $2^{k^{\mathcal{O}(1)}} n^{\mathcal{O}(z)}$, where z is the order of the tight odd cycle cut in the output guarantee of Theorem 3.1. The polynomial term in the exponent has a large degree, which is related to the size of the cut covering sets used to shrink the bipartite part of an odd cycle cut in terms of its width. While we expect that some improvements can be made by a more refined analysis, it would be more interesting to see whether an algorithmic approach that avoids color coding can lead to significantly faster algorithms.

An odd cycle cut (X_B, X_C, X_R) of width $|X_C| = k$ in a graph G gives rise to a k-secluded bipartite subgraph $G[X_B]$; recall that a subgraph is called k-secluded if its open neighborhood has size k. For enumerating inclusion-maximal *connected* k-secluded subgraphs that satisfy a property Π, a bounded-depth branching strategy was recently proposed [12] that generalizes the enumeration of important separators. Can such branching techniques be used to improve the running time for the search-space reduction problem considered in this chapter to $2^{\mathcal{O}(k)} n^{\mathcal{O}(z)}$?

The dependence on the complexity z of the certificate is another topic for further investigation. The search-space reduction algorithm for FEEDBACK VERTEX SET by Donkers and Jansen [5] that inspired this work, also incurs a factor $n^{\mathcal{O}(z)}$ in its running time. For FEEDBACK VERTEX SET, it is conjectured but not proven that such a dependence on z is unavoidable. The situation is the same for ODD CYCLE TRANSVERSAL. Is there a way to rule out the existence of an algorithm for the task of Theorem 3.1 that runs in time $f(k, z) \cdot n^{\mathcal{O}(1)}$?

A last theoretical challenge concerns the definition of the substructures that are used to certify membership in an optimal odd cycle transversal. Our definition of an odd cycle cut (X_B, X_C, X_R) prohibits the existence of any edges between X_B and X_R. Together with the requirement that $G[X_B]$ is bipartite, this ensures that all odd cycles intersecting X_B are intersected by X_C. In principle, one could also obtain the latter conclusion from a slightly

less restricted graph decomposition. Since any odd cycle enters a bipartite subgraph on one edge and leaves via another, knowing that each connected component H of $G[X_B]$ is connected to X_R by at most one edge is sufficient to guarantee that all odd cycles visiting X_B are intersected by X_C. The prior work on antler structures for FEEDBACK VERTEX SET allows the existence of one pendant edge per component, and manages to detect such antler structures efficiently. It would be interesting to see whether our approach can be generalized for *relaxed* odd cycle cuts in which each component of $G[X_B]$ has at most one edge to X_R. To adapt to this setting, one would have to refine the type of three-way separation problem that is used in the graph reduction step.

For ODD CYCLE TRANSVERSAL, one could relax the definition of the graph decomposition even further: to ensure that odd cycles visiting X_B are intersected by X_C, it would suffice for each connected component H of $G[X_B]$ to have at most one neighbor v_H in X_R, as long as all vertices of H adjacent to v_H belong to the *same* side of a bipartition of H.

3.8.0.2 Practical challenges. Since the investigation of search-space reduction is inspired by practical considerations, we should not neglect to discuss practical aspects of this research direction. While we do not expect the algorithm as presented here to be practical, it serves as a proof of concept that rigorous guarantees on efficient search-space reduction can be formulated. Our work also helps to identify the types of substructures that can be used to reason locally about membership in an optimal solution. Apart from finding faster algorithms in theory and experimenting with their results, one could also target the development of specialized algorithms for concrete values of k and z.

For $k = 1$, a tight odd cycle cut of width 1 effectively consists of a cutvertex c of the graph whose removal splits off a bipartite connected component B but for which the subgraph induced by $B \cup \{v\}$ contains an odd cycle. Preliminary investigations suggest that in this case, an algorithm that computes the block-cut tree, analyzes which blocks form non-bipartite subgraphs, and which cut vertices break all the odd cycles in their blocks, can be engineered to run in time $\mathcal{O}(|V(G)| + |E(G)|)$ to find a vertex v belonging to an optimal odd cycle transversal when given a graph that has a tight odd cycle cut of width $k = 1$. Do linear-time algorithms exist for $k > 1$? These would form valuable reduction steps in algorithms solving ODD CYCLE TRANSVERSAL exactly, such as the one developed by Wernicke [23].

The $k = 1$ case of the *relaxed* odd cycle cuts described above are in fact used as one of the reduction rules in Wernicke's algorithm [23, Rule 7]. His reduction applies whenever there is a triangle $\{u, v, w\}$ in which w has degree two and v has degree at most three. Under these circumstances, there is an optimal solution that contains u while avoiding v and w: since the removal of u decreases the degree of w to one, while w is one of the at most two remaining neighbors of v, the removal of u breaks all odd cycles intersecting $\{u, v, w\}$. This corresponds to the fact that the triple $(X_B = \{v, w\}, X_C = \{u\}, X_R = V(G) \setminus \{u, v, w\})$ forms a tight *relaxed* odd cycle cut. We interpret the fact that the $k = 1$ case was developed naturally in an existing algorithm as encouraging evidence that refined research into search-space reduction steps can eventually lead to impact in practice.

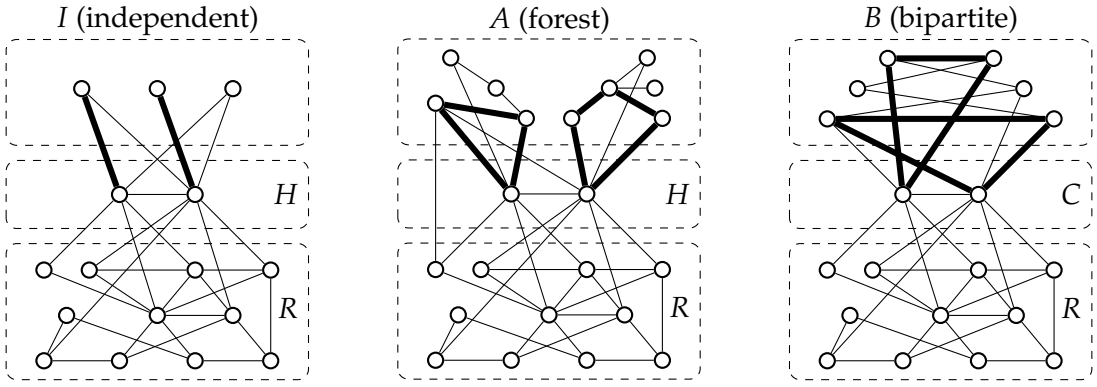


Figure 3.1: Examples of crown decomposition (left), antler decomposition for FEEDBACK VERTEX SET (middle), and a tight OCC for ODD CYCLE TRANSVERSAL (right). Packings of forbidden subgraphs are highlighted in bold.

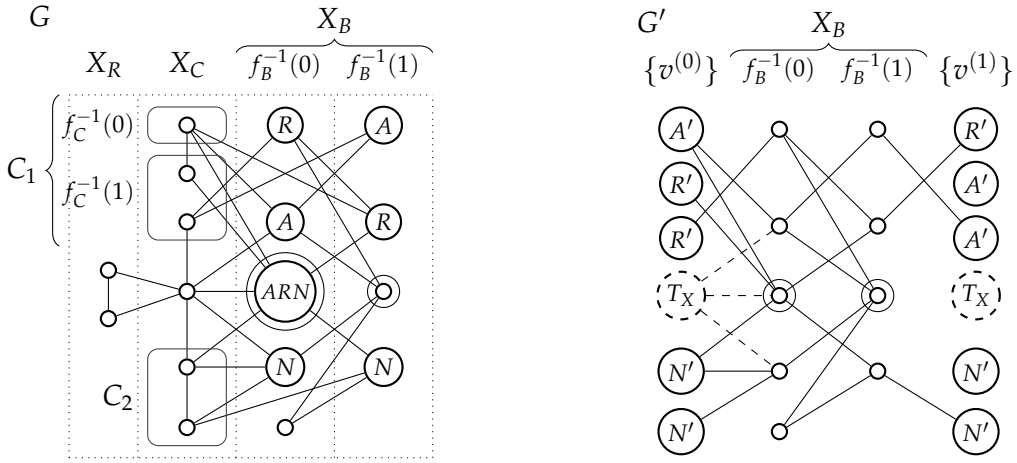


Figure 3.2: An illustration of the auxiliary graph used in the proof of Lemma 3.26. The left figure shows an example graph G with the given OCC (X_B, X_C, X_R), disjoint sets $C_1, C_2 \subseteq X_C$, a proper 2-coloring $f_B : X_B \to \{0,1\}$, and another (not necessarily proper) 2-coloring $f_C : C_1 \to \{0,1\}$. Possibly overlapping terminals A, R, N are determined as described in the proof. The right shows the auxiliary graph G' for G, constructed from a copy of $G[X_B]$ with additional $2|X_C|$ vertices. Terminals are partitioned into (A', R', N', T_X), where T_X is deleted when we examine restricted 3-way cuts. For both figures, the minimum 3-way separators (in $G[X_B]$ and $G' - T_X$ (restricted), resp.) are shaped in double circles.

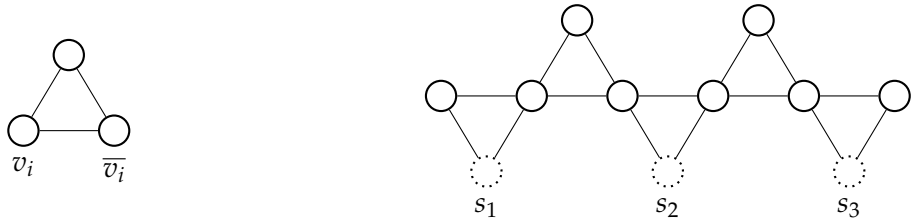


Figure 3.3: Gadgets used for the proof of Lemma 3.35. The left shows a *variable gadget*, including two vertices representing literals. The right shows a *clause gadget*, where s_1, s_2, s_3 are connected to corresponding literals.

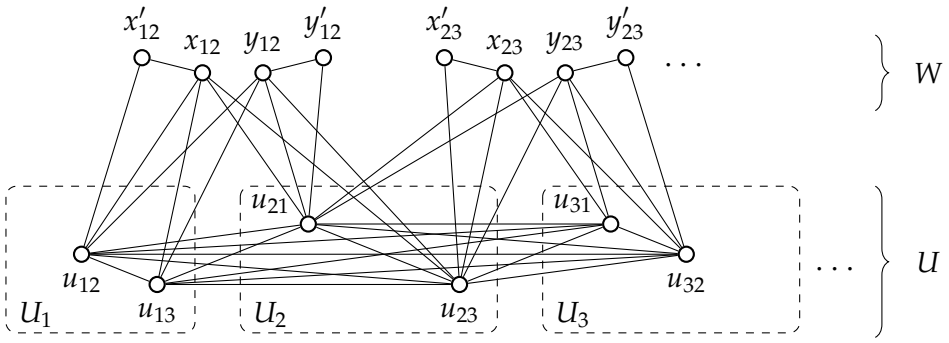


Figure 3.4: A visualization of an auxiliary graph constructed from an instance (G, k) of MULTICOLORED CLIQUE. The set U consists of $n(k-1)$ vertices representing the vertices in G and their adjacent colors. The set W consists of $4m$ vertices representing the edges in G replaced by a gadget for creating odd cycles.

REFERENCES

[1] F. N. Abu-Khzam, M. R. Fellows, M. A. Langston, and W. H. Suters, *Crown structures for vertex cover kernelization*, Theory Comput. Syst., 41 (2007), pp. 411–430.

[2] B. M. Bumpus, B. M. P. Jansen, and J. J. H. de Kroon, *Search-space reduction via essential vertices*, in Proceedings of the 30th Annual European Symposium on Algorithms, ESA 2022, S. Chechik, G. Navarro, E. Rotenberg, and G. Herman, eds., vol. 244 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 30:1–30:15.

[3] M. Cygan, F. V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh, *Parameterized Algorithms*, Springer, 2015.

[4] M. Cygan, M. Pilipczuk, M. Pilipczuk, and J. O. Wojtaszczyk, *On multiway cut parameterized above lower bounds*, ACM Trans. Comput. Theory, 5 (2013), pp. 3:1–3:11.

[5] H. Donkers and B. M. Jansen, *Preprocessing to reduce the search space: Antler structures for feedback vertex set*, Journal of Computer and System Sciences, 144 (2024).

[6] H. Donkers and B. M. P. Jansen, *A Turing Kernelization Dichotomy for Structural Parameterizations of -Minor-Free Deletion*, in Graph-Theoretic Concepts in Computer Science: 45th International Workshop, WG 2019, Vall de Núria, Spain, June 19–21, 2019, Revised Papers, Berlin, Heidelberg, June 2019, Springer-Verlag, pp. 106–119.

[7] M. R. Fellows, *Blow-ups, win/win's, and crown rules: Some new directions in FPT*, in Proceedings of the 29th International Workshop on Graph-theoretic Concepts in Computer Science, WG 2003, H. L. Bodlaender, ed., vol. 2880 of Lecture Notes in Computer Science, Springer, 2003, pp. 1–12.

[8] T. D. Goodrich, E. Horton, and B. D. Sullivan, *An updated experimental evaluation of graph bipartization methods*, ACM J. Exp. Algorithmics, 26 (2021), pp. 12:1–12:24.

[9] T. D. Goodrich, B. D. Sullivan, and T. S. Humble, *Optimizing adiabatic quantum program compilation using a graph-theoretic framework*, Quantum Information Processing, 17 (2018).

[10] F. Hüffner, *Algorithm engineering for optimal graph bipartization*, J. Graph Algorithms Appl., 13 (2009), pp. 77–98.

[11] B. M. P. Jansen and J. J. H. de Kroon, *FPT algorithms to compute the elimination distance to bipartite graphs and more*, in Graph-Theoretic Concepts in Computer Science - 47th International Workshop, WG 2021, Warsaw, Poland, June 23-25, 2021, Revised Selected Papers, L. Kowalik, M. Pilipczuk, and P. Rzazewski, eds., vol. 12911 of Lecture Notes in Computer Science, Springer, 2021, pp. 80–93.

[12] B. M. P. JANSEN, J. J. H. DE KROON, AND M. WLODARCZYK, *Single-exponential FPT algorithms for enumerating secluded $\mathcal{F}$-free subgraphs and deleting to scattered graph classes*, in Proceedings of the 34th International Symposium on Algorithms and Computation, ISAAC 2023, S. Iwata and N. Kakimura, eds., vol. 283 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 42:1–42:18.

[13] B. M. P. JANSEN, J. J. H. DE KROON, AND M. WŁODARCZYK, *Vertex Deletion Parameterized by Elimination Distance and Even Less*, 2022. arXiv:2103.09715 [cs].

[14] B. M. P. JANSEN, Y. MIZUTANI, B. D. SULLIVAN, AND R. F. A. VERHAEGH, *Preprocessing to reduce the search space for odd cycle transversal*, 2024.

[15] B. M. P. JANSEN AND R. F. A. VERHAEGH, *Search-space reduction via essential vertices revisited: Vertex multicut and cograph deletion*, in Proceedings of the 19th Scandinavian Symposium on Algorithm Theory, SWAT 2024, H. L. Bodlaender, ed., LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. In press.

[16] S. KRATSCH AND M. WAHLSTRÖM, *Compression via matroids: A randomized polynomial kernel for odd cycle transversal*, ACM Trans. Algorithms, 10 (2014), pp. 20:1–20:15.

[17] ——, *Representative sets and irrelevant vertices: New tools for kernelization*, J. ACM, 67 (2020), pp. 16:1–16:50.

[18] D. LOKSHTANOV, N. S. NARAYANASWAMY, V. RAMAN, M. S. RAMANUJAN, AND S. SAURABH, *Faster parameterized algorithms using linear programming*, ACM Trans. Algorithms, 11 (2014), pp. 15:1–15:31.

[19] M. NAOR, L. J. SCHULMAN, AND A. SRINIVASAN, *Splitters and near-optimal derandomization*, in 36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, USA, 23-25 October 1995, IEEE Computer Society, 1995, pp. 182–191.

[20] M. PILIPCZUK AND M. ZIOBRO, *Experimental evaluation of parameterized algorithms for graph separation problems: Half-integral relaxations and matroid-based kernelization*, CoRR, abs/1811.07779 (2018).

[21] B. A. REED, K. SMITH, AND A. VETTA, *Finding odd cycle transversals*, Operations Research Letters, 32 (2004), pp. 299–301.

[22] A. SCHRIJVER, *Combinatorial optimization: polyhedra and efficiency*, vol. 24, Springer, 2003.

[23] S. WERNICKE, *On the algorithmic tractability of single nucleotide polymorphism (SNP) analysis and related problems*, diplom.de, 2014.

CHAPTER 4

PARAMETERIZED COMPLEXITY OF HAPPY SET PROBLEMS

So far, we have seen algorithms for natural parameters. However, it is also common to parameterize a problem by structural graph parameters. In this chapter, we present fixed-parameter tractable (FPT) algorithm for two problems, MAXIMUM HAPPY SET (MAXHS) and DENSEST k-SUBGRAPH (DkS)—also known as MAXIMUM EDGE HAPPY SET. Given a graph $G = (V, E)$ and an integer k, MAXHS asks for a set S of k vertices such that the number of *happy vertices* with respect to S is maximized, where a vertex v is happy if v and all its neighbors are in S. We show that MAXHS can be solved in time $\mathcal{O}\left(2^{\mathsf{mw}} \cdot \mathsf{mw} \cdot k^2 \cdot |V|\right)$ and $\mathcal{O}\left(8^{\mathsf{cw}} \cdot k^2 \cdot |V|\right)$, where mw and cw denote the *modular-width* and the *clique-width* of G, respectively. This answers the open questions on fixed-parameter tractability posed in [1]. The DkS problem asks for a subgraph with k vertices maximizing the number of edges. If we define *happy edges* as the edges whose endpoints are in S, then DkS can be seen as an edge-variant of MAXHS. In this work, we show that DkS can be solved in time $f(\mathsf{nd}) \cdot |V|^{\mathcal{O}(1)}$ and $\mathcal{O}(2^{\mathsf{cd}} \cdot k^2 \cdot |V|)$, where nd and cd denote the *neighborhood diversity* and the *cluster deletion number* of G, respectively, and f is some computable function. This result implies that DkS is also fixed-parameter tractable by *twin cover number*.

This project began as a class project in CS 6958, taught by Dr. Sullivan, where my team explored several open problems in the field. Later, Sullivan and I turned our results into the conference paper "Parameterized Complexity of Maximum Happy Set and Densest k-Subgraph", which appeared at International Symposium on Parameterized and Exact Computation (IPEC) '22 [21]. I was single-handedly in charge of writing the manuscript draft.

4.1 Introduction

In the study of large-scale networks, *communities*—cohesive subgraphs in a network—play an important role in understanding complex systems and appear in sociology, biology and computer science, etc. [11, 18]. For example, the concept of homophily in sociology explains the tendency for individuals to associate themselves with similar people [20]. Homophily is a fundamental law governing the structure of social networks, and finding groups of people sharing similar interests has many real-world applications [8].

People have attempted to frame this idea as a graph optimization problem, where a vertex represents a person and an edge corresponds to some relation in the social network. The notion of *happy* vertices was first introduced by Zhang and Li in terms of graph coloring [24], where each color represents an attribute of a person (possibly fixed). A vertex is *happy* if all of its neighbors share its color. The goal is to maximize the number of happy vertices by changing the color of unfixed vertices, thereby achieving the greatest social benefit.

Later, Asahiro *et al.* introduced MAXIMUM HAPPY SET (MAXHS) which defines that a vertex v is *happy* with respect to a *happy set* S if v and all of its neighbors are in S [1]. The MAXHS problem asks for a vertex set S of size k that maximizes the number of happy vertices. They also define its edge-variant, MAXIMUM EDGE HAPPY SET (MAXEHS) which maximizes the number of *happy edges*, an edge with both endpoints in the *happy set*. It is clear to see that MAXEHS is equivalent to choosing a vertex set S such that the number of edges in the induced subgraph on S is maximized. This problem is known as DENSEST k-SUBGRAPH (DkS) in other literature. Both MAXHS and MAXEHS are NP-hard [1, 9], 9and we study their parameterized complexity throughout this chapter.

4.1.1 Parameterized Complexity and Related Work

Graph problems are often studied with a variety of structural parameters in addition to natural parameters (size k of the happy set in our case). Specifically, we investigate the parameterized complexity with respect to *modular-width* (mw), *clique-width* (cw), *neighborhood diversity* (nd), *cluster deletion number* (cd), *twin cover number* (tc), *treewidth* (tw), and *vertex cover number* (vc), all of which we define in Section 2.3. Figure 4.1 illustrates the hierarchy

of these parameters by inclusion; hardness results are implied along the arrows, and FPT[1] algorithms are implied in the reverse direction.

Asahiro *et al.* showed that MAXHS is W[1]-hard with respect to k by a parameterized reduction from the q-CLIQUE problem [1]. They also presented FPT algorithms for MAXHS on parameters: clique-width plus k, neighborhood diversity, cluster deletion number (which implies FPT by twin cover number), and treewidth.

MAXEHS (i.e., DkS) has been extensively studied in different names (for example, the k-CLUSTER problem [7], the HEAVIEST UNWEIGHTED SUBGRAPH problem [16], and k-CARDINALITY SUBGRAPH problem [5]). For parameterized complexity, Cai showed the W[1]-hardness parameterized by k [6]. Bourgeois employed Moser's technique in [22] to show that MAXEHS can be solved in time $2^{\mathsf{tw}} \cdot n^{\mathcal{O}(1)}$ [3]. Broersma *et al.* proved that MAXEHS can be solved in time $k^{\mathcal{O}(\mathsf{cw})} \cdot n$, but it cannot be solved in time $2^{o(\mathsf{cw} \log k)} \cdot n^{\mathcal{O}(1)}$ unless the Exponential Time Hypothesis (ETH) fails [4]. To the best of our knowledge, the parameterized complexity by modular-width, neighborhood diversity, cluster deletion number and twin cover number remained open prior to our work. Figure 4.2 summarizes the known and established hardness results for MAXHS and MAXEHS.

4.1.2 Our Contributions

In this work, we present four novel parameterized algorithms for MAXHS and MAXEHS. First, we shall provide a dynamic-programming algorithm that solves MAXHS in time $\mathcal{O}(2^{\mathsf{mw}} \cdot \mathsf{mw} \cdot k^2 \cdot |V|)$, answering the question posed by the authors of [1]. Second, we show that MAXHS is FPT by clique-width, giving an $\mathcal{O}(8^{\mathsf{cw}} \cdot k^2 \cdot |V|)$ algorithm, which removes the exponential term of k from the best known result, $\mathcal{O}(6^{\mathsf{cw}} \cdot k^{2(\mathsf{cw}+1)} \cdot |V|)$ [1]. While bounded modular-width implies bounded clique-width (Proposition 1.10), we give both algorithms because the one for modular-width has asymptotically faster running time.

Turning to MAXEHS, we prove it is FPT by neighborhood diversity, using an integer quadratic programming formulation. Lastly, we provide an FPT algorithm for MAXEHS parameterized by cluster deletion number with running time $\mathcal{O}(2^{\mathsf{cd}} \cdot k^2 \cdot |V|)$, which

[1] An FPT (fixed-parameter tractable) algorithm solves the problem in time $f(k) \cdot n^{\mathcal{O}(1)}$ for some computable function f.

also implies the problem is FPT by twin cover number. These new results complete the previously-open parameterized complexities in Figure 4.2, except the one of DkS parameterized by modular-width.

4.1.2.1 Independent work. Independently and simultaneously, Hanaka also showed the parameterized complexity of DkS by neighborhood diversity and cluster deletion number [15]. For neighborhood diversity, the complexity was implied by [19], as we discuss in Sections 4.3.3 and 4.5.1. Further, the parameterized complexity by cluster deletion number was shown by an algorithm solving DkS in time $2^{\mathsf{bd}}\left((k^3+\mathsf{bd})\,|V|+|E|\right)$, where bd denotes the block deletion number (note that it holds $\mathsf{cd} \leq \mathsf{bd}$).

4.2 Preliminaries

We use standard graph theory notation as in Section 1.3.1. Also, refer to Section 1.3.4 for the structural graph parameters considered in this chapter.

4.2.1 Problem Definitions

Asahiro *et al.* first introduced the MAXIMUM HAPPY SET problem in [1].

MAXIMUM HAPPY SET (MAXHS)

Input: A graph $G = (V, E)$ and a positive integer k.

Problem: Find a subset $S \subseteq V$ of k vertices that maximizes the number of *happy* vertices v with $N[v] \subseteq S$.

Figure 4.3 illustrates an example instance with $k = 5$. Let us call a vertex that is not happy an *unhappy* vertex, and observe that the set of unhappy vertices is given by $N[V \setminus S]$, providing an alternative characterization of happy vertices.

Proposition 4.1. *Given a graph $G = (V, E)$ and a happy set $S \subseteq V$, the set of happy vertices is given by $V \setminus N[V \setminus S]$.*

In addition, Asahiro *et al.* define an edge variant [1]:

MAXIMUM EDGE HAPPY SET (MAXEHS)

Input: A graph $G = (V, E)$ and a positive integer k.

Problem: Find a set $S \subseteq V$ of k vertices that maximizes the number of happy edges. An edge $uv \in E$ is *happy* if and only if $\{u, v\} \subseteq S$.

It is known that MAXEHS is identical to the DENSEST k-SUBGRAPH problem (DkS), as the number of happy edges is equal to $m(G[S])$. Some literature (e.g., [10]) also phrases this problem as the dual of the SPARSEST k-SUBGRAPH problem.

4.3 Background

Before describing our algorithms, we introduce some building blocks for our argument.

4.3.1 Entire Subgraphs

Structural parameters such as modular-width and clique-width entail the join operation in their underlying construction trees. When joining two subgraphs in MAXHS, it is important to distinguish whether all the vertices in the subgraph are included in the happy set. Formally, we introduce the notion of *entire subgraphs*.

Definition 4.2. Given a graph G and a happy set S, an *entire subgraph* with respect to S is a subgraph G' of G such that $V(G') \subseteq S$.

By definition, the empty subgraph is always entire. The following lemma is directly derived from the definition of happy vertices.

Lemma 4.3. *Let G be a complete join of subgraphs G_1 and G_2. For any happy set $S \subseteq V(G)$, $V(G_1)$ contains a happy vertex only if G_2 is entire with respect to S.*

Proof. If G_2 is not entire, there must exist $v \in V(G_2)$ such that $v \notin S$. Recall Proposition 4.1, and we have $N[V(G) \setminus S] \supseteq N(v) \supseteq V(G_1)$, which implies that any vertex in $V(G_1)$ cannot be happy. □

4.3.2 Knapsack Variant with Nonlinear Values

The classic KNAPSACK problem has a number of variants, including 0-1 KNAPSACK [14] and QUADRATIC KNAPSACK [13]. In this chapter we consider another variant, where

the objective function is the sum of nonlinear functions, but the function range is limited to integers. Specifically, each item has unit weight, but its value may vary depending on the number of copies of each type of item. We also require the weight sum to be exact and call this problem f-KNAPSACK, where f stands for function.

f-KNAPSACK

Input: Given a set of n items numbered from 1 to n, a weight capacity $W \in \mathbb{Z}_0^+$ and a value function $f_i : D_i \to \mathbb{Z}_0^+$, defined on a non-negative integral domain D_i for each item i.

Problem: For every $1 \leq i \leq n$, find the number $x_i \in D_i$ of instances of item i to include in the knapsack, maximizing $\sum_{i=1}^{n} f_i(x_i)$, subject to $\sum_{i=1}^{n} x_i = W$.

We show that this problem is solvable in polynomial-time.

Lemma 4.4. *f-KNAPSACK can be solved in time $\mathcal{O}(nW^2)$.*

Proof. First, define the value $\phi[t, w]$ to be the maximum possible sum $\sum_{i=1}^{t} f_i(x_i)$, subject to $\sum_{i=1}^{t} x_i = w$ and $x_i \in D_i$ for every i. Then, perform bottom-up dynamic programming as follows.

- Initialize: $\phi[0, w] = \begin{cases} 0 & \text{if } w = 0, \\ -\infty & \text{if } w > 0 \text{ (meaning Infeasible).} \end{cases}$
- Update: $\phi[t, w] = \max_{x_t \in D_t \wedge x_t \leq w} f_t(x_t) + \phi[t-1, w - x_t]$
- Result: $\phi[n, w]$ for $0 \leq w \leq W$ is the optimal value for weight w.

The base case ($\phi[0, w]$) represents the state where no item is in the knapsack, so both the objective and weight are 0; otherwise, infeasible. For the inductive step, any optimal solution $\phi[t, w]$ can be decomposed into $f_t(x_t) + \sum_{i=1}^{t-1} f_i(x_i)$ for some x_t, and the latter term ($\sum_{i=1}^{t-1} f_i(x_i)$) must equal $\phi[t-1, w - x_t]$ by definition. We consider all possible integers x_t, and thus the algorithm is correct.

Since $0 \leq t \leq n$, $0 \leq w \leq W$, and the update takes time $\mathcal{O}(W)$, the total running time is $\mathcal{O}(nW^2)$. By using the standard technique of backlinks, one can reconstruct the solution $\{x_i\}$ within the same asymptotic running time. □

The following result is a natural by-product of the algorithm above.

Corollary 4.5. *Given an integer W, f-*KNAPSACK *for all weight capacities $0 \leq w \leq W$ can be solved in total time $\mathcal{O}(nW^2)$.*

4.3.3 Integer Quadratic Programming

For MAXEHS, we use the following known result that INTEGER QUADRATIC PROGRAMMING is FPT by the number of variables and coefficients.

INTEGER QUADRATIC PROGRAMMING (IQP)

Input: An $n \times n$ integer matrix Q, an $m \times n$ integer matrix A and an m-dimensional integer vector b.

Problem: Find a vector $x \in \mathbb{Z}^n$ minimizing $x^T Qx$, subject to $Ax \leq b$.

Proposition 4.6 (Lokshtanov [19])**.** *There exists an algorithm that given an instance of* IQP*, runs in time $f(n, \alpha)L^{\mathcal{O}(1)}$, and outputs a vector $x \in \mathbb{Z}^n$. If the input IQP has a feasible solution then x is feasible, and if the input IQP is not unbounded, then x is an optimal solution. Here α denotes the largest absolute value of an entry of Q and A, and L is the total number of bits required to encode the input.*

It is convenient to have a linear term in the objective function. This can be achieved by introducing a new variable $\hat{x} = 1$ and adding $[0, q]$ as the corresponding row in Q [19].

Corollary 4.7. *Proposition 4.6 holds if we generalize the objective function from $x^T Qx$ to $x^T Qx + q^T x$ for some n-dimensional integer vector q. Here α is the largest absolute value of an entry of Q, q and A.*

4.4 Algorithms for Maximum Happy Set

Now we describe our FPT algorithms for MAXHS with respect to modular-width and clique-width. At a high level, we employ a bottom-up dynamic programming (DP) approach on the parse-tree of a given graph, considering each node once. At each node, we use several techniques on precomputed results to update the DP table. For simplicity, our DP tables store the maximum number of happy vertices. Like other DP applications, a *certificate*, i.e., the actual happy set, can be found by using backlinks within the same asymptotic running time.

4.4.1 Parameterized by Modular-Width

We give an algorithm whose running time is singly-exponential in the modular-width, quadratic in k and linear in the graph size.

Theorem 4.8. MAXHS *is solvable in time* $\mathcal{O}(2^{\mathsf{mw}} \cdot \mathsf{mw} \cdot k^2 \cdot |V(G)|)$, *where* mw *is the modular-width of the input graph* G.

Our algorithm follows the common framework seen in [12]. Given a graph G, a parse-tree with modular-width mw can be computed in linear-time [23]. The number of nodes in the parse-tree is linear in $|V(G)|$ [12]. Our algorithm traverses the parse-tree from the bottom, considering only operation (O4), as operations (O2)-(O3) can be replaced with a single operation (O4) with at most two arguments [12]. Further, we assume $2 \leq \mathsf{mw} < k$ without loss of generality.

Each node in the parse-tree corresponds to an induced subgraph of G, which we write $\mathcal{G}$. We keep track of a table $\phi[\mathcal{G}, w]$, the maximum number of happy vertices for $\mathcal{G}$ with regard to a happy set of size w. We may assume $0 \leq w \leq k$ because we do not have to consider a happy set larger than size k. The entries of the DP table are initialized with $\phi[\mathcal{G}, w] = -\infty$. For the base case, a graph G_0 with a single vertex introduced by operation (O1), we set $\phi[G_0, 0] = 0$ and $\phi[G_0, 1] = 1$. The solution to the original problem is given by $\phi[G, k]$.

Our remaining task is to compute, given a graph substitution $\mathcal{G} = H(G_1, \ldots, G_n)$ ($n \leq \mathsf{mw}$), the values of $\phi[\mathcal{G}, w]$ provided partial solutions $\phi[G_1, \cdot], \ldots, \phi[G_n, \cdot]$. We first choose a set of entire subgraphs from $G_1, \ldots, G_n$. Then, we identify the *subgraph type* for each G_i during a graph substitution.

Definition 4.9 (subgraph type)**.** Given a graph substitution $H(G_1, \ldots, G_n)$, where $v_i \in V(H)$ is substituted by G_i, and a happy set S, we categorize each substituted subgraph G_i into the following four types.

- Type I: G_i is entire and for every j such that $v_j \in N_H(v_i)$, G_j is entire.
- Type II: G_i is not entire and for every j such that $v_j \in N_H(v_i)$, G_j is entire.
- Type III: G_i is entire and not Type I.

- Type IV: G_i is not entire and not Type II.

Intuitively, Type I and II subgraphs are surrounded by entire subgraphs in H, the *metagraph* to substitute, and Type I and III subgraphs are entire. A pictorial representation of this partition is presented in Figure 4.4. Observe that from Lemma 4.3, the subgraphs with Type III and IV cannot include any happy vertices. Further, Type II subgraphs are independent in H because their neighbors must be of Type III. This ensures that the choice of a happy set in Type II is independent of other subgraphs.

Lastly, we formulate an f-KNAPSACK instance as described in the following algorithm for updating the DP table on a single operation (O4).

Algorithm 4.10 (MaxHS-MW)**.** Given a graph substitution $\mathcal{G} = H(G_1, \ldots, G_n)$ and partial solutions $\phi[G_1, \cdot], \ldots, \phi[G_n, \cdot]$, consider all combinations of entire subgraphs from $G_1, \ldots, G_n$ and proceed the following steps.

(Step 1) Identify subgraph types for $G_1, \ldots, G_n$. (Step 2) Formulate an f-KNAPSACK instance with capacity k and value functions f_i, based on the subgraph G_i's type as follows.

- Type I : $f_i(x) = |G_i|\,, \qquad x = |G_i|.$
- Type II : $f_i(x) = \phi[G_i, x], \quad 0 \leq x < |G_i|.$
- Type III : $f_i(x) = 0, \qquad x = |G_i|.$
- Type IV : $f_i(x) = 0, \qquad 0 \leq x < |G_i|.$

Then, update the DP table entries $\phi[\mathcal{G}, w]$ for $0 \leq w \leq k$ with the solution to f-KNAPSACK, if its value is greater than the current value.

We now prove that the runtime of this algorithm is FPT with respect to modular-width.

Lemma 4.11. *Algorithm 4.10 correctly computes $\phi[\mathcal{G}, w]$ for every $0 \leq w \leq k$ in time $\mathcal{O}(2^{\mathsf{mw}} \cdot \mathsf{mw} \cdot k^2)$.*

Proof. First, the algorithm considers all possible sets of entire substituted subgraphs $(G_1, \ldots, G_n)$. The optimal solution must belong to one of them. It remains to prove the correctness of the f-KNAPSACK formulation in step 2. From Lemma 4.3, the subgraphs of Type III and IV cannot increase the number of happy vertices, so we set

$f_i(x) = 0$. For Type I, the algorithm has no option but to include all of $V(G_i)$ in the happy set, and they are all happy.

The subgraphs of Type II are the only ones that use previous results, $\phi[G_i, \cdot]$. Since the new neighbors to G_i are required to be in the happy set, for any choice of the happy set in G_i, happy vertices remain happy, and unhappy vertices remain unhappy. Thus, we can directly use $\phi[G_i, \cdot]$; its choice does not affect other substituted subgraphs, as Type II subgraphs are independent in H. The domain of functions f_i is naturally determined by the definition of entire subgraphs.

Now, consider the running time of Algorithm 4.10. It considers 2^n possible combinations of entire subgraphs. Step 1 can be done by checking neighbors for each vertex in H, so the running time is $\mathcal{O}(|E(H)|) = \mathcal{O}(n^2)$. And step 2 takes time $\mathcal{O}(nk^2)$ from Corollary 4.5. The total running time is $\mathcal{O}(2^n(n^2 + nk^2)) = \mathcal{O}(2^{\mathsf{mw}} \cdot \mathsf{mw} \cdot k^2)$ as we assume $n \leq \mathsf{mw} < k$. □

Proof of Theorem 4.8. It is trivial to see that the base case of the DP is valid, and the correctness of inductive steps is given by Lemma 4.11. We process each node of the parse-tree once, and it has $\mathcal{O}(|V(G)|)$ nodes [12]. Thus, the overall runtime is $\mathcal{O}(2^{\mathsf{mw}} \cdot \mathsf{mw} \cdot k^2 \cdot |V(G)|)$. □

4.4.2 Parameterized by Clique-Width

We provide an algorithm for MaxHS parameterized only by clique-width (cw), which no longer requires a combined parameter with solution size k as presented in [1].

Theorem 4.12. *Given a* cw*-expression tree of a graph G with clique-width* cw*,* MaxHS *can be solved in time $\mathcal{O}(8^{\mathsf{cw}} \cdot k^2 \cdot |V(G)|)$.*

Here we assume that we are given a cw-expression tree, where each node t represents a *labeled* graph G_t. A labeled graph is a graph whose vertices are labeled by integers in $L = \{1, \ldots, \mathsf{cw}\}$. Every node must be one of the following: introduce node $i(v)$, union node $G_1 \oplus G_2$, relabel node $\rho(i, j)$, or join node $\eta(i, j)$. We write V_i for the set of vertices with label i.

Our algorithm traverses the cw-expression tree from the leaves and performs dynamic programming. For every node t, we keep track of the annotated partial

solution $\phi[t, w, X, T]$, for every integer $0 \leq w \leq k$ and sets of labels $X, T \subseteq L$. We call X the *entire labels* and T the *target labels*. $\phi[t, w, X, T]$ is defined to be the maximum number of happy vertices having target labels T for G_t with respect to a happy set $S \subseteq V(G_t)$ of size w such that V_ℓ is entire to S if and only if $\ell \in X$. The entries of the DP table are initialized with $\phi[t, w, X, T] = -\infty$. The solution to the original graph G is computed by $\max_{X \subseteq L} \phi[r, k, X, L]$, where r is the root of the cw-expression tree. Now we claim the following recursive formula for each node type.

Lemma 4.13 (Formula for introduce nodes)**.** *Suppose t is an introduce node, where a vertex v with label i is introduced. Then, the following holds.*

$$\phi[t, w, X, T] = \begin{cases} 1 & \text{if } w = 1,\ X = L \text{ and } i \in T; \\ 0 & \text{if } w = 1,\ X = L \text{ and } i \notin T; \\ 0 & \text{if } w = 0 \text{ and } X = L \setminus \{i\}; \\ -\infty & \text{otherwise.} \end{cases}$$

Proof. First, notice that all labels but i are empty and thus entire. If we include v in the happy set, then we get $w = 1$ and $X = L$ (all labels are entire). The resulting value depends on the target labels. If i is a target label, i.e., $i \in T$, then v is a happy vertex having a target label, resulting in $\phi[t, w, X, T] = 1$. Otherwise, $\phi[t, w, X, T] = 0$. If $w = 0$, then label i cannot be entire, and the only feasible solution is $\phi[t, 0, L \setminus \{i\}, T] = 0$. □

Lemma 4.14 (Formula for union nodes)**.** *Suppose t is a union node with child nodes t_1 and t_2. Then, the following holds.*

$$\phi[t, w, X, T] = \max_{0 \leq \tilde{w} \leq w} \max_{\substack{X_1, X_2 \subseteq L \\ : X_1 \cap X_2 = X}} \phi[t_1, \tilde{w}, X_1, T] + \phi[t_2, w - \tilde{w}, X_2, T]$$

Proof. At a union node, since G_{t_1} and G_{t_2} are disjoint, any maximum happy set in G_t must be the disjoint union of some maximum happy set in G_{t_1} and that in G_{t_2} for the same target labels. We consider all possible combinations of partial solutions to G_{t_1} and G_{t_2}, so the optimality is preserved. Note that a label in G_t is entire if and only if it is entire in both G_{t_1} and G_{t_2}. □

Lemma 4.15 (Formula for relabel nodes)**.** *Suppose t is a relabel node with child node t', where label i in graph $G_{t'}$ is relabeled to j. Then, the following holds.*

$$\phi[t, w, X, T] = \begin{cases} -\infty & \text{if } i \notin X; \\ \phi[t', w, X, T'] & \text{if } i \in X \text{ and } j \in X; \\ \max\limits_{Y \in \{\emptyset, \{i\}, \{j\}\}} \phi[t', w, X \setminus \{i\} \cup Y, T'] & \text{if } i \in X \text{ and } j \notin X, \end{cases}$$

where $T' = T \cup \{i\}$ *if* $j \in T$ *and* $T \setminus \{i\}$ *otherwise.*

Proof. At a relabel node $\rho(i, j)$, label i becomes empty, so it must be entire in G_t, leading to the first case. The variable T' converts the target labels in $G_{t'}$ to those in G_t. If label j is a target in G_t, then i and j must be targets in $G_{t'}$. Likewise, if label j is not a target in G_t, then neither i nor j should be targets in $G_{t'}$.

If label j is entire in G_t, then the maximum happy set must be the same as the one in $G_{t'}$ where both labels i and j are entire. If j is not entire in G_t, then we need to choose the best solution from the following: i is entire but j is not, j is entire but i is not, neither i nor j is entire. Because G_t and $G_{t'}$ have the same underlying graph, the optimal solution must be one of these. □

Figure 4.5 illustrates the third case of the formula in Lemma 4.15.

Lemma 4.16 (Formula for join nodes)**.** *Suppose t is a join node with the child node t', where labels i and j are joined. Then, the following holds.*

$$\phi[t, w, X, T] = \begin{cases} \phi[t', w, X, T] & \text{if } i \in X \text{ and } j \in X \\ \phi[t', w, X, T \setminus \{i\}] & \text{if } i \in X \text{ and } j \notin X \\ \phi[t', w, X, T \setminus \{j\}] & \text{if } i \notin X \text{ and } j \in X \\ \phi[t', w, X, T \setminus \{i, j\}] & \text{if } i \notin X \text{ and } j \notin X \end{cases}$$

Proof. At a join node $\eta(i, j)$, first observe that for any happy set, the vertices labeled other than i, j are unaffected; happy vertices remain happy. Further, if label j is not entire in G_t, then all vertices in V_i cannot be happy from Lemma 4.3. Thus, the maximum happy set in G_t is equivalent to the one in $G_{t'}$ such that label i is not a target label. The same argument applies to the other cases. □

Figure 4.6 illustrates the second case of the formula in Lemma 4.16. Lastly, we examine the running time of these computations.

Proposition 4.17. *Given a* cw*-expression tree and its node t, and partial solutions $\phi[t', \cdot, \cdot, \cdot]$ for every child node t' of t, we can compute $\phi[t, w, X, T]$ for every w, X, T in time $\mathcal{O}(8^{\mathsf{cw}} \cdot k^2)$.*

Proof. It is clear to see that for fixed t, w, X, T, the formulae for introduce, relabel, and join nodes take $\mathcal{O}(1)$. If we compute $\phi[t, \cdot, \cdot, \cdot]$ for every w, X, T, the total running time is $\mathcal{O}((2^{\mathsf{cw}})^2 \cdot k)$ since w is bounded by k and there are 2^{cw} configurations for X and T.

For the union node formula, observe that X can be determined by the choice of X_1 and X_2, so it is enough to consider all possible values for $w, T, \tilde{w}, X_1, X_2$, which results in the running time $\mathcal{O}((2^{\mathsf{cw}})^3 \cdot k^2)$, or $\mathcal{O}(8^{\mathsf{cw}} \cdot k^2)$. □

This completes the proof of Theorem 4.12, as we process each node of the cw-expression tree once, and it has $\mathcal{O}(|V(G)|)$ nodes.

4.5 Algorithms for Maximum Edge Happy Set

In addition to MAXHS, we also study its edge-variant MAXEHS. One difference from MAXHS is that when joining two subgraphs, we may increase the number of edges between those subgraphs, even if they are not entire. In other words, the number of edges between joining subgraphs depends on two variables, and quadratic programming naturally takes part in this setting. Here, we present FPT algorithms for two parameters—neighborhood diversity and cluster deletion number—to investigate the boundary between parameters that are FPT (e.g., treewidth) and W[1]-hard (e.g., clique-width) (see Figure 4.2).

4.5.1 Parameterized by Neighborhood Diversity

As shown in Figure 4.1, neighborhood diversity is a parameter specializing modular-width. To obtain a finer classification of structural parameters, we now show MAXEHS is FPT parameterized by neighborhood diversity.

Let nd be the neighborhood diversity of the given graph G. We observe that any instance (G, k) of MAXEHS can be reduced to the instance of INTEGER QUADRATIC PROGRAMMING (IQP) as follows.

Lemma 4.18. MAXEHS *can be reduced to* IQP *with* $\mathcal{O}(\mathsf{nd})$ *variables and bounded coefficients in time* $\mathcal{O}(|V(G)| + |E(G)|)$.

Proof. First, we compute the set of twins (modules) $\mathcal{M} = M_1, \ldots, M_{\mathsf{nd}}$ of G, and obtain the quotient graph H on the modules $\mathcal{M}$ in time $\mathcal{O}(|V(G)| + |E(G)|)$ [17]. Note that each

module M_i is either a clique or an independent set. Let us define a vector $q \in \mathbb{Z}^{\mathsf{nd}}$ such that $q_i = 1$ if M_i is a clique, and $q_i = 0$ if M_i is an independent set. Further, let $A \in \mathbb{Z}^{\mathsf{nd} \times \mathsf{nd}}$ be the adjacency matrix of H where $A_{ij} = 1$ if $M_i M_j \in E(H)$, and $A_{ij} = 0$ otherwise.

We then formulate an IQP instance as follows:

- Variables: $x \in \mathbb{Z}^{\mathsf{nd}}$.
- Maximize: $f(x) = x^T(A + qq^T)x - q^T x$ (equivalently, minimize $-f(x)$).
- Subject to: $\sum_i x_i = k$ and $0 \leq x_i \leq |M_i|$ for every $1 \leq i \leq \mathsf{nd}$.

This formulation has nd variables, and its coefficients are either 0 or ± 1, thus bounded. After finding the optimal vector x, pick any x_i vertices from module M_i and include them in the happy set S. We claim that S maximizes the number of happy edges.

For any happy set S, the number of happy edges is given by the sum of the number of happy edges inside each module M_i, which we call *internal edges*, and the number of edges between each module pair M_i and M_j, or *external edges*. Let $x \in \mathbb{Z}^{\mathsf{nd}}$ be a vector such that $x_i = |S \cap M_i|$ for every i. Then, the number of internal edges of module M_i is $q_i \cdot \binom{x_i}{2}$, and the number of external edges between modules M_i and M_j is $A_{ij} \cdot x_i x_j$. The number of happy edges, i.e., $|E(G[S])|$, is given by: $h(x) = \left[\sum_{i=1}^{\mathsf{nd}} q_i \cdot \binom{x_i}{2}\right] + \left[\sum_{1 \leq i < j \leq \mathsf{nd}} A_{ij} \cdot x_i x_j\right]$. One can trivially verify $f(x) = 2h(x)$.

If the IQP instance is feasible, then we can find a happy set S of size k maximizing $h(x)$, which must be the optimal solution to MaxEHS. Otherwise, $\sum_i |M_i| = |V(G)| < k$, and MaxEHS is also infeasible. □

Figure 4.7 exemplifies a quotient graph of a graph with $\mathsf{nd} = 5$, along with vector q and matrix A. The following is a direct result from Lemma 4.18 and Proposition 4.6.

Theorem 4.19. MaxEHS *can be solved in time* $f(\mathsf{nd}) \cdot |V(G)|^{\mathcal{O}(1)}$, *where* nd *is the neighborhood diversity of the input graph* G *and* f *is a computable function.*

4.5.2 Parameterized by Cluster Deletion Number

Finally, we present an FPT algorithm for MaxEHS parameterized by the cluster deletion number of the given graph.

Theorem 4.20. *Given a graph $G = (V, E)$ and its cluster deletion set X of size* cd*,* MAXEHS *can be solved in time $\mathcal{O}(2^{\mathsf{cd}} \cdot k^2 \cdot |V|)$.*

Recall that by definition, $G[V \setminus X]$ is a set of disjoint cliques. Let $C_1, \ldots, C_p$ be the clusters appeared in $G[V \setminus X]$. Our algorithm first guesses part of the happy set S', defined as $S \cap X$, and performs f-KNAPSACK with p items.

Algorithm 4.21 (MaxEHS-CD)**.** Given a graph $G = (V, E)$ and its cluster deletion set X, consider all sets of $S' \subseteq X$ such that $|S'| \leq k$ and proceed the following steps.

(Step 1) For each clique C_i, sort its vertices in non-increasing order of the number of neighbors in S'. Let $v_{i,1}, \ldots, v_{i,|C_i|}$ be the ordered vertices in C_i. (Step 2) For each $1 \leq i \leq p$, construct a function f_i as follows: $f_i(0) = 0$ and for every $1 \leq j \leq |C_i|$, $f_i(j) = f_i(j-1) + \left|N(v_{i,j}) \cap S'\right| + j - 1$. (Step 3) Formulate an f-KNAPSACK instance with capacity $k - |S'|$ and value functions f_i for every $1 \leq i \leq p$. Then, obtain the solution $\{x_i\}$ with the exact capacity $k - |S'|$ if feasible. (Step 4) Construct S as follows: Initialize with S' and for each clique C_i, pick x_i vertices in order and include them in S. That is, update $S \leftarrow S \cup \{v_{i,1}, \ldots, v_{i,x_i}\}$ for every $1 \leq i \leq p$. Finally, return S that maximizes $|E(G[S])|$.

Intuitively, we construct function f_i in a greedy manner. When we add a vertex v in clique C_i to the happy set S, it will increase the number of happy edges by the number of v's neighbors in S' and the number of the vertices in C_i that are already included in S. Therefore, it is always advantageous to pick a vertex having the most neighbors in S'. Figure 4.8 illustrates the key ideas of Algorithm 4.21. The following proposition completes the proof of Theorem 4.20.

Proposition 4.22. *Given a graph $G = (V, E)$ and its cluster deletion set X of size* cd*, Algorithm 4.21 correctly finds the maximum edge happy set in time $\mathcal{O}(2^{\mathsf{cd}} \cdot k^2 \cdot |V|)$.*

Proof. The algorithm considers all possible sets of $S \cap X$, so the optimal solution should extend one of them. It is clear to see that when the f-KNAPSACK instance is feasible, S ends up with k vertices, since the sum of the obtained solution must be $k - |S'|$. The objective of the f-KNAPSACK is equivalent to $|E(G[S])| - |E(G[S'])|$, that is, the number of happy edges extended by the vertices in $V \setminus X$. Since S' has been fixed at this point, the

optimal solution to f-KNAPSACK leads to that to MAXEHS. Lastly, the value function f_i is correct because for each clique C_i, the number of extended edges is given by $\binom{|S_i|}{2} + \sum_{v \in S_i} |N(v) \cap S'|$, where $S_i = S \cap C_i$ and $x_i = |S_i|$. This is maximized by choosing $|S_i|$ vertices that have the most neighbors in S', if we fix $|S_i|$, represented as x_i in f-KNAPSACK. This is algebraically consistent with the recursive form in step 2.

For the running time, the choice of S' adds the complexity of 2^{cd} to the entire algorithm. Having chosen S', vertex sorting (step 1) can be accomplished by checking the edges between S' and $V \setminus X$, so it takes only $\mathcal{O}(k \cdot |V|)$. The f-KNAPSACK (step 3) takes time $\mathcal{O}(pk^2) = \mathcal{O}(k^2 \cdot |V|)$ from Corollary 4.5, because there are p items and weights are bounded by k. Steps 2 and 4 do not exceed this asymptotic running time. The total runtime is $\mathcal{O}(2^{\mathsf{cd}} \cdot k^2 \cdot |V|)$. □

4.6 Conclusions and Future Work

We present four algorithms using a variety of techniques, two for MAXIMUM HAPPY SET (MAXHS) and two for MAXIMUM EDGE HAPPY SET (MAXEHS). The first shows that MAXHS is FPT with respect to the modular-width parameter, which is stronger than clique-with but generalizes several parameters such as neighborhood diversity and twin cover number. We then give an FPT dynamic-programming algorithm for MAXHS parameterized by clique-width. This improves the best known complexity result of FPT when parameterized by clique-width plus k.

For MAXEHS, we prove that it is FPT by neighborhood diversity, using a result for INTEGER QUADRATIC PROGRAMMING as a black box. Lastly, we show an FPT algorithm parameterized by cluster deletion number, the distance to a cluster graph, which then implies FPT by twin cover number. These results have answered several open questions of [1] (Figure 4.2). While it is FPT, there cannot be a polynomial kernel with respect to nd and cd, due to the lower-bounds on CLIQUE parameterized by vertex cover number, unless $\mathsf{NP} \subseteq \mathsf{co\text{-}NP/poly}$ [2].

There are multiple potential directions for future research. As highlighted in Figure 4.2, the parameterized complexity of MAXEHS with respect to modular-width is still open. Another direction would be to find the lower bounds for known algorithms.

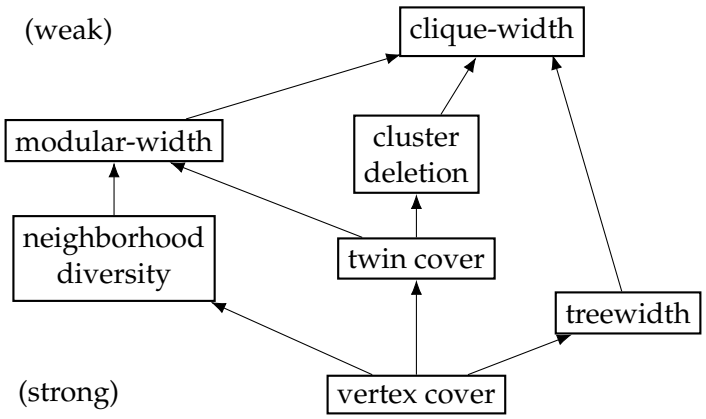


Figure 4.1: Hierarchy of relevant structural graph parameters. Arrows indicate generalizations.

Parameter	MaxHS	MaxEHS
Size k of happy set	W[1]-hard[1]	W[1]-hard[6]
Clique-width + k	FPT[1]	FPT[4]
Clique-width	FPT	W[1]-hard[4]
Modular-width	FPT	Open
Neighborhood diversity	FPT[1]	FPT
Cluster deletion number	FPT[1]	FPT
Twin cover number	FPT[1]	FPT
Treewidth	FPT[1]	FPT[3]
Vertex cover number	FPT[1]	FPT[3]

Figure 4.2: Known and established hardness results under select parameters for MaxHS and MaxEHS (as known as Densest k-Subgraph). New results from this work in red.

Figure 4.3: Given a graph above, if $k = 5$, choosing $S = \{c, d, e, f, g\}$ makes only one vertex (e) happy (left). On the other hand, $S = \{a, b, c, d, e\}$ is an optimal solution, making four vertices (a, b, c, d) happy (right).

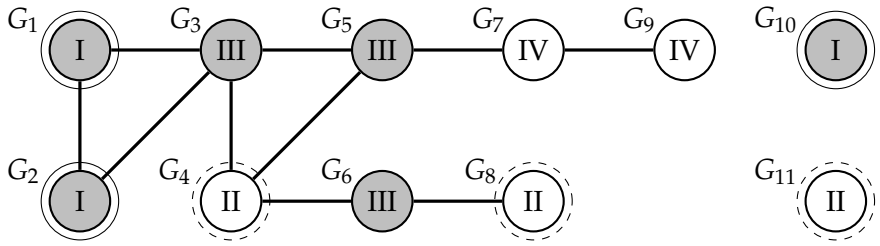


Figure 4.4: Four types of the subgraphs after applying operation (O4). Entire subgraphs (Type I and III) are shaded in gray. A subgraph becomes Type III or IV if it has a non-entire neighbor (e.g., G_3, G_9). In a Type I subgraph, all vertices are happy. Type II subgraphs may or may not admit a happy vertex. Type III and IV subgraphs cannot contain a happy vertex, as it is adjacent to a non-entire subgraph.

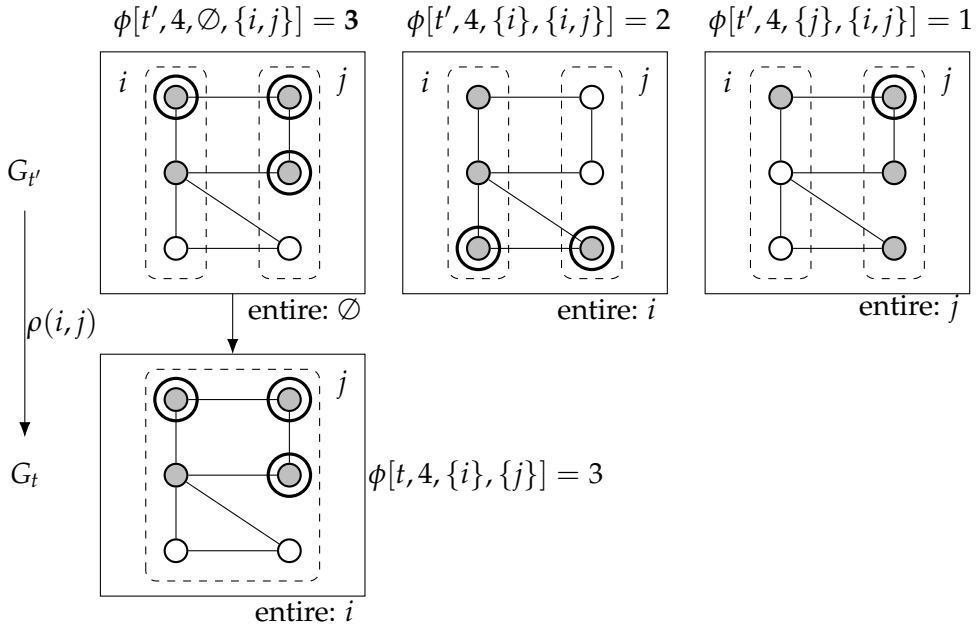


Figure 4.5: Visualization of the relabel operation $\rho(i, j)$ in the cw-expression tree of labels i, j. Figures show happy sets of size 4 (shaded in gray) maximizing the number of happy vertices (shown with double circles). After relabeling i to j, label i becomes empty and thus entire. Since label j in G_t corresponds to labels i and j in $G_{t'}$, to compute $\phi[t, 4, \{i\}, \{j\}]$, we need to look up three partial solutions $\phi[t', 4, \emptyset, T']$, $\phi[t', 4, \{i\}, T']$, and $\phi[t', 4, \{j\}, T']$, where $T' = \{i, j\}$, and keep the one with the largest value (the left one in this example).

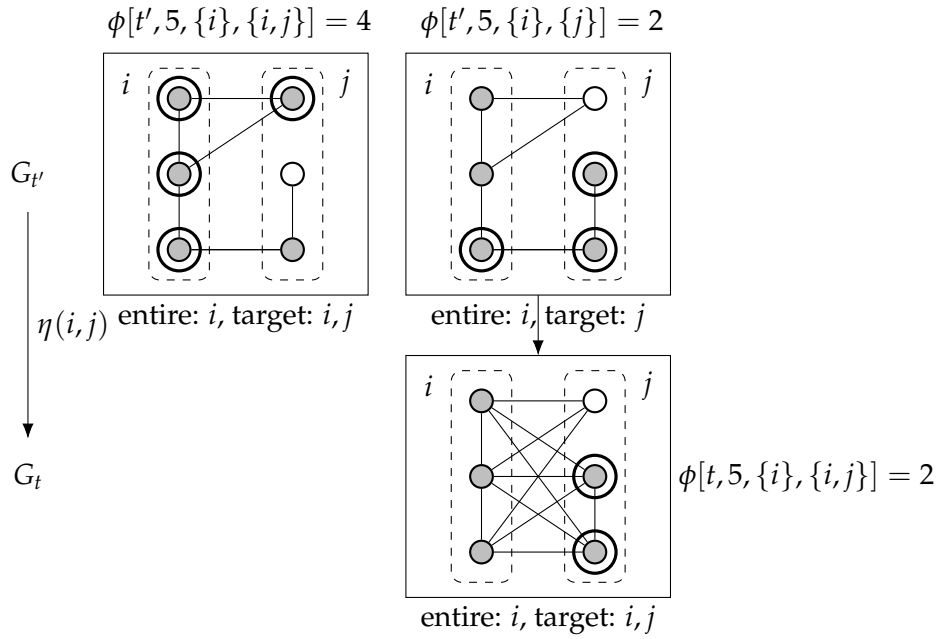


Figure 4.6: Visualization of the join operation $\eta(i, j)$ in the cw-expression tree of labels i, j. Figures show happy sets of size 5 (shaded in gray) maximizing the number of happy vertices (shown with double circles) for different target labels. We consider the case where label i is entire and j is not. The graph $G_{t'}$ admits 4 happy vertices if both i and j are target labels. However, this is no longer true after the join because label j is not entire. Instead, the optimal happy set for G_t can be found where label i is excluded from the target labels for $G_{t'}$, i.e., $\phi[t', 5, \{i\}, \{j\}]$, which admits 2 happy vertices with label j. Notice that we do not count the happy vertices with label i if it is not a target.

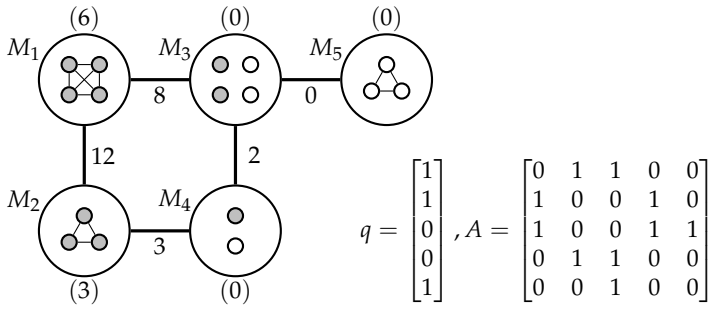


Figure 4.7: Example instance of MAXEHS with $\mathsf{nd} = 5$. The figure shows the quotient graph H of the given graph G on its modules $M_1, \ldots, M_5$. Every edge in H forms a biclique in G. The maximum edge happy set for $k = 10$ are shaded in gray. It also shows the number of internal edges for each module (e.g., (6) for M_1), and that of external edges between modules (e.g., 12 between M_1 and M_2). Vector q indicates if each module is a clique or an independent set (e.g., $q_1 = 1$ because M_1 is a clique), and A is the adjacency matrix of the quotient graph.

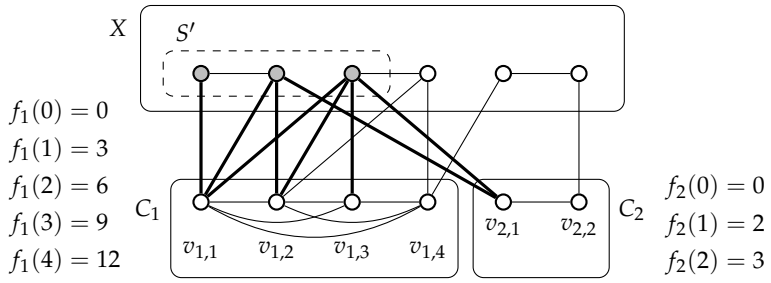


Figure 4.8: Visualization of MaxEHS-CD, given a graph $G = (V, E)$ with its cluster deletion set X and a fixed partial solution S' ($|S'| = 3$, shaded in gray). The graph after removing X, i.e., $G[V \setminus X]$, forms cliques C_1 and C_2. For each clique, vertices are sorted in decreasing order of the number of neighbors in S' (edges to S' in thicker lines). Functions f_1 and f_2 are constructed as described in the algorithm and used for f-KNAPSACK. For example, $f_1(3) = f_1(2) + 1 + (3 - 1)$ as vertex $v_{1,3}$ has one edge to S' and two edges to previously-added $v_{1,1}$ and $v_{1,2}$. If $k = 6$, then we pick $k - |S'| = 3$ vertices from C_1 and C_2. The optimal solution would be $\{v_{1,1}, v_{1,2}, v_{1,3}\}$ because $f_1(3) + f_2(0) = 9$ gives the maximum objective value in the f-KNAPSACK formulation.

REFERENCES

[1] Y. ASAHIRO, H. ETO, T. HANAKA, G. LIN, E. MIYANO, AND I. TERABARU, *Parameterized algorithms for the happy set problem*, Discrete Applied Mathematics, 304 (2021), pp. 32–44.

[2] H. L. BODLAENDER, B. M. P. JANSEN, AND S. KRATSCH, *Kernelization lower bounds by cross-composition*, SIAM Journal on Discrete Mathematics, 28 (2014), pp. 277–305.

[3] N. BOURGEOIS, A. GIANNAKOS, G. LUCARELLI, I. MILIS, AND V. T. PASCHOS, *Exact and approximation algorithms for densest k-subgraph*, in 7th International Workshop on Algorithms and Computation (WALCOM 2013), vol. 7748, 2013, pp. 114–125.

[4] H. BROERSMA, P. A. GOLOVACH, AND V. PATEL, *Tight complexity bounds for fpt subgraph problems parameterized by the clique-width*, Theoretical Computer Science, 485 (2013), pp. 69–84.

[5] M. BRUGLIERI, M. EHRGOTT, H. W. HAMACHER, AND F. MAFFIOLI, *An annotated bibliography of combinatorial optimization problems with fixed cardinality constraints*, Discrete Applied Mathematics, 154 (2006), pp. 1344–1357.

[6] L. CAI, *Parameterized Complexity of Cardinality Constrained Optimization Problems*, The Computer Journal, 51 (2007), pp. 102–121.

[7] D. G. CORNEIL AND Y. PERL, *Clustering and domination in perfect graphs*, Discrete Applied Mathematics, 9 (1984), pp. 27–39.

[8] D. EASLEY, J. KLEINBERG, ET AL., *Networks, crowds, and markets: Reasoning about a highly connected world*, Significance, 9 (2012), pp. 43–44.

[9] U. FEIGE AND M. SELTSER, *On the Densest K-Subgraph Problem*, Algorithmica, 29 (1997), p. 2001.

[10] F. V. FOMIN, P. A. GOLOVACH, D. LOKSHTANOV, AND S. SAURABH, *Algorithmic lower bounds for problems parameterized by clique-width*, in Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete algorithms, SODA '10, USA, 2010, Society for Industrial and Applied Mathematics, pp. 493–502.

[11] S. FORTUNATO, *Community detection in graphs*, Physics Reports, 486 (2010), pp. 75–174.

[12] J. GAJARSKÝ, M. LAMPIS, AND S. ORDYNIAK, *Parameterized algorithms for modular-width*, in Parameterized and Exact Computation, G. Gutin and S. Szeider, eds., Springer International Publishing, 2013, pp. 163–176.

[13] G. GALLO, P. L. HAMMER, AND B. SIMEONE, *Quadratic knapsack problems*, in Combinatorial Optimization, M. W. Padberg, ed., Mathematical Programming Studies, Springer, Berlin, Heidelberg, 1980, pp. 132–149.

[14] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability; A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., USA, 1979.

[15] T. HANAKA, *Computing Densest k-Subgraph with Structural Parameters*, 2022. arXiv:2207.09803 [cs].

[16] G. KORTSARZ AND D. PELEG, *On choosing a dense subgraph*, Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science, (1993).

[17] M. LAMPIS, *Algorithmic Meta-theorems for Restrictions of Treewidth*, Algorithmica, 64 (2012), pp. 19–37.

[18] A. LI AND P. PENG, *Community Structures in Classical Network Models*, Internet Mathematics, 7 (2011).

[19] D. LOKSHTANOV, *Parameterized integer quadratic programming: Variables and coefficients*, 2015. arXiv:1511.00310 [cs].

[20] M. MCPHERSON, L. SMITH-LOVIN, AND J. M. COOK, *Birds of a feather: Homophily in social networks*, Annual review of sociology, 27 (2001), pp. 415–444.

[21] Y. MIZUTANI AND B. D. SULLIVAN, *Parameterized Complexity of Maximum Happy Set and Densest k-Subgraph*, in 17th International Symposium on Parameterized and Exact Computation (IPEC 2022), H. Dell and J. Nederlof, eds., vol. 249 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2022, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 23:1–23:18.

[22] H. MOSER, *Exact algorithms for generalizations of vertex cover*, Institut für Informatik, Friedrich-Schiller-Universität Jena, (2005).

[23] M. TEDDER, D. CORNEIL, M. HABIB, AND C. PAUL, *Simpler linear-time modular decomposition via recursive factorizing permutations*, in Automata, Languages and Programming, Berlin, Heidelberg, 2008, Springer Berlin Heidelberg, pp. 634–645.

[24] P. ZHANG AND A. LI, *Algorithmic aspects of homophyly of networks*, Theoretical Computer Science, 593 (2015), pp. 117–131.

CHAPTER 5

TWIN-WIDTH SOLVER

The previous chapter investigated several structural parameters and gave FPT algorithms for specific problems. However, a drawback of such an approach is that real-world graphs rarely have small values of most parameters that admit FPT algorithms. The relatively new parameter twin-width can be seen as a good compromise—filling the gap between efficiency and practicality. Although its origin was the permutation width of Guillemot and Marx [8], real-world graph instances tend to have surprisingly small twin-widths. The study of twin-width has been producing fruitful results, including FPT algorithms for 3-COLORING [5] and k-INDEPENDENT SET [4], as well as polynomial-time approximation algorithms for MINIMUM DOMINATING SET [4] and other problems. Still, there is a big gap between theory and practice, and one of the main obstacles is that computing the twin-width of a given graph is notoriously hard.

In the parameterized algorithms community, the annual Parameterized Algorithms and Computational Experiments Challenge (PACE) was conceived in Fall 2015 to deepen the relationship between parameterized algorithms and practice. I participated in this challenge in 2021 through 2023, but here I want to highlight our achievements in PACE 2023, when the task was to compute the twin-width of a graph and output its certificate (called a *contraction sequence*). Working with Dursteler and Sullivan, we submitted our solver Hydra Prime and won the Exact Track. This work was also awarded the PACE Theory Award at International Symposium on Parameterized and Exact Computation (IPEC) '23 [14]. This chapter elaborates our technical contributions, as implemented in Hydra Prime.

5.1 Introduction

The goal of the 2023 PACE Challenge (`https://pacechallenge.org/2023/`) was to compute twin-width [6], a structural graph parameter which measures how close a given graph is to a *cograph* – a graph which can be reduced to a single vertex by repeatedly

merging (*contracting*) pairs of *twins* – vertices with identical open neighborhoods. More generally, twin-width measures the minimum number of "mistakes" made in such a process when the pairs being contracted are no longer twins. If u and v are being merged, we say uy becomes a *red edge* if y is a neighbor of u but not v (and analogously for edges vy). The width of a contraction sequence is then the maximum number of red edges incident to any vertex (*red degree*) at any time during the process, and the twin-width of a graph is the minimum width of all valid contraction sequences. While graphs with bounded twin-width admit many FPT algorithms, computing the parameter is NP-hard, and prior to the PACE challenge its exact computation had remained impractical even on relatively small graphs.

Most twin-width solvers naturally begin by removing twins, as all groups of twins can be collapsed without incurring any red edges, making it a safe operation. In Hydra Prime, we employ a stronger notion of this via *modular decompositions* [9], which decompose a graph into a hierarchy of maximal modules. A key property of these decompositions is that the twin-width of the original graph is exactly the maximum of the twin-width of the twin-free, prime quotient graphs (Theorem 3.1 from [15]). We thus begin by running a re-implemented linear-time modular decomposition solver based on [17], then process each prime graph separately, maintaining a global lower bound. If a prime graph is a tree, we run PrimeTreeSolver, otherwise we run a series of lower- and upper-bound algorithms (listed at the end of this section) alternatively until the bounds match, from the quickest algorithms to the slowest. Those algorithms marked with (*) in Figure 5.1 use a SAT solver as a subroutine; the implementation submitted to PACE uses the Kissat solver [3].

In this chapter, we focus on two additional contributions to solving twin-width which are used in the LocalSearch and Separate algorithms implemented in Hydra Prime: "timeline encoding" and "hydra decomposition". *Timeline encoding* is a novel data structure which enables faster computation of twin-width by storing red "sources" and "intervals" indicating the cause and window of each red edge. In the Separate upper- and lower-bound algorithms, we introduce *hydra decomposition*, an iterative refinement strategy using small vertex separators. After defining necessary notation, we briefly describe these in Sections 5.2 and 5.3, respectively. Additional details are in the appendix available on the code repository.

Notation We follow standard graph-theoretic notation (e.g., found in [7]), the original definition of twin-width [6], and terminology introduced by Schidler and Szeider [15]. Refer to [9] and [17] for the definitions of a *module*, *modular decomposition*, a *prime* graph, etc. We write $u \leftarrow v$ when vertex v is contracted into vertex u. Given a trigraph G, the *weak red potential* of $u, v \in V(G), u \neq v$ is the red degree of u after contraction $u \leftarrow v$. We further define the *unshared neighbors* of vertices u and v, denoted by $\triangle(u,v)$ as $N(u) \triangle N(v) \setminus \{u,v\}$, where $\triangle$ denotes the symmetric difference of two sets. We write $[n]$ for $\{1,\dots,n\}$.

5.2 Timeline Encoding

In this work we developed the *timeline encoding*, a data structure to compute the width of a given contraction sequence. An instance of the timeline encoding stores the following data:

- G: input graph with n vertices.
- $\phi : V(G) \to [n]$: bijection that encodes an elimination ordering (vertex v is eliminated at time $\phi(v)$ if $\phi(v) < n$).
- $p : [n-1] \to [n]$: encoding of a contraction tree. For $i < j$, $p(i) = j$ if vertex $\phi^{-1}(i)$ is merged into vertex $\phi^{-1}(j)$ (i.e., j is the *parent* of i in the contraction tree).

For internal data structures, we introduce a few terms. First, define $\triangle_{>}(j,i) := \{\phi(w) \mid w \in \triangle(\phi^{-1}(i), \phi^{-1}(j)), \phi(w) > i\}$. Then, the *red sources* at time t are a set of red edges introduced at time t, defined as $\{\{p(t),k\} \mid k \in \triangle_{>}(p(t),t)\}$. Red sources determine the *red intervals* – non-overlapping, continuous intervals where an edge is red, defined as follows: for $i < j$, red source (i,j) at time t creates an interval $[t,i)$ (red edge ij disappears at time i). If $p(i) \neq j$, then we recurse this process as if red source $\{p(i),j\}$ was created (red edge ij transfers to $\{p(i),j\}$), as illustrated in Figure 5.2.

Now we aggregate red intervals by vertices. We maintain a *multiset* of intervals for each vertex such that a red interval of an edge accounts to its both endpoints. The maximum number of the overlaps of such intervals gives the maximum red degree at a vertex over time. Finally, we obtain the width of the contraction sequence by taking the maximum of the red degrees over all vertices.

A key observation is that we can dynamically compute the number of overlaps of a multiset of intervals efficiently with a balanced binary tree (e.g., modification in time $\mathcal{O}(\log n)$, getting the maximum number of overlaps in $\mathcal{O}(1)$, etc.). For local search, we implemented methods for modifying a contraction tree and also updating a bijection ϕ.

5.3 Hydra Decomposition

We also implemented an iterative refinement strategy which we term *hydra decomposition*, based on finding a small vertex separator. A *hydra* is a structured trigraph which consists of a (possibly empty) set of *heads* and a (possibly empty) vertex set *tail*. Each head is a set of vertices containing one *top vertex* and a nonempty set of *boundary vertices*. The neighbors of the top vertex must be a subset of the boundary vertices. All red edges in the trigraph must be incident to one of the top vertices. Heads must be vertex-disjoint, but the tail may contain boundary vertices (but not a top vertex). A *compact hydra* is a hydra consisting of its tail and one extra vertex, with no heads. A head of a hydra H can additionally be viewed as a compact hydra C, where the boundary vertices of H are the tail of C. Now that we have defined the parts of a hydra, we will now show the operations performed in hydra decomposition:

1. *Separate*: partitions the vertices of a hydra into three parts S, A, B such that S separates A from B. The part S should not contain any vertices from the heads, and any tail vertices cannot be in A. Figure 5.3 shows two ways of choosing a separator S of a hydra.

2. *Contract*: takes a hydra and contracts all vertices but its tail. The output is a contraction sequence and the resulting compact hydra.

3. *Join*: combines a compact hydra C and another hydra H such that $V(C) \cap V(H)$ is the tail of C. The output is the union of C and H, where the heads and tail of H remain and C becomes an additional head.

We now present a description of UBSeparate. Given a graph H and a target width d for a contraction sequence, UBSeparate runs *contract* on the original graph without any heads or tails. The *contract* operation works as follows: If the input H is small enough, or a vertex separator of size at most d is not found, we directly search for a contraction sequence of

width at most d for all but tail vertices, which can be done by modified UBGreedy and other exact algorithms. Otherwise, we perform *separate* to obtain a partition S, A, B. We recursively call *contract* with $H[A \cup S]$ with S being the tail. Then, we have a contraction sequence s_1 and a compact hydra C. Next, we *join* C with $H[B \cup S]$ and obtain a hydra H'. Notice that the tail of C must be S. We again call *contract* with H' and get a contraction sequence s_2, resulting in a compact hydra C' with the original tail of H. Finally, C' is returned along with the concatenation of s_1 and s_2 as the result of the original *contract* operation.

A key observation is that since red edges reside only in heads and the size of separators are bounded by d, the red degree of a hydra is also upper-bounded by d, which helps construct a d-contraction sequence part by part. For $d = 1$ we use a linear-time algorithm to find a vertex separator, or a cut vertex (articulation point); for $d \geq 2$, we instead call a SAT solver.

5.4 Known Theoretical Results

We utilized the following known facts extensively in our implementation.

- If G' is an induced subgraph of a graph G, then $\mathsf{tww}(G') \leq \mathsf{tww}(G)$ [6].
- If $\overline{G}$ is the complement graph of a graph G, then $\mathsf{tww}(G) = \mathsf{tww}(\overline{G})$ [6].
- $\mathsf{tww}(G) = \max_{H \in \mathcal{P}} \mathsf{tww}(H)$, where $\mathcal{P}$ is a set of prime graphs found in the modular decomposition of G [15].
- For a graph G, $\min_{\substack{u,v \in V(G) \\ : u \neq v}} |\triangle(u, v)| \leq \mathsf{tww}(G)$ [15].
- If every component of a graph G has at most one cycle, then $\mathsf{tww}(G) \leq 2$ [2].
- For a tree T, $\mathsf{tww}(T) \leq 1$ if and only if T is a caterpillar[1] [2].

5.5 SAT Formulation

Recent PACE challenges have shown that modern SAT solvers are versatile at solving NP-hard problems; TWIN-WIDTH is not an exception. As part of our solver, we adopted

[1] A *caterpillar* is a tree containing a path P such that all other vertices are adjacent to some vertex in P.

a known SAT encoding by Schidler and Szeider [15] with a few additional clauses which we will detail in Section 5.5.3. We also used novel SAT encodings to find lower bounds (LBCore) as well as to find vertex separators (LBSeparate, UBSeparate). For our SAT encodings, we assumed that a given graph G is prime and $V(G) = [n]$, and we work with binary variables having a value either 0 (false) or 1 (true).

We use the Kissat SAT solver 3.0.0 [3] as an external library. In addition, we implemented the sequential counter [16] for encoding cardinality constraints as it performed better than the iterative totalizer [11] (typically used in MaxSAT[2]) and other algorithms.

5.5.1 Lower-Bound: LBCore

For a graph G, we define the *minimum neighborhood difference* ℓ_G as $\min_{\substack{u,v \in V(G) \\ : u \neq v}} |\triangle_G(u,v)|$ (refer to Section 5.1 for notation). The algorithm LBCore finds an induced subgraph maximizing the minimum neighborhood difference, using the fact that $\ell_{G'} \leq \mathsf{tww}(G') \leq \mathsf{tww}(G)$ for any induced subgraph G' of G.

To find a vertex set S maximizing $\ell_{G[S]}$, we guess an integer d with $d \geq 1$ and query a SAT solver if there exists a set $S \subseteq V(G)$ such that $\ell_{G[S]} \geq d$. Note that since the given graph is prime, there are no twins in G and we have $\ell_G \geq 1$; there is no need to test for $d = 0$. We observed that SAT solvers quickly find solutions for small graphs (see Figure 5.4).

5.5.1.1 Correctness. First, suppose that all clauses are satisfied. Then, the solution S has at least 4 vertices and for every vertex pair i, j in S, $|\triangle_{G[S]}(i,j)| \geq d$ since $x(i) \wedge x(j) \rightarrow \sum_{k \in \triangle(i,j)} x(k) \geq d$, which leads to $\ell_{G[S]} \geq d$. Conversely, suppose that there is a set S such that $\ell_{G[S]} \geq d \geq 1$. Since S does not contain twins, $|S| \geq 4$, satisfying the first constraint. Also, we know that for every distinct vertex pair i, j in S, $|\triangle_{G[S]}(i,j)| = |\triangle_G(i,j) \cap S| = \sum_{k \in \triangle_G(i,j)} x(k) \geq d$. This satisfies the second constraint[3].

5.5.2 Finding Vertex Separators

To perform the hydra decomposition (used in algorithms LBSeparate and UBSeparate), we need to find a vertex separator under certain constraints. We formulate a problem as

[2] A problem to maximize satisfying SAT clauses. We did not use MaxSAT for the solver.

[3] Implication denoted by $x \rightarrow y$ is equivalent to $\neg x \vee y$.

follows: given a connected graph G and its hydra structure (defined in Section 5.3) with head $H \subseteq V(G)$ and tail $T \subseteq V(G)$ vertices, and an integer k, find a vertex separator $S \subseteq V(G) \setminus H$ of size at most k, such that $G - S$ can be partitioned into two parts A, B with the following constraints. (1) A and B must be nonempty, (2) there are no edges between A and B, (3) B is a superset of $T \setminus S$, and (4) at least one vertex from T must be in B.

When $k = 1$, we can solve this problem in linear-time by testing all articulation points (or cut vertices) in G. Otherwise, we use color coding on $V(G)$ indicating the guess of a partition A, B, S with colors $\dot{A}, \dot{B}, \dot{S} : \chi : V(G) \to \{\dot{\mathsf{A}}, \dot{\mathsf{B}}, \dot{\mathsf{S}}\}$. If all vertices are properly colored, we have $A = \chi^{-1}(\dot{\mathsf{A}}), B = \chi^{-1}(\dot{\mathsf{B}}), S = \chi^{-1}(\dot{\mathsf{S}})$. See Figure 5.5 for the encoding.

5.5.2.1 Correctness. Notice that each SAT clause encodes all constraints in the problem definition. Let $A = x^{-1}(1)$, $B = y^{-1}(1)$, and $S = z^{-1}(1)$. Then, from the reachability constraints, there cannot be an edge between A and B, as if there is an edge ij such that $i \in A, j \in B$, $x(i) = y(j) = 1$, which violates $x(i) \to \neg y(j)$. Also, since $\neg x(i)$ for every $i \in T$, we have $T \cap A = \emptyset$. Then, $T \subseteq B \cup S$ and $T \setminus S \subseteq B$. For others, it is straightforward to see relations between the problem definition and SAT clauses.

5.5.3 Direct Solver

Our SAT formulation for computing exact twin-width is based on the relative encoding presented in [15], which is faster than the absolute encoding in most instances. We further add extra hints for vertex pairs having many unshared neighbors. Specifically, for a given target twin-width d and any integer $s > 0$, if vertices $u, v \in V(G)$ has $d + s$ unshared neighbors, then contraction $u \leftarrow v$ cannot appear in the first s contractions. Otherwise, the red degree of u after the contraction will exceed d. Even stronger, we can state that at least s of such unshared neighbors must be contracted before contraction $u \leftarrow v$. With these hints, some instances witnessed speed-ups in a SAT solver's running time.

See Figure 5.6 for the SAT encoding. For any distinct $1 \leq i, j \leq n$, we use shorthand notation $o^*(i, j)$ for $o(i, j)$ if $i < j$ and $\neg o(i, j)$ if $i > j$. Also, let $a^*(i, j) = a(\min\{i, j\}, \max\{i, j\})$ and similarly $r^*(i, j, k) = r(i, \min\{j, k\}, \max\{j, k\})$.

5.5.3.1 Correctness. The main proof is in [15]. Additional hints do not affect the fidelity of the encoding.

5.6 ML-Based Approach to TWIN-WIDTH

This content is not included in the IPEC '23 proceedings.

Recent trends in algorithm engineering include using machine learning (ML) to optimize the performance of heuristic solvers [10]. In collaboration with Deepak Ajwani[4], Alexander Leonhardt[5], Holger Dell[5], Johannes Meintrup[6], and Manuel Penschuck[5], we have been developing a trained model to predict the twin-width of a given graph. The team consists of PACE 2023 competitors and participants of Dagstuhl Seminar 23331 "Recent Trends in Graph Decomposition." So far, I have made two major contributions to the project.

One of my contributions was creating diverse datasets for machine learning. We have already tried several ML architectures with different graph embeddings and handcrafted features, but the biggest challenge is to prepare an unbiased dataset. First, even state-of-the-art solvers struggle to find the exact twin-width when the graph has more than 50 vertices. Therefore, if we want supervised learning for categorizing graphs by their exact twin-width, original graph instances have to be very small[7], leading to mostly single-digit twin-widths. Second, if we generate "random" graphs, there is a strong correlation between the twin-width and the density of the graph. For instance, the expected twin-width of an Erdős-Rényi $G(n, p)$ graph is known to be a function of n and p [1]. To cope with these issues, we designed several data augmentation techniques, which control n, m, and twin-width individually so that we can reduce biases in the dataset as much as possible.

Additionally, I proposed and implemented a new random graph model called the *random near-modular graph*[8], intended to have twin-widths that poly-logarithmically grow with n. A random near-modular graph $\mathsf{RNM}(n, r, d)$ takes three integral parameters n, r,

[4]University College Dublin, Ireland.

[5]Goethe University Frankfurt, Germany.

[6]Technische Hochschule Mittelhessen, Germany.

[7]For unsupervised learning, we use heuristic solvers to estimate twin-width.

[8]This idea was motivated by a question for the written portion of my qualifying examination from Prof. Jeffrey Phillips.

and d. The first step is to construct an r-ary tree T with n leaves where every level except the last is completely filled, and the last level has all nodes filled in from left to right (such a tree must exist). Each node represents a graph, and all leaves are singletons.

We process this tree in the bottom-up manner. At a node v in T with children $G_1, \ldots, G_\ell$, we create a graph G_v as follows:

1. Create the disjoint union of all graphs $G_1, \ldots, G_\ell$.
2. Randomly add edges between $u \in G_i$ and $v \in G_j$ such that $i \neq j$ while maintaining the property $|\{uv \mid u \in G_i, v \in V(G_v) \setminus G_i\}| \leq d$ for every $1 \leq i \leq \ell$, or equivalently, the size of the cut $(V(G_i), V(G_v \setminus G_i))$ in G_v is at most d.

Intuitively, we want to create a graph G_v such that when we contract all vertices in G_i for every i into a single vertex while keeping multiedges (i.e., $G_v/(G_1, G_2, \ldots, G_\ell)$ with the quotient notation), we obtain a d-regular multigraph. In the end, output the graph at the root of T. Step 2 can be done by an algorithm similar to one for random regular graphs. Initially, all children G_i are unmarked. Pick distinct unmarked i, j, and add an edge between random $u \in V(G_i)$ and $v \in V(G_j)$. If the size of the cut $(V(G_i), V(G_v - G_i))$ in G_v reaches d, we mark G_i (and similarly G_j). Continue until we cannot add any edges.

Proposition 5.1. *A random near-modular graph* $\mathsf{RNM}(n, r, d)$ *has n vertices and twin-width in* $\mathcal{O}(r + d \log n)$.

Proof. It is straightforward to see that the output graph G has n vertices. Now I claim the twin-width of G is in $\mathcal{O}(r + d \log n)$. Consider a contraction sequence that contracts from the bottom of T, that is, we always work on a particular leaf node G_i and contract all vertices in G_i except one. We remove G_i if it becomes a single vertex. At any point of contractions, let G_t be the remaining graph. Then, the red degree of a vertex $v \in G_i$ is the sum of the red degree of v in G_i (*internal red degree*) and the number of edges between v and $G_t - V(G_i)$ (*external red degree*). The internal red degree is at most r, and the external red degree is at most dh, where h is the height of tree T, which is $\mathcal{O}(\log n)$. Hence, the maximum red degree is $\mathcal{O}(r + dh) = \mathcal{O}(r + d \log n)$. □

We observe that the $\mathsf{RNM}(n, r, d)$ model generates a graph whose twin-width grows

poly-logarithmically[9] when $r = \log^2 n$, $d = \log n$, which is verified by our computational experiments[10]. I believe this model captures a structure of social networks, where each individual belongs to a hierarchy of groups (e.g., industry, country, institution, etc.), and internal interactions are much more frequent than external ones.

Another contribution was to update our twin-width solver Hydra Prime [13] to support trigraphs (graphs with red edges) as input. To integrate an ML model into a heuristic solver, it is necessary to predict the twin-width in the middle of a contraction sequence. For example, during a search of contraction sequences, a solver considers several important branches, predicts the twin-width for each branch, and takes the best one. Source code (Hydra Prime X) is available at [12].

[9] $\mathcal{O}(\log^2 n + \log n \cdot \log n) = \mathcal{O}(\log^2 n)$.

[10] Code & experimental results: `https://github.com/mogproject/graph-generators/wiki`. Explanation video: `https://www.youtube.com/watch?v=yNvM76KLcWw`.

Exact algorithms

PrimeTreeSolver	*Linear-time exact solver for trees without twins.*
BranchSolver	*Brute-force solver equipped with caching mechanism and reduction rules.*
DirectSolver (*)	*SAT-based solver implementing the relative encoding presented in [15].*

Lower-bound algorithms

LBGreedy	*Greedily removes a vertex u from the graph G such that $\lvert\triangle(u,v)\rvert$ is minimized for some v. Reports the maximum value of* $\min_{u,v\in V(G),u\neq v} \lvert\triangle_G(u,v)\rvert$.
LBCore (*)	*SAT-based algorithm to find* $\max_{S\subseteq V(G)} \min_{u,v\in S,u\neq v} \lvert\triangle_{G[S]}(u,v)\rvert$.
LBSample	*Sampling-based algorithm. Finds a connected induced subgraph G' of G by random walk and computes the exact or lower-bound twin-width of G'.*
LBSeparate (*)	*Similar to* LBSample*, but uses the hydra decomposition to find an induced subgraph to check for the lower-bound.*

Upper-bound algorithms

UBGreedy	*Iteratively contract a vertex pair minimizing the weak red potential.*
UBLocalSearch	*Using the timeline encoding, we make small changes to the elimination ordering and the contraction tree to see if there is a better solution.*
UBSeparate (*)	*Iterative refinement algorithm using the hydra decomposition.*

Figure 5.1: Algorithm list

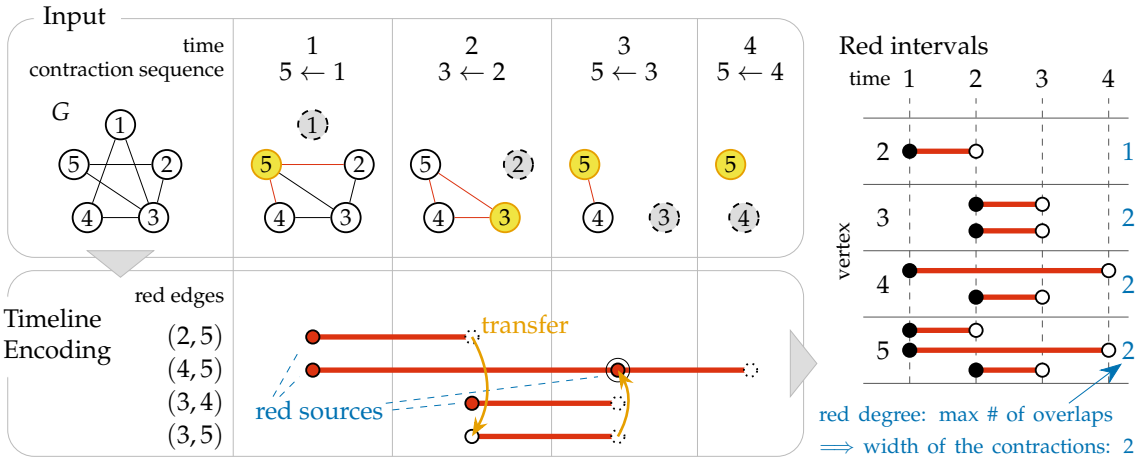


Figure 5.2: An illustration of the timeline encoding given a graph and its contraction sequence. Vertex labels show the elimination ordering. For each time i with contraction $j \leftarrow i$ $(i < j)$, we create red sources $\{k, j\}$ for every $k \in \triangle_{>}(j, i)$, which determines red intervals $[i, \min\{k, j\})$ that will then disappear or transfer at time $\min\{k, j\}$. The red degree corresponds to the number of overlaps of red intervals aggregated by vertices, and its maximum value is the width of the contraction sequence.

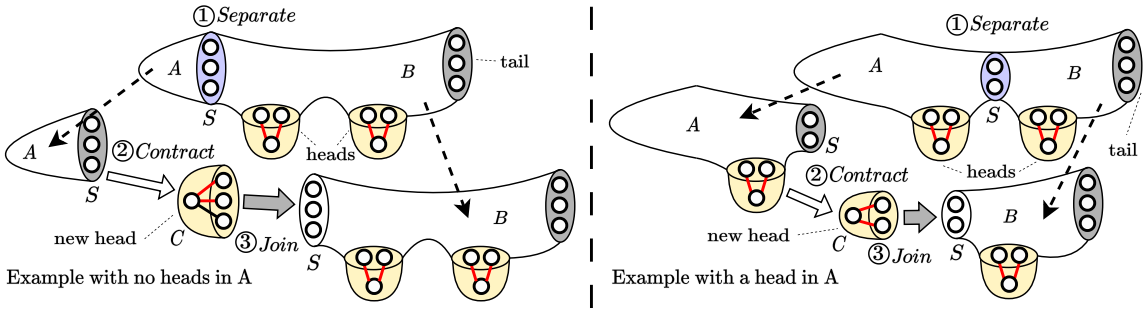


Figure 5.3: Structure of the hydra and two examples of performing a round of hydra decomposition.

Variables

$x(i)$ for $1 \leq i \leq n$ — *$x(i)$ is true iff vertex i is in a solution S.*

Semantics of $x(i)$

1 $\sum_{i=1}^{n} x(i) \geq 4$

Semantics of $\triangle(i, j)$

2 $x(i) \wedge x(j) \rightarrow \sum_{k \in \triangle(i,j)} x(k) \geq d$ for every $1 \leq i < j \leq n$

Figure 5.4: SAT encoding for LBCore.

Variables

$x(i)$ for $i \in V(G)$ — *$x(i) = 1$ if and only if vertex i has color* Ȧ.

$y(i)$ for $i \in V(G)$ — *$y(i) = 1$ if and only if vertex i has color* Ḃ.

$z(i)$ for $i \in V(G)$ — *$z(i) = 1$ if and only if vertex i has color* Ṡ.

Every vertex has a color

1 $x(i) + y(i) + z(i) = 1$ for every $i \in V(G)$

Semantics of the heads and the tail

2a $\neg z(i)$ for every $i \in H$ — *Head vertex cannot be in solution.*

2b $\neg x(i)$ for every $i \in T$ — *Tail vertex cannot be in part A.*

2c $\bigvee_{i \in T} y(i)$ — *At least one tail vertex is not in solution.*

Encoding reachability

3a $x(i) \to \neg y(j)$ for every edge $ij \in E(G)$

3b $y(i) \to \neg x(j)$ for every edge $ij \in E(G)$

Parts A and B must be nonempty

4a $\sum_{i \in V(G)} x(i) \geq 1$

4b $\sum_{i \in V(G)} y(i) \geq 1$

Constraints on solution size

5 $\sum_{i \in V(G)} z(i) \leq k$

Figure 5.5: SAT encoding for hydra decomposition.

Variables

$o(i,j)$	for $1 \leq i < j \leq n$	*encodes an elimination ordering (relative encoding); $o(i,j) = 1$ iff vertex i is earlier than vertex j in the elimination ordering.*
$p(i,j)$	for $1 \leq i < j \leq n$	*encodes a contraction tree; $p(i,j) = 1$ if vertex i is merged into vertex j.*
$r(i,j,k)$	for $1 \leq i,j \leq n$ and $j < k \leq n$	*encodes red edges; $r(i,j,k) = 1$ if after eliminating vertex i, there is a red edge jk.*
$a(i,j)$	for $1 \leq i < j \leq n$	*auxiliary variable for computing transferred red edges; $a(i,j) = 1$ if a red edge ij is present at any time.*

Semantics of o

1 $o^*(i,j) \wedge o^*(j,k) \rightarrow o^*(i,k)$ — *transitivity*
for every mutually distinct $1 \leq i,j,k \leq n$

Semantics of p

2a $\sum_{j=i+1}^{n} p(i,j) = 1$ for every $1 \leq i < n$ — *all but the root vertices must have one parent*

2b $p(i,j) \rightarrow o(i,j)$ for every $1 \leq i < j \leq n$ — *parent must be present*

Semantics of r

3a $p(i,j) \wedge o^*(i,k) \rightarrow r^*(i,j,k)$ — *encodes red edges*
for every $1 \leq i < j \leq n$ and $k \in \triangle(i,j)$

3b $p(i,j) \rightarrow \sum_{1 \leq k \leq n\colon i,j \neq k} o^*(k,i) \geq s$ — *additional hints*
for every $1 \leq i < j \leq n$ and $1 \leq s \leq d$

3c $p(i,j) \wedge o^*(i,k) \wedge a^*(i,k) \rightarrow r^*(i,j,k)$ — *transfer red edges*
for every mutually distinct $1 \leq i,k \leq n$ and $i < j \leq n$

3d $o^*(i,j) \wedge o^*(j,k) \wedge o^*(j,m) \wedge r^*(i,k,m) \rightarrow r^*(j,k,m)$ — *maintain red edges*
for every mutually distinct $1 \leq i,j \leq n$ and $1 \leq k < m \leq n$

Semantics of a

4 $o^*(k,i) \wedge o^*(k,j) \wedge r^*(k,i,j) \rightarrow a^*(i,j)$ for every mutually distinct $1 \leq i,j,k \leq n$

Constraints on the solution size

5 $\sum_{1 \leq y \leq n\colon i,x \neq y} r(i,x,y) \leq d$ for every distinct $1 \leq i,x \leq n$

Figure 5.6: SAT encoding for DirectSolver.

REFERENCES

[1] J. AHN, D. CHAKRABORTI, K. HENDREY, D. KIM, AND S.-I. OUM, *Twin-width of random graphs*, 2024. arXiv:2212.07880 [cs, math].

[2] J. AHN, K. HENDREY, D. KIM, AND S.-I. OUM, *Bounds for the Twin-width of Graphs*, SIAM Journal on Discrete Mathematics, 36 (2022), pp. 2352–2366.

[3] A. BIERE AND M. FLEURY, *Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022*, in Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions, vol. B-2022-1 of Department of Computer Science Series of Publications B, 2022, pp. 10–11.

[4] E. BONNET, C. GENIET, E. J. KIM, S. THOMASSÉ, AND R. WATRIGANT, *Twin-width III: Max Independent Set, Min Dominating Set, and Coloring*, in 48th International Colloquium on Automata, Languages, and Programming (ICALP 2021), vol. 198 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2021, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 35:1–35:20.

[5] E. BONNET, E. J. KIM, A. REINALD, AND S. THOMASSÉ, *Twin-width VI: the lens of contraction sequences*, in Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Proceedings, Society for Industrial and Applied Mathematics, 2022, pp. 1036–1056.

[6] E. BONNET, E. J. KIM, S. THOMASSÉ, AND R. WATRIGANT, *Twin-width I: tractable FO model checking*, in 2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS), 2020, pp. 601–612.

[7] R. DIESTEL, *Graph Theory*, vol. 173 of Graduate Texts in Mathematics, Springer-Verlag, Berlin, third ed., 2005.

[8] S. GUILLEMOT AND D. MARX, *Finding small patterns in permutations in linear time*, in Proceedings of the 2014 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Proceedings, Society for Industrial and Applied Mathematics, 2013, pp. 82–101.

[9] M. HABIB AND C. PAUL, *A survey of the algorithmic aspects of modular decomposition*, Computer Science Review, 4 (2010), pp. 41–59.

[10] J. LAURI, S. DUTTA, M. GRASSIA, AND D. AJWANI, *Learning fine-grained search space pruning and heuristics for combinatorial optimization*, 2020. arXiv:2001.01230 [cs].

[11] R. MARTINS, S. JOSHI, V. MANQUINHO, AND I. LYNCE, *Incremental Cardinality Constraints for MaxSAT*, in Principles and Practice of Constraint Programming (CP14), B. O'Sullivan, ed., Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 531–548.

[12] Y. MIZUTANI, *Hydra Prime X*. `https://github.com/mogproject/hydraprimex`.

[13] Y. MIZUTANI, D. DURSTELER, AND B. D. SULLIVAN, *Hydra prime*. `https://github.com/TheoryInPractice/hydraprime`.

[14] ——, *PACE Solver Description: Hydra Prime*, in DROPS-IDN/v2/document/10.4230/LIPIcs.IPEC.2023.36, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

[15] A. SCHIDLER AND S. SZEIDER, *A SAT Approach to Twin-Width*, in 2022 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX), Proceedings, Society for Industrial and Applied Mathematics, 2022, pp. 67–77.

[16] C. SINZ, *Towards an optimal CNF encoding of Boolean cardinality constraints*, in Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming, CP'05, Berlin, Heidelberg, 2005, Springer-Verlag, pp. 827–831.

[17] M. TEDDER, D. CORNEIL, M. HABIB, AND C. PAUL, *Simpler linear-time modular decomposition via recursive factorizing permutations*, in Automata, Languages and Programming, Berlin, Heidelberg, 2008, Springer Berlin Heidelberg, pp. 634–645.

CHAPTER 6

FPT ALGORITHMS FOR INSPECTION PLANNING

Similarly to the PACE Challenge's motto, one of my ultimate goals is to bring the theory of parameterized algorithms into practice. In Chapters 6 and 7, we show the impact of FPT algorithms on real-world applications in robotics. *Autonomous robotic inspection*, where a robot moves through its environment and inspects points of interest, has applications in industrial settings, structural health monitoring, and medicine. Planning the paths for a robot to safely and efficiently perform such an inspection is an extremely difficult algorithmic challenge. In this work we consider an abstraction of the inspection planning problem, which we term GRAPH INSPECTION. We give two exact algorithms for this problem (using dynamic programming and integer linear programming), analyze the performance of these methods, and present multiple approaches to achieve scalability. We demonstrate significant improvement both in path weight and inspection coverage over a state-of-the-art approach on two robotics tasks in simulation, a bridge inspection task by a UAV and a surgical inspection task using a medical robot.

In collaboration with Daniel Coimbra Salomao[1], Alex Crane[1], Matthias Bentert[2], Pål Grønås Drange[2], Felix Reidl[3], Alan Kuntz[1], and Blair D. Sullivan[1], we published our paper "Leveraging Fixed-Parameter Tractability for Robot Inspection Planning" at the 16th International Workshop on the Algorithmic Foundations of Robotics (WAFR 2024) [25]. As the lead author, I engaged in each aspect of the project. I engineered DP algorithms and devised ILP formulations. After analyzing the state-of-the-art solver (IRIS-CLI), I implemented most of our algorithms (with extensive unit tests) and conducted computational

[1]University of Utah, USA.

[2]University of Bergen, Norway.

[3]Birkbeck, University of London, UK.

experiments using our university's high-performance computing resources. I wrote the majority of the manuscript, including visualization of the experimental results, and presented our work at the WAFR conference in October 2024.

6.1 Introduction

Inspection planning, where a robot is tasked with planning a path through its environment to sense a set of points of interest (POIs) has broad potential applications. These include the inspection of surfaces to identify defects in industrial settings such as car surfaces [3], urban structures [5], and marine vessels [12], as well as in medical applications to enable the mapping of subsurface anatomy [6, 7] or disease diagnosis.

Consider the demonstrative medical example of diagnosing the cause of pleural effusion, a medical condition in which a patient's pleural space—the area between the lung and the chest wall—fills with fluid, collapsing the patient's lung [23, 18, 26]. Pleural effusion is a symptom, albeit a serious one, of one of over fifty underlying causes, and the treatment plan varies significantly, depending heavily on which of the underlying conditions has caused the effusion. To diagnose the underlying cause, physicians will insert an endoscope into the pleural space and attempt to inspect areas of the patient's lung and chest wall. Automated medical robots have been proposed as a potential assistive technology with great promise to ease the burden of this difficult diagnostic procedure. However, planning the motions to inspect the inside of a patient's body with a medical robot, or indeed *any environment with any robot* is an extremely challenging problem.

Planning a motion for a robot to move from a single configuration to another configuration is, under reasonable assumptions, known to be PSPACE-hard [17]. Inspection planning extends this typical motion planning problem by requiring the traversal of multiple configurations. The planned route may need to include complexities such as tracing back to where the robot has already been and/or traversing circuitous routes. Consider Figure 6.1, where examples are given of inspections that necessitate circuitous paths or backtracking during inspection. While the specific examples given are intuitive in the robots' workspaces, cycles and backtracking may be required in the c-space graph in ways that don't manifest intuitively in the workspace as well.

Further, it is almost certainly not sufficient to only consider the ability to inspect the

POIs, but one must also consider the *cost* of the path taken to inspect them. This is because while inspecting POIs may be an important objective, it is not the only objective in real robotics considerations; unmanned aerial vehicles (UAVs) must operate within their battery capabilities, and medical robots must consider the time a patient is subjected to a given procedure.

The state-of-the-art in inspection planning, presented by Fu *et al.* [15] and named IRIS-CLI, casts this problem as an iterative process with two phases. In the first phase, a rapidly-exploring random graph (RRG) [20] is constructed. In this graph, each vertex represents a possible configuration of the robot, edges indicate the ability to transition between configuration states, and edge weights indicate the cost of these transitions. Additionally, every vertex is labeled with the set of POIs which may be inspected by the robot when in the associated configuration state. In the second phase, a walk is computed in this graph with the dual objectives of (a) inspecting all POIs and (b) minimizing the total weight (cost) of the walk. By repeating both phases iteratively, Fu *et al.* are able to guarantee asymptotic optimality[4] of the resulting inspection plan.

In this work we focus on improving the second phase. We formulate this phase as an algorithmic problem on edge-weighted and vertex-multicolored graphs, which we call GRAPH INSPECTION (formally defined in Section 6.2). GRAPH INSPECTION is a generalization of the well-studied TRAVELING SALESPERSON (TSP) problem[5] [2]. As such, it is deeply related to the rich literature on "color collecting" problems studied by the graph algorithms community.

GRAPH INSPECTION is closely related to the GENERALIZED TRAVELING SALESPERSON PROBLEM (also known as GROUP TSP), in which the goal is to find a "Hamiltonian cycle visiting a collection of vertices with the property that exactly one vertex from each [color] is visited" [27]. If each vertex can belong to several color classes, the instance can be transformed into an instance of GTSP [22, 11]. However, in the GRAPH INSPECTION problem, we do not demand that a vertex cannot be visited several times; indeed, we expect that to be the case for many real-world cases. Rice and Tsotras [28] gave an exact

[4]See [15] for specific definition of asymptotic optimality in their case.

[5]Given an edge-weighted graph and a start-vertex s, compute a minimum-weight closed walk from s visiting every vertex exactly once.

algorithm for GTSP in $\mathcal{O}^*(2^k)$ time[6], and although being inapproximable to a logarithmic factor [30], they gave an $O(r)$-approximation in running time $\mathcal{O}^*(2^{k/r})$ [29].

Two other related problems are the T-CYCLE problem, and the MAXIMUM COLORED (s,t)-PATH problem[7]. Björklund, Husfeldt, and Taslaman provided an $\mathcal{O}^*(2^{|T|})$ randomized algorithm for the T-CYCLE problem, in which we are asked to find a (simple) cycle that visits all vertices in T [4]. In the MAXIMUM COLORED (s,t)-PATH problem, we are given a vertex-colored graph G, two vertices s and t, and an integer k, and we are asked if there exists an (s,t)-path that collects at least k colors, and if so, return one with minimum weight. Fomin *et al.* [14] gave a randomized algorithm running in time $\mathcal{O}^*(2^k)$ for this problem. Again, in both of these problems a crucial restriction is the search for *simple* paths or cycles.

Though GRAPH INSPECTION is distinct from the problems mentioned above, we leverage techniques from this literature to propose two algorithms which can solve GRAPH INSPECTION optimally[8]. First, in Section 6.3.1 we show that while GRAPH INSPECTION is NP-hard (as a TSP generalization), a dynamic programming approach can solve our problem in $2^{|\mathcal{C}|} \cdot \text{poly}(n)$ time and memory, where $|\mathcal{C}|$ is the number of POIs. Additionally, we draw on techniques used in the study of TRAVELING SALESPERSON [9] to provide a novel integer linear programming (ILP) formulation, which we describe in Section 6.3.2.

To deal with the computational intractability of GRAPH INSPECTION, Fu *et al.* took the approach of relaxing the problem to *near optimality*, enabling them to leverage heuristics in the graph search to achieve reasonable computational speed when solving the problem. We take two approaches. Using the ILP, we show that on several practical instances drawn from [15], GRAPH INSPECTION can be solved almost exactly in reasonable runtime. However, we note the ILP will not directly scale to very large graphs. For the dynamic programming routine, our approach is more nuanced: First, we note that as the running time and memory consumption of this algorithm is exponential only in the number of

[6]The $\mathcal{O}^*$ notation hides polynomial factors.

[7]Also studied under the names TROPICAL PATH [8] and MAXIMUM LABELED PATH [10].

[8]Note that in this case, and subsequently in the chapter unless otherwise indicated, 'optimal' refers to an optimal walk on the given graph and is distinct from the asymptotic optimality guarantees provided in [15].

POIs, while remaining polynomial in the size of the graph, it can optimally solve GRAPH INSPECTION when only a few POIs are present. This situation naturally arises in some application areas, particularly in medicine when the most relevant anatomical POIs may be few and known in advance. When the number of POIs is large, we adapt the dynamic programming algorithm into a heuristic by selecting several small subsets of POIs in a principled manner (see Section 6.4.1), running the dynamic program independently for each small subset, and then "merging" the resulting walks (see Section 6.4.2). Though our implementation is heuristic, it is rooted in theory: it is possible to combine dynamic programming with a "partition and merge" strategy such that, given enough runtime, the resulting walk is optimal (see Section 6.6).

We demonstrate the practical efficacy of our algorithms on GRAPH INSPECTION instances drawn from two scenarios which were used to evaluate the prior state-of-the-art planner IRIS-CLI [15]. The first is planning inspection for a bridge using a UAV (the "drone" scenario), and the second is planning inspection of the inside of a patient's pleural cavity using a continuum medical robot (the "crisp" scenario). We implemented our algorithms, DP-IPA (Dynamic Programming) and ILP-IPA (ILP), where IPA stands for Inspection Planning Algorithm. We show (see Section 6.5.1 and Figure 6.2) that on GRAPH INSPECTION instances of sizes similar to those used by [15], ILP-IPA produces walks with lower weight and higher coverage than those produced by IRIS-CLI. Indeed, ILP-IPA can produce walks with perfect coverage, even on much larger instances. However, for these larger instances DP-IPA provides a compelling alternative, producing walks with much lower weight, with some sacrifice in coverage.

In summary, this work takes steps toward the application of GRAPH INSPECTION as a problem formalization for inspection planning, and importantly provides (i) multiple novel algorithms with quality guarantees, (ii) an extensive discussion of methods used to implement these ideas in practice, and (iii) reusable software which outperforms the state-of-the-art on two relevant scenarios from the literature.

6.2 Preliminaries

Refer to Section 1.3.1 for graph-theoretic definitions and notation. We also refer to Section 1.3.2 for the general notion of parameterized complexity. We use $\chi(v)$ to denote

the *colors* (or *labels*) of a vertex (which, with nuance described below, correspond to POIs in the robot's workspace), and we write $\chi(S)$ to denote $\bigcup_{v \in S} \chi(v)$. We next define the main problem we investigate in this chapter.

GRAPH INSPECTION

Input: An undirected graph $G = (V, E)$, a set $\mathcal{C}$ of colors, an edge-weight function $w \colon E \to \mathbb{R}_{\geq 0}$, a coloring function $\chi \colon V \to 2^{\mathcal{C}}$, a start vertex $s \in V$, and an integer t.

Problem: Find a closed walk $P = (v_0, v_1, \ldots, v_p)$ in G with $v_0 = v_p = s$ and $|\bigcup_{i=1}^{p} \chi(v_i)| \geq t$ minimizing $\sum_{i=1}^{p} w(v_{i-1}v_i)$.

Note that t is the minimum number of colors to collect. For the sake of simplicity, we may assume that G is connected, $t \leq |C|$, and $\chi(s) = \emptyset$.

6.3 Graph Search

In this section, we present two algorithms for solving GRAPH INSPECTION, along with strategies for finding upper and lower bounds on the optimal solution.

6.3.1 Dynamic Programming Algorithm

We begin by establishing that GRAPH INSPECTION is fixed-parameter tractable with respect to the number of colors by giving a dynamic programming algorithm we refer to as DP-IPA.

Theorem 6.1. GRAPH INSPECTION *can be solved in* $\mathcal{O}((2^{|\mathcal{C}|}(n + |\mathcal{C}|) + m + n \log n)n)$ *time, where* $n = |V|$, $m = |E|$, *and* $\mathcal{C}$ *is the set of colors.*

Proof. Recall that we may assume $\chi(s) = \emptyset$. Otherwise, we can collect all colors in $\chi(s)$ for free; removing the colors $\chi(s)$ from the coloring function and decreasing t by $|\chi(s)|$ gives an equivalent instance. We solve GRAPH INSPECTION using dynamic programming. First, we compute the all-pairs shortest paths of G in $\mathcal{O}(nm + n^2 \log n)$ by n calls of Dijkstra's algorithm ($\mathcal{O}(m + n \log n)$ time) using a Fibonacci heap. Note that the new distance function w' is complete and metric. Hence, we may assume that an optimal solution collects at least one new color in each step (excluding the last step where it returns to s). Hence, we store in a table $T[v, S]$ with $v \in V \setminus \{s\}$ and $S \subseteq \mathcal{C}$, where S contains at least one color

in $\chi(v)$, the length of a shortest walk that starts in s, ends in v, and collects (at least) all colors in S. We fill T by increasing size of S by the recursive relation:

$$T[v,S] = \begin{cases} \infty & \text{if } S \cap \chi(v) = \emptyset \\ \min_{u \in V} T[u, S \setminus \chi(v)] + w'(uv) & \text{otherwise.} \end{cases}$$

Therein, we assume that $T[s,S] = 0$ if $S = \emptyset$ and $T[s,S] = \infty$, otherwise. We will next prove that the table is filled correctly. We do so via induction on the size of S. To this end, assume that T was computed correctly for all entries where the respective set S has size at most i. Now consider some entry $T[v,S]$ where $S \cap \chi(v) \neq \emptyset$ and $|S| = i+1$. Let ℓ' be the value computed by our dynamic program and let opt be the length of a shortest walk that starts in s, ends in v, and collects all colors in S. It remains to show that $\ell' = \text{opt}$ and to analyze the running time. We first show that $\ell' \leq \text{opt}$. To this end, let $W = (s, v_1, \ldots, v_p = v)$ be a walk of length opt that collects all colors in S. If $p = 1$, then $S \subseteq \chi(v)$ and $\ell' \leq T[s,\emptyset] + w'(sv) = w'(sv)$. Moreover, the shortest path from s to v has length $w'(sv)$ and hence $\ell' \leq w'(sv) \leq \text{opt}$. If $p > 1$, then $W' = (s, v_1, \ldots, v_{p-1})$ is a walk from s to v_{p-1} that collects all colors in $S' = S \setminus \chi(v)$. Hence, by construction $T[v_{p-1}, S'] \leq \text{opt} - w'(v_{p-1}v)$ and hence $\ell' = T[v,S] \leq T[v_{p-1}, S'] + w'(v_{p-1}v) \leq \text{opt}$.

We next show that $\ell' \geq \text{opt}$. To this end, note that whenever $T[v,S]$ is updated, then there is some vertex u such that $T[v,S] = T[u, S \setminus \chi(v)] + w'(uv)$ (where possibly $u = s$ and $S \setminus \chi(v) = \emptyset$). By induction hypothesis, there is a walk from s to u that collects all colors in $S \setminus \chi(v)$ of length $T[u, S \setminus \chi(v)]$. If we add vertex v to the end of this walk, we get a walk of length ℓ' that starts in s, ends in v, and collects all colors in S. Thus, $\text{opt} \leq \ell'$.

After the table is completely filled for all color sets S with $|S| \leq t$, then we just need to check whether there exists a vertex $v \in V \setminus \{s\}$ and a set S with $|S| = t$ such that $T[v,S] + w'(vs) \leq \ell$. Note that the number of table entries is $2^{|\mathcal{C}|}n$, computing one table entry takes $\mathcal{O}(n + |\mathcal{C}|)$ time, and the final check in the end takes $\mathcal{O}(2^{|\mathcal{C}|}n)$ time. Thus, the overall running time of our algorithm is in $\mathcal{O}(nm + n^2 \log(n) + 2^{|\mathcal{C}|}(n + |\mathcal{C}|)n) = \mathcal{O}((2^{|\mathcal{C}|}(n + |\mathcal{C}|) + m + n \log n)n)$. □

6.3.2 Integer Linear Programming Algorithm

In this section, we present ILP-IPA, an Integer Linear Programming (ILP) formulation of the problem, inspired by the flow-based technique for TSP [9]. As a (trivially checkable)

precondition, we require that a solution walk includes at least two vertices. We observe that there is a simpler formulation if the input is a complete metric graph; however, in practice, creating the completion and applying this approach significantly degrades performance because of the runtime's dependence on the number of edges.

Our ILP formulation for GRAPH INSPECTION can be found in Figure 6.3. Intuitively, the *flow amount* at a vertex encodes the number of occurrences of the vertex in a walk. Constraints (1a) and (1b) implement flow conditions, and (2a) and (2b) ensure that the flow originates at vertex s. An edge included in a solution and not touching s emits 2 *charges*, and the charges are distributed among the edge's endpoints. If every solution edge is part of a walk from s, then a charge consumption at each vertex can be slightly less than 2 per incoming flow. There are $\mathcal{O}(|\mathcal{C}| + m)$ constraints if $t = |\mathcal{C}|$, and $\mathcal{O}(|\mathcal{C}|m)$ constraints if $t < |\mathcal{C}|$.

6.3.2.1 Correctness. Before showing the correctness of the ILP formulation, we characterize solution walks for GRAPH INSPECTION. In the following, we view a solution walk as a sequence of directed edges. For a walk $P = v_0 v_1 \dots v_\ell$, we write $|P|$ for the number of edges in P, i.e., $|P| = \ell$, and $E(P)$ for the set of *directed* edges in the walk, i.e., $E(P) = \{v_{i-1}v_i \mid 1 \leq i \leq \ell\}$. We write $w(P)$ for the length of the walk, that is, $w(P) := \sum_{uv \in E(P)} w(u, v)$. We now prove a simple lemma.

Lemma 6.2. *For any feasible instance of* GRAPH INSPECTION, *there exists an optimal solution without repeated directed edges.*

Proof. Assume not, and let $P = sP_1uvP_2uvP_3s$ be a solution minimizing $|P|$. Consider a walk $P' = sP_1u\overline{P_2}vP_3s$, where $\overline{P_2}$ is the reversed walk of P_2. Since $|P'| < |P|$ and both visit the same set of vertices, we have that $w(P') > w(P)$ by our choice of P. However, $w(P') = w(sP_1u) + w(u\overline{P_2}v) + w(vP_3s) \leq w(sP_1u) + w(u, v) + w(vP_2u) + w(u, v) + w(vP_3s) = w(P)$, a contradiction. □

The following is a simple observation.

Observation 6.3. *Given an instance of* GRAPH INSPECTION, *there exists a closed walk P of length $\hat{w}$ visiting vertices $V' \subseteq V(G)$ if and only if there exists a connected Eulerian multigraph $G' = (V', E')$ such that $\hat{w} = \sum_{uv \in E'} w(uv)$, where E' is the multiset of the edges in P.*

This leads to a structural lemma about solutions.

Lemma 6.4. *For any feasible instance of* GRAPH INSPECTION, *there exists an optimal solution with at most* $2n - 2$ *edges. This bound is tight.*

Proof. Let P be an optimal solution with the minimum number of edges. From Observation 6.3, we may assume there exists a connected Eulerian multigraph H that encodes P. Since H is connected, it has a spanning tree T as a subgraph. Let $H' = H - E(T)$. If H' contains a cycle C, then $H - C$ is also connected and Eulerian, as removing a cycle from a multigraph does not change the parity of the degree of each vertex. Hence, there exists a shorter solution P' that is an Euler tour in $H - C$, a contradiction. Knowing that both T and H' are acyclic, we have $|E(H)| = |E(T)| + |E(H')| \leq 2n - 2$. This is tight whenever G is a tree where all leaves have a unique color. □

Now we are ready to prove the correctness of the ILP formulation.

Theorem 6.5. *The ILP formulation in Figure 6.3 is correct.*

Proof. We show that we can translate a solution for GRAPH INSPECTION to a solution for the corresponding ILP and vise versa.

For the forward direction, let $P = v_0 v_1 \dots v_\ell$ with $v_0 = v_\ell = s$ be a solution walk with $\ell \geq 2$ collecting at least k colors. From Lemma 6.2, we may assume that there are no indices i, j such that $i < j$ and $v_i v_{i+1} = v_j v_{j+1}$. For constraint (1), we set $x_{u,v} = 1$ if $uv \in E(P)$ and 0 otherwise. It is clear to see that all flow conditions are satisfied. Moreover, observe that for any vertex $v \in V(G) \setminus \{s\}$, the flow amount $\sum_{u \in N(v)} x_{u,v}$ corresponds to the number of occurrences of v in P, which we denote by $\deg_P(v)$.

Next, if $|P| = 2$, then constraint (2) is trivially satisfied by setting $y_{e,v} = 0$ for all e, v. Otherwise, let P' be a continuous part of P such that s appears only at the beginning and at the end. Then, P' contains $|P'| - 2$ edges that do not touch s and emit two charges each. We know that $|P'| - 1 = \sum_{v \in V(P') \setminus \{s\}} \deg_{P'}(v)$. For a directed edge $e \in E(P')$ and its endpoint $v \in e$, let $y_{e,v}^{(P')}$ be part of $y_{e,v}$ charged only by P'. We distribute the charges by setting $y^{(P')}_{v'_{i-1}v'_i, v'_{i-1}} = 2 - \frac{2i}{|P'|-1}$ and $y^{(P')}_{v'_{i-1}v'_i, v'_i} = \frac{2i}{|P'|-1}$ for every $1 \leq i < |P'|$, where $P' = v'_0 v'_1 \dots v'_{|P'|}$ with $v'_0 = v'_{|P'|} = s$.

Note that $\sum_{u \in N(v) \setminus \{s\}} y^{(P')}_{uv,v} = \deg_{P'}(v) \cdot \frac{2(|P'|-2)}{|P'|-1} = (2 - \frac{2}{|P'|-1}) \cdot \deg_{P'}(v) \leq (2 - \frac{2}{2n-3}) \cdot \deg_{P'}(v)$ for every $v \in V(P') \setminus \{s\}$. The last inequality is due to Lemma 6.4. This inequality still holds when we concatenate closed walks P' from s since $\sum_{u \in N(v) \setminus \{s\}} y_{uv,v} = \sum_{P'} \sum_{u \in N(v) \setminus \{s\}} y^{(P')}_{uv,v}$ and $\sum_{u \in N(v)} x_{u,v} = \sum_{P'} \deg_{P'}(v)$. Constraint (2) is now satisfied.

Finally, in order to collect colors $\chi(v)$, there most be an edge uv in the solution. Notice that constraint (3b) encodes this and constraint (3c) ensures that we collect at least k distinct colors. Finally, observe that the objective is properly encoded.

For the backward direction, we show that there cannot be a closed flow, i.e., *circulation*, avoiding s. For the sake of contradiction, let C be such a circulation. Then, since $x_{u,v} = 1$ for every $uv \in E(C)$, we have $\sum_{uv=e \in E(C)} y_{e,u} + y_{e,v} = 2|E(C)|$. This is considered as the total charge emitted from C, and it must be consumed by the vertices in C. We have $\sum_{v \in V(C)} \sum_{u \in N(v)} y_{uv,u} + y_{uv,v} \geq 2|E(C)|$, and by the pigeonhole principle, there must be a vertex $v \in V(C)$ such that its charge consumption is at least $\deg(v)$, violating constraint (2b). Hence, there must be a closed walk from s that realizes a circulation obtained by ILP. From constraint (3), the walk also collects at least k colors. □

6.3.2.2 Solution recovery. A closed walk in a multigraph is called an *Euler tour* if it traverses every edge of the graph exactly once. A multigraph is called *Eulerian* if it admits an Euler tour. It is known that a connected multigraph is Eulerian if and only if every vertex has even degree [13] and given an Eulerian multigraph with m edges, we can find an Euler tour in time $\mathcal{O}(m)$ [19].

Given a certificate of an optimal solution for the aforementioned ILP, we construct a solution walk as follows. First, let D be the set of directed edges uv such that $x_{u,v} = 1$. Next, we find an Euler tour P starting from s using all the edges in D. Then, P is a solution for GRAPH INSPECTION.

6.3.3 Upper and Lower Bounds

When evaluating solutions, having upper and lower bounds on the optimal solution provides useful context. For GRAPH INSPECTION, a polynomial-time computable lower bound follows directly from the LP relaxation of the ILP in Section 6.3.2. For an upper bound, we consider Algorithm ST (Algorithm 5), which uses a 2-approximation algorithm

Algorithm 5: Algorithm ST

Input: A graph $G = (V, E)$, a set $\mathcal{C}$ of colors, an edge-weight function $w : E \to R_{\geq 0}$, a coloring function $\chi : V \to 2^{\mathcal{C}}$, a start vertex $s \in V$, and an integer t.
Output: A closed walk in G from s collecting at least t colors.

1 $S \leftarrow \{s\}$.
2 **while** $|\mathcal{C}| < t$ **do**
  // Choose the vertex with a new color closest to s.
3 $S \leftarrow S \cup \{\arg\min_{v \in V} d(s, v) \mid \chi(v) \setminus \chi(S) \neq \emptyset\}$
4 Compute a 2-approximation T for STEINER TREE on G with terminals S.
5 Construct a closed walk W from s using the all edges in T.
6 **return** W

for STEINER TREE[9] [21] as a subroutine. The algorithm proceeds by first choosing the vertices closest to s collecting t colors and then finding a Steiner tree of those vertices. A closed walk can be obtained by using each edge of the Steiner tree twice.

Theorem 6.6. Algorithm ST *returns a closed walk collecting at least t colors with length at most $t \cdot$ opt, where* opt *denotes the optimal walk length. The algorithm runs in $\mathcal{O}(tm \log(n + t))$ time.*

Proof. Since the algorithm returns a walk including all vertices in S, it collects at least t colors. Let $d_c = \min_{v \in \chi^{-1}(c)} d(s, v)$ for every $c \in \mathcal{C}$. Then, let $\tilde{d}$ be the t-th smallest such value, and due to Steps 1-3, for every $u \in S$, we have $d(s, u) \leq \tilde{d}$. Since $|S| \leq t + 1$, the weight of the minimum Steiner tree is at most $t\tilde{d}$, which results in that the length ℓ' of the walk returned by our algorithm is at most $2t\tilde{d}$. Now, suppose that P is an optimal walk of length opt collecting at least t colors $\mathcal{C}'$. Then, it is clear to see that opt $\geq 2 \cdot d_c$ for any $c \in \mathcal{C}'$. From $|\mathcal{C}'| \geq t$, we have opt $\geq 2\tilde{d}$, which implies $\ell' \leq t \cdot$ opt.

We next analyze the running time. Steps 1-3 takes $\mathcal{O}(m \log n + tn \log t)$ time for sorting vertices and computing the union of colors. Step 4 can be done by computing the transitive closure on S, which takes $\mathcal{O}(tm \log n)$ time. Step 5 takes $\mathcal{O}(n + m)$ time, so the overall running time is in $\mathcal{O}(tm \log(n + t))$.

Lastly, we show that this bound cannot be smaller. Let G be a star $K_{1,t+1}$ with s being the center with no colors. One leaf u has t colors, and each of the other t leaves has a

[9]The STEINER TREE problem takes a graph G and a set of vertices S (called *terminals*) and asks for a minimum-weight tree in G that spans S.

unique single color. Every edge has weight 1. The optimal walk is (s, u, s) and has length 2, whereas the algorithm may choose $V \setminus \{u\}$ as S. This gives a walk of length $2t$. □

6.4 Graph Simplification

This section introduces strategies for transforming our exact algorithms into heuristics with improved scalability, including principled sub-sampling of colors (POIs) and creating plans by merging walks which inspect different regions of the graph. Figure 6.4 illustrates this idea, the partition-and-merge framework.

6.4.1 Color Reduction

The algorithms of Sections 6.3.1 and 6.3.2 give us the ability to exactly solve GRAPH INSPECTION, but the running time is exponential in the number of colors (i.e., POIs). In some applications, this may not be prohibitive. In surgical robotics, a doctor may identify a small number of POIs which must be inspected to enable surgical intervention. However in other settings, it is unrealistic to assume that the number of colors is small. In the datasets we explore in Section 7.8 for instance, the POIs are drawn from a mesh of the object to be inspected, and we have no a priori information about the relative importance of inspecting individual POIs.

To deal with this challenge, we find a "representative" set $\mathcal{C}' \subseteq \mathcal{C}$ of colors, with $|\mathcal{C}'| = k$ small enough that our FPT algorithms run efficiently on the instance in which vertex colors are defined by $\chi(v) \cap \mathcal{C}'$ for each vertex v. We can then find a minimum-weight walk P on the color-reduced instance and reconstruct the set of inspected colors by computing $\bigcup_{v \in P} \chi(v)$.

Formally, we assume that there exists some function $f : \mathcal{C}^2 \to \mathbb{R}_{\geq 0}$ which encodes the "similarity" of colors, that is, for colors c_1, c_2, c_3, if $f(c_1, c_2) < f(c_1, c_3)$, then c_1 is more similar to c_2 than it is to c_3. In this chapter, the function f is always a Euclidean distance, but we emphasize that our techniques apply also to other settings. For example, one may imagine applications in which POIs are partitioned into categorical *types*, and it is desirable that some POIs of each type are inspected. In this case, one could define f as an indicator function which returns 0 or 1 according to whether or not the input colors are of the same type. The core idea behind our methods is to select a small set of colors having

Algorithm 6: GreedyMD

Input: $\mathcal{C}, \chi_0, f$, and a positive integer $k \leq |\mathcal{C} \setminus \chi_0|$.
Output: $\mathcal{C}' \subseteq \mathcal{C}$ with $|\mathcal{C}'| = k$.

```
1 C' ← χ_0.                                  // Every walk collects the colors visible from s.
2 while |C'| < k + |χ_0| do
      // Choose the most dissimilar color.
3     C' ← C' ∪ {argmax_{c∈C} min_{c'∈C'} f(c, c')}
4 C' ← C' \ χ_0;
5 return C'
```

maximum dispersal, meaning that as much as possible, every color in $\mathcal{C}$ should be highly similar (according to the function f) to at least one representative color in $\mathcal{C}'$.

We evaluate four algorithms for this task. The baseline (which we call Rand) selects colors uniformly at random. This is the strategy employed by IRIS-CLI when needed [15]. The second (called GreedyMD—MD for Maximum Dispersal) is a greedy strategy based on the Gonzalez algorithm for k-center [16]; this algorithm is described in more detail in Algorithm 6, where we set $\chi_0 = \chi(s)$. The final two algorithms (MetricMD, OutlierMD) are modified versions of this strategy. The interested reader is referred to Section 6.7 for a detailed description and the results of our comparative study. We note that all of our algorithms outperform the baseline Rand in terms of the resulting coverage. We perform our final comparisons (see Section 6.5.1) using GreedyMD for color reduction.

6.4.2 Merging Walks

When using DP-IPA or ILP-IPA on a color-reduced graph, the computed walk is minimum weight for the reduced color set, but the corresponding walk in the original graph may not collect many additional colors. To increase the coverage in the original graph, we merge two or more walks into a single closed walk. Now, the challenge is how to keep the combined walk short. Suppose we have a collection of W solution walks $\{P_i\}$ and want a combined walk P that visits all vertices in $\bigcup_i V(P_i)$. We model this task as the following problem[10].

[10] An underlying simple graph of a multigraph is obtained by deleting loops and replacing multiedges with single edges.

MINIMUM SPANNING EULERIAN SUBGRAPH

Input: A loopless connected Eulerian multigraph G and edge weights $w : E \to \mathbb{R}_{\geq 0}$, where E denotes the edges in G's underlying simple graph.

Problem: Find a spanning subgraph G' of G such that G' is a connected Eulerian multigraph minimizing the weight sum, *i.e.* $\sum_{e \in E(G')} w(e)$.

We showed in Section 6.3.2 (see Observation 6.3) that each solution walk for GRAPH INSPECTION is an Euler tour in a multigraph. We hence use these two characterizations interchangeably. Unfortunately, this problem is NP-hard as we can see by reducing from HAMILTONIAN CYCLE, which asks to find a cycle visiting all vertices in a graph. Given an instance G with n vertices of HAMILTONIAN CYCLE, we duplicate all the edges in G so that the graph becomes Eulerian. If we set a unit weight function w for $E(G)$, i.e., $w(e) = 1$ for every $e \in E(G)$, then G has a Hamiltonian cycle if and only if (G, w) has a spanning subgraph of weight n.

In this chapter, we propose and evaluate three simple heuristics for MINIMUM SPANNING EULERIAN SUBGRAPH: ConcatMerge, GreedyMerge, and ExactMerge. ConcatMerge simply concatenates all walks. Since all walks start and end at vertex s, their concatenation is also a closed walk. In Section 6.6, we give an algorithm which uses ConcatMerge and solves GRAPH INSPECTION optimally. We implemented a simplified version which is better by a factor of n in both running time and memory usage. GreedyMerge, detailed in Section 6.8, is a polynomial-time heuristic including simple preprocessing steps for MINIMUM SPANNING EULERIAN SUBGRAPH. At a high level, GreedyMerge builds a minimum spanning tree and removes as many redundant cycles as possible from the rest. ExactMerge is an exact algorithm using the ILP formulation for GRAPH INSPECTION.

6.4.2.1 Algorithm ExactMerge**.** We construct an instance of GRAPH INSPECTION by taking the underlying simple graph of the instance G of MINIMUM SPANNING EULERIAN SUBGRAPH. We pick an arbitrary vertex $s \in V(G)$ as the starting vertex, and set unique colors to the other vertices. After formulating the ILP for GRAPH INSPECTION with $t = n - 1$ (collecting all colors) as in Section 6.3.2, we add the following constraints: $x_{u,v} + x_{v,u} \leq 1$ for every edge $uv \in E(G)$ with multiplicity 1. Lastly, we map a solution for the ILP to the corresponding multigraph. This multigraph should be spanning as we collect all colors

in GRAPH INSPECTION, and it is by definition Eulerian. If each optimal solution contains at most $2n - 2$ edges (which we have already shown; see Lemma 6.4) and W is constant, then our ILP formulation has $\mathcal{O}(n)$ variables. Thus (unlike the original instance of GRAPH INSPECTION) we can often quickly solve the walk merging problem exactly.

6.4.3 Partitioning Colors

Because we want to combine multiple ($W > 1$) walks to form our solution, it is useful to first partition the colors. This way, each independently computed walk collects (at least) some disjoint subset of colors. We propose two algorithms. The first (which we call Ord-part, short for *ordered* partitioning) takes an ordered color set $\mathcal{C}$ as input and partitions it sequentially, i.e., by selecting the first $|\mathcal{C}|/W$ colors as one subset, the second $|\mathcal{C}|/W$ colors as another, and so on. The second (called Geo-part, short for *geometric partitioning*) executes GreedyMD with parameter W, and then partitions $\mathcal{C}$ by assigning each color to the most similar (according to the function f) of the W selected "representatives".

We also tested whether to perform color partitioning *before* or *after* color reduction. In the former case, the full color set $\mathcal{C}$ is partitioned by one of the algorithms described above[11], and then color reduction is performed on each subset. In the latter, color reduction is performed to obtain $W \cdot k$ colors, and then these colors are partitioned into W sets of size k using one of the algorithms described above. In Figure 6.5 we display the results of our partitioning experiments on the instances used in [15], one for a surgical inspection task (CRISP1000) and another for a bridge inspection task (DRONE1000); results for extended datasets are deferred to Figure 6.6. We now draw attention to two trends. First, we note that while using Ord-part before color reduction performs well in terms of coverage, particularly for the larger k values, we believe that this result is confounded somewhat by non-random ordering of the POIs in the input data. That is, we conjecture that the POIs arrive in an order which conveys some geometric information. Second, we note that while MetricMD seems to outperform GreedyMD (in terms of coverage) as a color reduction strategy for DRONE1000 with $k = 10$, this effect is lessened when $k = 20$. We believe that this trend is explainable, as for small k values the greedy procedure may select *only*

[11] In Figure 6.5, Ord-part is referred to as Label-part when it is performed before color reduction, to emphasize that in this case the partitioning is based on the (potentially not random) sequence of POI labels given as input.

peripheral POIs, while a larger k enables good representation of the entire space, including a potentially POI-dense "core" of the surface to be inspected. Given the complexity of the comparative results presented in Figure 6.5, we favor the simplest, most generalizable, and most explainable strategy. For this reason, the experiments of Section 6.5.1 are performed using GreedyMD to reduce colors before partitioning using Ord-part.

6.5 Empirical Evaluation

To assess the practicality of our proposed algorithms, we ran extensive experiments on a superset of the real-world instances used in [15]. Figure 6.7 shows an overview of the experiment pipeline. We first built RRGs using IRIS-CLI, originating from the CRISP and DRONE datasets (Figure 6.7 (a)). We tested IRIS-CLI and ILP-IPA on these instances with no additional color reduction. As IRIS-CLI iteratively outputs an s-t walk for some vertex $t \in V(G)$, we completed each walk with the shortest t-s path[12] to ensure a fair comparison while still giving as much credit as possible to IRIS-CLI (Figure 6.7 (b)). For DP-IPA, we filter and partition POIs to obtain 3 sets of k POIs, where $k = 10, 20$ (Figure 6.7 (c)). Then, we ran DP-IPA to exactly solve GRAPH INSPECTION for POI-reduced instances. In addition, we ran ILP-IPA for comparing color reduction/partitioning algorithms (Figure 6.7 (d)) and measured speedups of those algorithms with different number of threads (Figure 6.7 (e)). Lastly, we merged the walks using our algorithms to construct a "combined" closed walk (Figure 6.7 (f)). Here we define the "search time" for the combined walk as the total of the search times of single-run walks plus the time taken for merging walks[12]. Except in experiment (e), we set the time limit of each algorithm to 900 seconds (15 minutes), and used 80 threads for DP-IPA and ILP-IPA.

We tested on four GRAPH INSPECTION instances, two of which replicate the instances used in [15]. The first dataset, CRISP, is a simulation for medical inspection tasks of the Continuum Reconfigurable Incisionless Surgical Parallel (CRISP) robot [1, 24]. The dataset simulates a scenario segmented from a CT scan of a real patient with a pleural effusion—a serious medical condition that can cause the collapse of a patient's lung. The second, DRONE, is an infrastructure inspection scenario, in which a UAV with a camera is tasked with inspecting the critical structural features of a bridge. Its inspection points

[12]The time taken for augmenting and merging walks was negligible.

are the surface vertices in the 3D mesh model of a bridge structure used in [15]. To match the experiments in [15], we used IRIS-CLI to build RRGs with $n_{\text{build}} = 1000$ and, for CRISP, uniformly randomly selected 4200 POIs. We call these instances CRISP1000 and DRONE1000. Also, for each dataset, we built RRGs with $n_{\text{build}} = 2000$ (denoted CRISP2000 and DRONE2000). Refer to Section 6.9 for more information on our graph instances and experiment environment. Code and data to replicate all experiments are available at `https://github.com/TheoryInPractice/robotic-brewing`.

6.5.1 Comparison to IRIS-CLI

First we compare the overall performance of our proposed algorithms to that of IRIS-CLI. In this experiment, we ran IRIS-CLI with all original instances, ILP-IPA with all original instances and $t = \frac{i}{10} \cdot |\mathcal{C}|$ for $5 \leq i \leq 10$, and DP-IPA with POI-reduced instances accompanied by walk-merging strategies, GreedyMD (MetricMD in Appendix Figure 6.8), Ord-part *after* color reduction, and ExactMerge with $k \in \{10, 20\}$. We additionally computed the upper and lower bounds from Section 6.3.3 for all possible k values.

Figure 6.2 plots the coverage and weight of each solution obtained within the time limit. IRIS-CLI achieved around 87% coverage on both CRISP instances. ILP-IPA outperformed IRIS-CLI on CRISP1000 by providing (i) for $t = 0.8 \cdot |\mathcal{C}|$, slightly better coverage paired with a 30% reduction in weight, and (ii) for $t = |\mathcal{C}|$, perfect coverage with only a 16% increase in weight. On CRISP2000, ILP-IPA failed to find a solution except with $t = |\mathcal{C}|$. Meanwhile, DP-IPA was competitive with IRIS-CLI, finding walks with moderate reductions in weight at the expense of slightly reduced coverage (83%). The differences between IRIS-CLI and our algorithms are more significant on DRONE, where DP-IPA (with $k = 20$) outperformed IRIS-CLI by providing more coverage (68% vs. 64%) while reducing weight by over 50%. ILP-IPA outperformed IRIS-CLI by even larger margins on DRONE1000, but did not produce many solutions within the time limit on DRONE2000.

To summarize, ILP-IPA is the most successful on smaller instances, and works with various values of t. With larger graphs, ILP-IPA is more likely to time out when $t < |\mathcal{C}|$ (as described in Section 6.3.2, the ILP formulation in this case is more involved). DP-IPA is more robust on larger instances, outperforming IRIS-CLI in terms of solution weight while providing similar coverage.

6.5.1.1 Upper and lower bounds. First, we observe that the curvatures of upper bounds (recall Section 6.3.3) are quite different in CRISP and DRONE. We believe the geometric distribution of POIs explains this difference; with CRISP, the majority of POIs are close to the POIs seen at the starting point, which leads to concave upper-bound curves. DRONE, on the other hand, exhibits a linear trend in upper bounds because POIs are more evenly distributed in the 3D space, and the obtained solutions are far from these bounds. On the lower bound side, we obtain little insight on CRISP but see that on DRONE, it allows us to get meaningful bounds on the ratio of our solution's weight to that of an optimal inspection plan. For example, the lower bounds with $t = |\mathcal{C}|$ for DRONE1000 and DRONE2000 are 466.74 and 364.72, respectively. The best weights by ILP-IPA are 658.35 and 773.39, giving approximation ratios of 1.4 and 2.1 (respectively).

6.5.1.2 Multithreading analysis. Our implementations of DP-IPA and ILP-IPA both allow for multithreading, but IRIS-CLI cannot be parallelized without extensive modification (i.e., the algorithm is inherently sequential). Figure 6.9 illustrates order-of-magnitude runtime improvements for DP-IPA when using multiple cores. Analogous results for ILP-IPA are deferred to Section 6.9.1.

6.5.1.3 Empirical analysis of walk-merging algorithms. The results reported in Figure 6.2 are all computed using ExactMerge for walk merging. We also evaluated the effectiveness of ConcatMerge and GreedyMerge in terms of both runtime and resulting (merged) walk weight. We observed that while GreedyMerge is a heuristic, it produces walks of nearly optimal weight with negligible runtime increase as compared to ConcatMerge.

Meanwhile, the runtime of ExactMerge was always within a factor of two of ConcatMerge; given the small absolute runtimes (<0.1 seconds in all cases), we chose to proceed with ExactMerge to minimize the weight of the merged walk. Complete experimental results are shown in Section 6.9.

6.5.1.4 Comparing ILP-IPA and DP-IPA. In this work we have contributed two new GRAPH INSPECTION solvers, namely DP-IPA and ILP-IPA. We conclude this section with a brief discussion of their comparative strengths and weaknesses. As discussed previously, for small graphs (e.g., CRISP1000 and DRONE1000) ILP-IPA is clearly the best option, as it provides higher coverage with less weight. However, DP-IPA performs better when the graph is large. In particular, if in some application minimizing weight is more important

than achieving perfect coverage, then DP-IPA is preferable in large graphs. One might ask whether this trade-off (between weight and coverage) can also be tuned for ILP-IPA by setting $t < |\mathcal{C}|$, but we emphasize that in practice this choice significantly increases the runtime of ILP-IPA, such that it is impractical on large graphs. This is clear from the data presented in Figure 6.2, and is also detailed in Section 6.9.2. We observe that ILP-IPA becomes less competitive with DP-IPA as n and k grow.

6.6 Walk Merging: Optimality in the Limit

We argue that our strategy for merging walks is a simplification of an algorithm that is optimal in the limit, given sufficient runtime. Note that the dynamic program behind Theorem 6.1 is optimal (always produces a walk of minimum weight that collects the given colors). A very simple strategy that is also optimal is to select an arbitrary permutation $(c_1, c_2, \ldots, c_k)$ of the colors and then compute a shortest walk that collects all colors in the respective order. If we repeat this for each possible permutation, then at some point, we will find an optimal solution.

We now observe that computing a walk that collects all colors in the guessed order can be computed in polynomial time by the following dynamic program T that stores for each vertex v and each integer $i \in [k]$ the length of a shortest walk between s and v that collects the first i colors in the guessed order. Therein, we use a second table D.

$$D[v,i] = \begin{cases} T[u,i-1] + \text{dist}(u,v) & \text{if } c_i \in \chi(v) \\ \infty & \text{else} \end{cases}$$
$$T[v,i] = \min_{u \in V} D[u,i] + \text{dist}(u,v)$$

We mention that we assume that $\text{dist}(v,v) = 0$ for each vertex v.

We now modify the above strategy to achieve a better success probability than when permutations are chosen randomly. Instead of guessing the entire sequence of colors, we guess *buckets* of colors, that is, a sequence of c sets for some integer c that form a partition of the set of colors into sets of size k/c (appropriately rounded, for this presentation, we will assume that k is a multiple of c). Note that there are $k!/((k/c)!)^c$ possible guesses for such buckets. For each bucket, let S_i be the set of colors in the bucket. We can now compute for each pair u, v of vertices a shortest walk between u and v that collects all colors in S_i by running the dynamic program behind Theorem 6.1 for color set S_i from all vertices $u \in V$.

Let this computed value be $S[u, v, S_i]$. Given a guess for a sequence of buckets, we can now compute an optimal solution corresponding to this guess by modifying the above dynamic program as follows.

$$T'[v, 1] = S[s, v, S_1]$$
$$T'[v, i] = \min_{u \in V} T[u, i-1] + S[u, v, S_i] \text{ if } i > 1$$

Note that $T'[v, i] \leq T[v, ci]$ if the sequence used to compute T corresponds to the set of buckets used to compute T'. This algorithm again achieves optimality in the limit (that is, given enough runtime, it will find a minimum weight walk).

In order to avoid computing S for all pairs of vertices, we decided to implement a simplification where we only compute $S'[S_i] = S[s, s, S_i]$. This version is not guaranteed to find an optimal solution like the full DP above, as there are examples where no optimal solution returns back to s before the very end. As an example, consider a star graph where s is a leaf and each other leaf has a unique color. However, this simplification is faster by a factor of n and uses a factor of n less memory while performing well in practice.

6.7 Color Reduction: Details

In this section, we describe our color reduction strategies in more detail and provide empirical data comparing their effectiveness.

Initializing GreedyMD. The Gonzalez [16] algorithm on which GreedyMD is based requires that the set $\mathcal{C}'$ be initialized with at least one color. The simplest strategy is to simply add a uniformly randomly selected color to $\mathcal{C}'$ and then proceed with the algorithm. In practice, we found it more effective to set $\mathcal{C}' = \chi(s) \neq \emptyset$ (see Algorithm 6). That is, we begin by adding some color which is visible from the source vertex s. We then greedily add k more colors. At the end of the algorithm, we return $\mathcal{C}' \setminus \chi(s)$. Intuitively, the justification for this approach is that we collect the colors $\chi(s)$ "for free" since every solution walk begins at s. Consequently, we do not need to ensure that $\mathcal{C}'$ is representative of these colors, or of colors which are very similar to them. Empirically, we found that this initialization strategy significantly improved the coverage of the resulting solutions.

Introducing MetricMD and OutlierMD. A potential shortcoming of GreedyMD is that it favors outlier colors. That is, because at each iteration it chooses the color which is most

Algorithm 7: OutlierMD

Input: $\mathcal{C}, \chi_0, f$, a positive integer $k \leq |\mathcal{C} \setminus \chi_0|$, and $r \in \mathbb{R}_{\geq 1}$.
Output: $\mathcal{C}' \subseteq \mathcal{C}$ with $|\mathcal{C}'| = k$.

$\mathcal{C}' \leftarrow \mathsf{GreedyMD}(\mathcal{C}, \chi_0, f, \lfloor rk \rfloor)$ // Choose at most (rk) representatives.
for $c' \in \mathcal{C}'$ **do**
  $X_{c'} \leftarrow \{c'\}$ // Initialize clusters for each $c' \in \mathcal{C}'$.
for $c \in \mathcal{C}$ **do**
  $\tilde{c} = \operatorname{argmin}_{c' \in \mathcal{C}'} f(c, c')$
  $X_{\tilde{c}} \leftarrow X_{\tilde{c}} \cup \{c\}$ // Add color c to the closest cluster.
Sort elements of $\mathcal{C}' : c'_1, c'_2, \ldots c'_{rk}$ according to the size of $X_{c'_i}$ (descending).
return $\{c'_1, c'_2, \ldots c'_k\}$

dissimilar to the previously selected colors, we can be sure that outlier colors which are very dissimilar to every other color will be selected. This may be undesirable for two reasons. First, if the similarity function f is correlated to colors being visible from the same vertices, then discarding outliers from $\mathcal{C}'$ may improve coverage (as computed on $\mathcal{C}$). Second, if the similarity function $f(c_1, c_2)$ is correlated to the shortest distance between vertices labeled with c_1 and c_2, then discarding outliers from $\mathcal{C}'$ may reduce the weight needed for a walk collecting all colors in $\mathcal{C}'$.

We designed and tested two strategies to mitigate these effects. The first, OutlierMD (see Algorithm 7), uses an additional scaling parameter $r \geq 1$. GreedyMD is used to form a representative color set $\mathcal{C}'$ of size rk. Next, $\mathcal{C}'$ is partitioned into rk clusters by assigning each color $c \in \mathcal{C}$ to a cluster uniquely associated with the representative $c' \in \mathcal{C}'$ to which c is most similar. Finally, we return the k colors in $\mathcal{C}'$ associated with the largest clusters.

The second strategy, MetricMD, assumes that our colors are embeddable in a metric space. This is true, for example, when colors represent positions in $\mathbb{R}^3$ on some surface mesh of the object to be inspected. In this case, we begin by using GreedyMD to find a representative colors set $\mathcal{C}'$ of size k. Next, we perform k-means clustering on $\mathcal{C}$, using $\mathcal{C}'$ as the initial centroids. The resulting centroids are positions in space, and may not perfectly match the positions of any colors. To deal with this, we simply choose the colors closest to the centroids, and return these as our representative color set. See Algorithm 8.

6.7.0.1 Empirical evaluation of color reduction schemes. We experimentally evaluated our color reduction schemes (GreedyMD, OutlierMD, and MetricMD) along with the baseline Rand on each of our four datasets, with $k \in \{10, 20\}$. To perform the evaluation,

Algorithm 8: MetricMD

Input: $\mathcal{C}, \chi_0, f$, and a positive integer $k \leq |\mathcal{C} \setminus \chi_0|$.
Output: $\mathcal{C}' \subseteq \mathcal{C}$ with $|\mathcal{C}'| = k$.

$\mathcal{C}' \leftarrow \text{GreedyMD}(\mathcal{C}, \chi_0, f, k)$
$S \leftarrow k\text{-Means}(\mathcal{C} \setminus \chi_0, \mathcal{C}')$ // Run k-Means on $\mathcal{C} \setminus \chi_0$ with initial centroids $\mathcal{C}'$.
$S' \leftarrow \emptyset$
for $s \in S$ **do**
  // For each centroid, choose the closest color.
  $S' \leftarrow S' \cup \{\text{argmin}_{c \in \mathcal{C}' \setminus S'} d(c, s)\}$
return S'

each algorithm was used to create a representative set of k colors, and then DP-IPA was used to solve GRAPH INSPECTION on the color-reduced graphs. We chose DP-IPA rather than ILP-IPA for comparison because the former is the solver which needs color reduction to compute walks on our datasets (i.e., ILP-IPA can run on the DRONE and CRISP datasets without any color reduction). The results of these experiments are displayed in Figure 6.10. In each plot displayed in Figure 6.10, every colored dot represents a GRAPH INSPECTION solution computed by DP-IPA after color reduction performed by either Rand, GreedyMD, OutlierMD, or MetricMD. For each color reduction strategy, the solution with the highest coverage (in the original, non-color-reduced graph) is indicated with a rightward-pointing arrow, and the solution with minimum weight is indicated with a downward-facing arrow.

Because our color reduction schemes are designed to ensure good coverage, we are primarily interested in comparing the coverage of solutions. The results indicate that, in general, our strategies outperform the baseline Rand in terms of coverage, often by large margins. In terms of coverage, the comparative performances of GreedyMD, MetricMD, and OutlierMD are somewhat difficult to disentangle. Given that GreedyMD is the simplest of the three, the most explainable, and the most generalizable (it does not require a metric embedding or any parameter tuning), we favor it for future experiments. However, we leave as an interesting open direction to perform a more extensive comparison of these methods on a larger data corpus.

We conclude this section with a discussion of the weight of solutions. We are not surprised that our strategies tend to produce higher-weight solutions than Rand since we are optimizing for coverage even if it means requiring that a solution walk visit outlier POIs. In

particular, it is expected that GreedyMD performs poorly with respect to solution weight, since it explicitly favors outlier POIs. In applications where the weight of the solution is of paramount interest, even at the expense of lowering coverage, the aforementioned extended comparison of color reduction strategies may be of particular interest.

6.8 Walk-Merging Algorithms

In this section, we present preprocessing steps for the MINIMUM SPANNING EULERIAN SUBGRAPH and the GreedyMerge algorithm in detail.

6.8.0.1 Preprocessing for MINIMUM SPANNING EULERIAN SUBGRAPH**.** We say an edge set $\tilde{E} \subseteq E(G)$ is *undeletable* if there is an optimal solution for MINIMUM SPANNING EULERIAN SUBGRAPH including all the edges in $\tilde{E}$. We apply the following rules as preprocessing.

- RULE 1: If there is an edge with multiplicity at least 4, then decrease its multiplicity by 2. This makes the multiplicity of every edge either 1, 2 or 3.
- RULE 2: If there is an edge cut $e_1, e_2 \in E(G)$ of size 2, then mark e_1 and e_2 as undeletable. For example, this includes (but is not limited to) the following:
  - edges incident to a vertex with degree 2.
  - an edge with multiplicity 2 that forms a bridge in the underlying simple graph of G.

6.8.0.2 Algorithm GreedyMerge**.** Consider the following heuristic for MINIMUM SPANNING EULERIAN SUBGRAPH.

This algorithm runs in $\mathcal{O}(m \log n)$ time. In practice, we apply (part of) RULE 2 to find undeletable edges and include them in S in Line 2. To prove the correctness of data reduction rules and algortihms, we start by a simple, well-known observation on Eulerian graphs.

Observation 6.7. *Let C be a closed walk in an Eulerian multigraph G. If $G - C$ is connected, then $G - C$ is also Eulerian.*

Algorithm 9: GreedyMerge

Input: A loopless connected Eulerian multigraph G and edge weights $w\colon E \to \mathbb{R}_{\geq 0}$, where E denotes the edges in G's underlying simple graph.
Output: Spanning subgraph G' of G such that G' is a connected Eulerian multigraph.

Apply RULE 1 exhaustively.
Construct a minimum spanning tree S of G using a known algorithm.
// Greedily find a maximal cycle packing $\mathcal{C}$ in $G - S$.
$\mathcal{C} \leftarrow \emptyset$
Let U be the empty multigraph with vertices $V(G)$.
for $e \in E(G - S)$ *in order of nonincreasing weights* **do**
    $U \leftarrow (V(U), E(U) \cup \{e\})$ // Add edge e to U.
    if U *contains a cycle* C **then**
        $\mathcal{C} \leftarrow \mathcal{C} \cup \{C\}$
        $U \leftarrow U - C$
return $G - \bigcup_{C \in \mathcal{C}} E(C)$

Proof. Removing a closed walk does not change the parity of the degree at each vertex. If all vertices in G have even degrees, so do those in $G - C$, which implies that $G - C$ is Eulerian if it is connected. □

We continue with the analysis of our two reduction rules.

Lemma 6.8. RULE 1 *is safe.*

Proof. By Lemma 6.2, there must be an optimal solution with edge multiplicity at most 2. From Observation 6.7, if there is an edge with multiplicity at least 4, then removing 2 of them (which can be seen as a closed walk) results in a connected Eulerian multigraph. □

Lemma 6.9. RULE 2 *is safe.*

Proof. Let $S \subset V(G)$ be a nonempty vertex set such that e_1, e_2 separates S from $V(G) \setminus S$. Then, any closed walk that visits $V(G)$ must contain at least one edge from S to $V(G) \setminus S$ and another from $V(G) \setminus S$ to S. Hence, both e_1 and e_2 must be in any solution. □

We conclude this section by analyzing GreedyMerge.

Theorem 6.10. GreedyMerge *correctly outputs a (possibly suboptimal) solution for* MINIMUM SPANNING EULERIAN SUBGRAPH *in* $\mathcal{O}(m \log n)$ *time.*

Proof. We need to show that GreedyMerge outputs a spanning Eulerian subgraph G' (in this context, a *subgraph* is also a multigraph) of G. From Line 2, we know that S is a spanning subgraph of G. From Line 5, we have that $E(C) \subseteq E(G - S)$ for every $C \in \mathcal{C}$. From Line 10, G' is clearly a subgraph of G, and from $\bigcup_{C \in \mathcal{C}} E(C) \subseteq E(G - S)$, we have $E(G') \supseteq E(S)$, and hence G' is spanning. Knowing that G' is connected and from Observation 6.7, a multigraph constructed by removing any closed walk remains Eluerian. Hence, G' is a spanning Eluerian subgraph of G.

We next analyze the running time. RULE 1 can be executed exhaustively in $\mathcal{O}(m)$ time by iterating over all edges. At this point, $|E(G)| \leq 3 \cdot \binom{n}{2} \leq 2n^2$. Now let us analyze the rest of the steps in GreedyMerge. The running time of Line 2 is the same as that of Kruskal's algorithm, which is $\mathcal{O}(m \log m) \subseteq \mathcal{O}(m \log(2n^2)) = \mathcal{O}(m \log n)$. Similarly, Lines 5 to 9 has the same running time as we iteratively examine edges and check for connectivity. Hence, the overall running time is in $\mathcal{O}(m \log n)$. □

6.9 Experiment Details and Supplemental Results

Table 6.1 summarizes the test instances we used for our experiment.

6.9.0.1 Experiment environment. We implemented our code with C++ (using C++17 standard). We ran all experiments on identical hardware, equipped with 80 CPUs (Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz) and 191000 MB of memory, and running Rocky Linux release 8.8. We used Gurobi Optimizer 9.0.3 as the ILP solver, parallelized over CPUs.

6.9.1 Multithreading Analysis

Here, we present results of our multithreading experiments for ILP-IPA (visualized in Figure 6.11). We configured the Gurobi ILP solver to output each feasible solution as soon as it is found, so to understand the impact of multithreading we are interested in determining the lowest-weight feasible solution identified in a given time limit, for various thread counts. Predictably, the general trend is clear: multi-threaded implementations provide lower-weight solutions faster than single threaded solutions.

6.9.2 Additional Empirical Results and Figures

Based on our findings in Section 6.7, we also compared all three inspection planning algorithms when DP-IPA is paired with MetricMD color reduction (instead of GreedyMD, as shown in Figure 6.2). The results are shown in Figure 6.8.

They are qualitatively quite similar, but we observe that this approach has the disadvantage of requiring the POI similarity function to be a metric.

We showed the results of applying our color partitioning methods in combination with GreedyMD and MetricMD for the 1000-node graphs in Figure 6.10; the analogous results for the 2000-node graphs are included below in Figure 6.6.

After applying color reduction and partitioning, we have a set of walks that need merging, as discussed in Section 6.4.2 with additional details in Section 6.8. The empirical results of comparing these approaches are shown in Figure 6.12.

Table 6.2 compares the solution weights generated by DP-IPA and ILP-IPA. In this table, both DP-IPA and ILP-IPA are used to produce solutions on color-reduced graphs. The table shows solution weights by ILP-IPA relative to the optimal weights that DP-IPA produces. The trend is clear: ILP-IPA becomes less competitive with DP-IPA as n (n_{build}) and k grow.

6.10 Conclusion

In this work, we took tangible and meaningful steps toward mapping the GRAPH INSPECTION planning problem in robotics to established problems (e.g., GENERALIZED TSP). We presented two algorithms, DP-IPA and ILP-IPA, to solve the problem under this abstraction, based on dynamic programming and integer linear programming. We presented multiple strategies for leveraging these algorithms on relevant robotics examples lending insight into the choices that can be made to use these methods in emerging problems. We then evaluated these methods and strategies on two complex robotics applications, outperforming the state of the art.

Our approach of creating several reduced color sets and merging walks offers a new paradigm for leveraging algorithms whose complexity has high dependence on the number of POIs, and opens the door for future exploration. We plan to see how these methods perform and scale with more than three walks. Further, it remains to implement these algorithms on real-world, physical robots and inspection tasks.

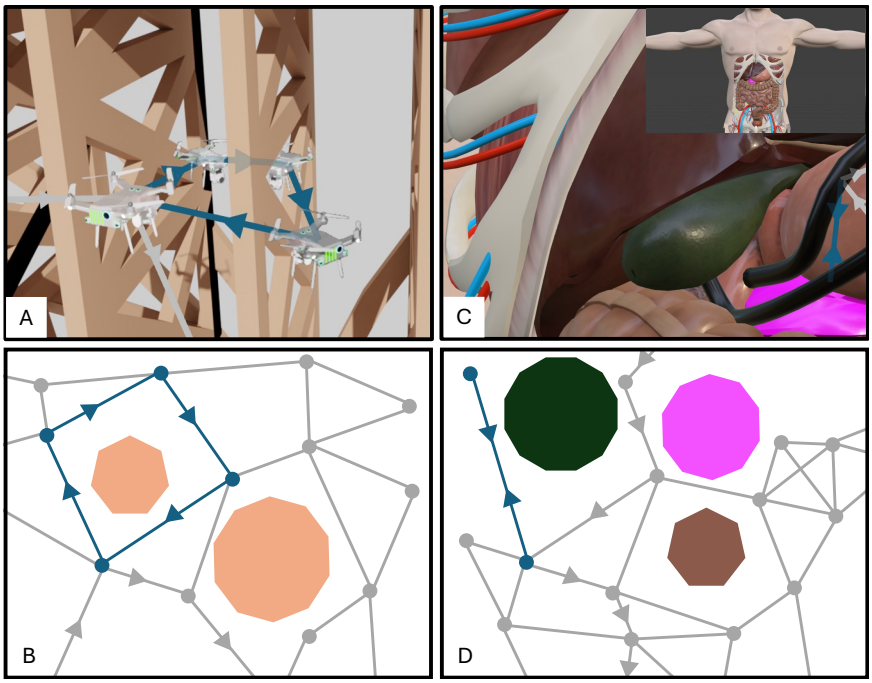


Figure 6.1: Inspection planning, in contrast to traditional motion planning, may necessitate leveraging cycles and backtracking on graphs embedded in the robot's configuration space. This necessitates computing a walk (rather than a path) on a graph. (A) A quadrotor, while inspecting a bridge for potential structural defects, may need to circle around obstacles, (B) leveraging a cycle in its c-space graph (teal). (C) A medical endoscopic robot (black) may need to move into and then out of an anatomical cavity to, e.g., visualize the underside of a patient's gallbladder (green), (D) requiring backtracking in its c-space graph (teal).

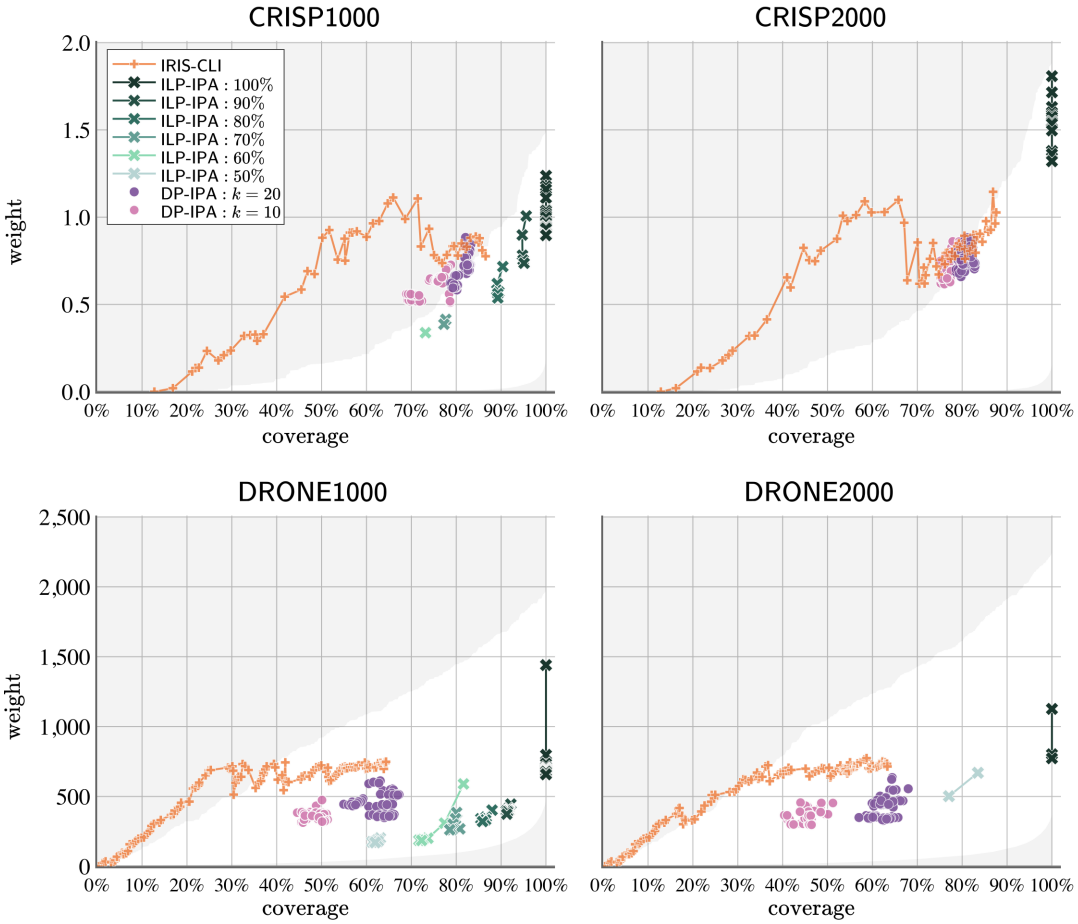


Figure 6.2: Performance of IRIS-CLI, ILP-IPA and DP-IPA (with GreedyMD) on DRONE and CRISP benchmarks. Each data point represents a computed inspection plan; coverage is shown as a percentage of all POIs in the input graph. The area shaded in gray is outside the upper/lower bounds given in Section 6.3.3.

Variables & Objective

$x_{u,v}, x_{v,u}$	$\in \{0,1\}$	for $uv \in E$	*$x_{u,v}$ is 1 iff the directed edge uv is present in the solution flow.*
$y_{e,v}$	≥ 0	for $e \in E, v \in e$	*Amount of charges the edge e emits to its endpoint v.*
z_c	$\in \{0,1\}$	for $c \in \mathcal{C}$	*Only used if $t < \lvert\mathcal{C}\rvert$. Value 1 iff color c is collected.*
$\min \sum_{uv \in E} w(uv)(x_{u,v} + x_{v,u})$			*Minimize weighted cost of walk.*

Flow conditions

1a	$\sum_{u \in N(v)} x_{u,v} - x_{v,u} = 0$	for every $v \in V$	*Flow preservation.*
1b	$\sum_{u \in N(s)} x_{s,u} \geq 1$		*Outgoing flow at source s.*

Prohibit closed flow disjoint from s

2a	$y_{uv,u} + y_{uv,v} = 2(x_{u,v} + x_{v,u})$	for every $uv \in E(G - s)$	*Solution edges not incident to s emit charges.*
2b	$\sum_{u \in N(v) \setminus \{s\}} y_{uv,v} \leq \left(2 - \frac{2}{2n-3}\right) \cdot \sum_{u \in N(v)} x_{u,v}$	for every $v \in V \setminus \{s\}$	*Vertices can consume slightly less than two charges per incoming flow.*

Count number of collected colors

Case $t = |C|$

3a	$\sum_{v \in \chi^{-1}(c)} \sum_{u \in N(v)} x_{u,v} \geq 1$	for every $c \in C$	*Collect every color.*

Case $t < |\mathcal{C}|$

3b	$z_c \leq \sum_{v \in \chi^{-1}(c)} \sum_{u \in N(v)} x_{u,v}$	for every $c \in C$	*Walk must cross vertex of color c to collect c.*
3c	$\sum_{c \in C} z_c \geq t$		*Collect at least t colors.*

Figure 6.3: ILP-IPA, an ILP for the GRAPH INSPECTION problem.

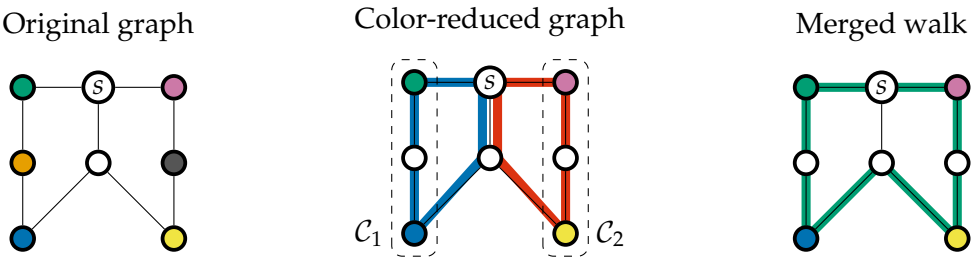


Figure 6.4: An illustration of the partition-and-merge framework. The color set in the original graph (left) is reduced to 2 color sets $\mathcal{C}_1$ and $\mathcal{C}_2$, each of which contains 2 colors (middle). For each color set, we find an optimal walk collecting all colors in the set, resulting in the blue and red walks. Those walks are merged into the green walk, collecting the same colors in the color-reduced graph (right).

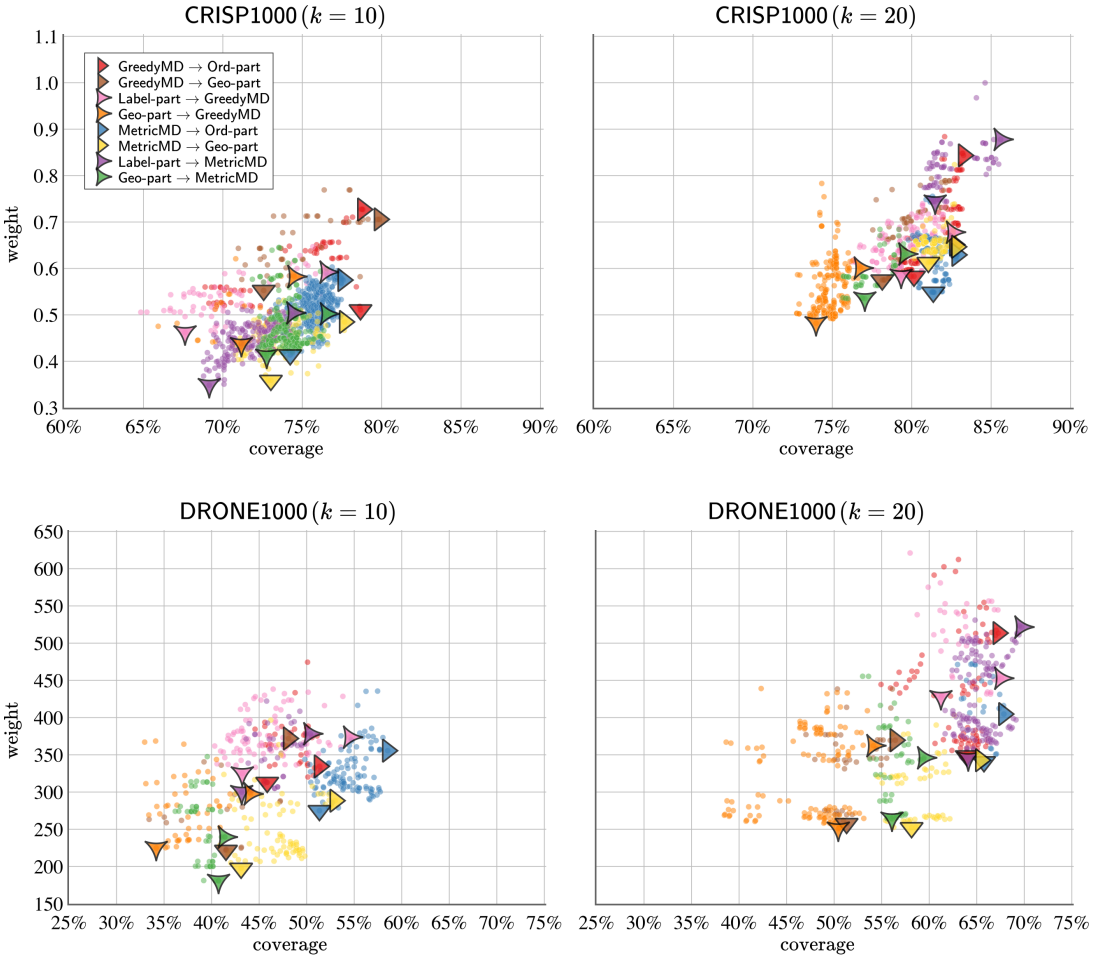


Figure 6.5: Selected results of color partitioning experiment on datasets CRISP1000 and DRONE1000 with $k \in \{10, 20\}$. Each data point represents a solution computed using DP-IPA and ExactMerge. For each combination of color reduction method and partitioning strategy, we highlight the solutions with maximum coverage (rightward arrow) and with minimum weight (downward arrow).

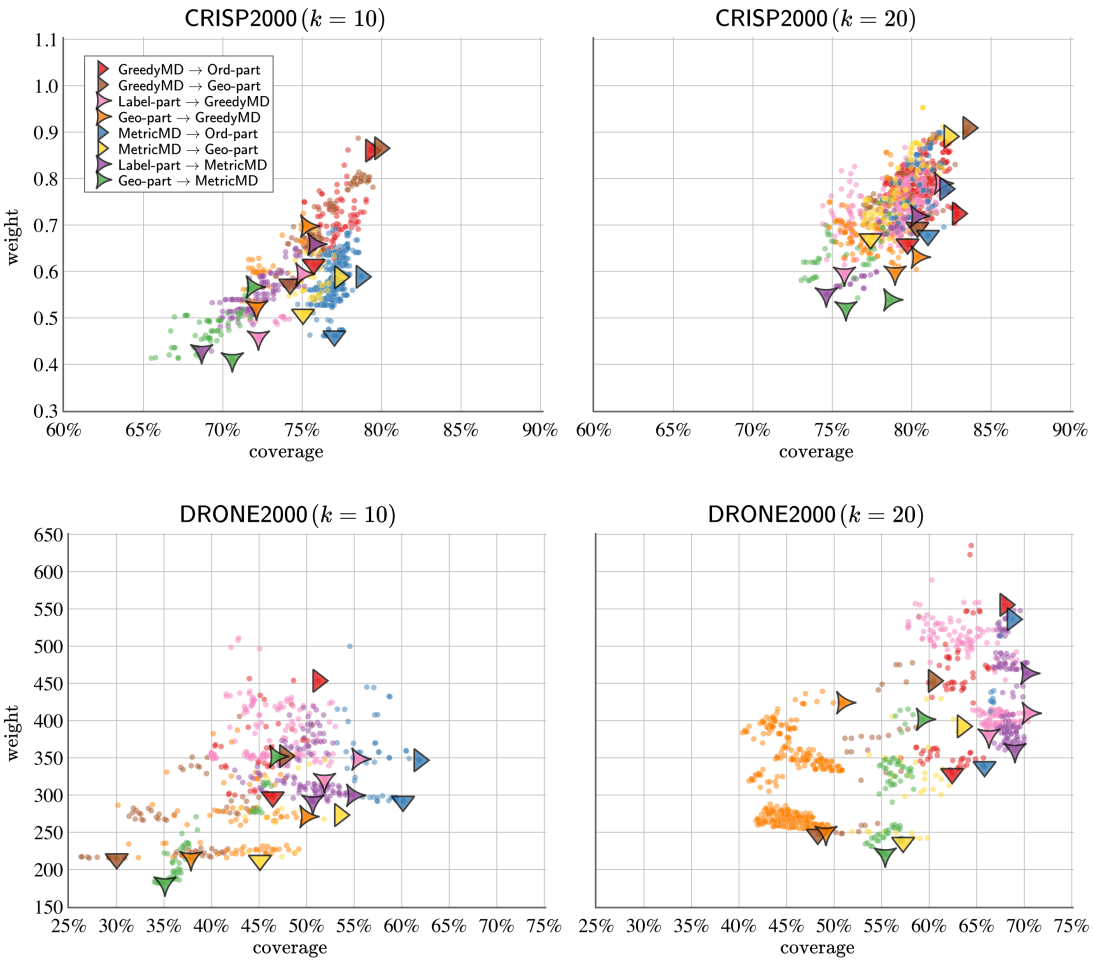


Figure 6.6: Results of our color partitioning experiments for datasets CRISP2000 and DRONE2000 with $k \in \{10, 20\}$. Every data point represents a solution computed using DP-IPA and ExactMerge. For each combination of color reduction and color partitioning strategies, the solution with maximum coverage is indicated with a rightward-pointing arrow, and the solution with minimum weight is indicated with a downward-pointing arrow.

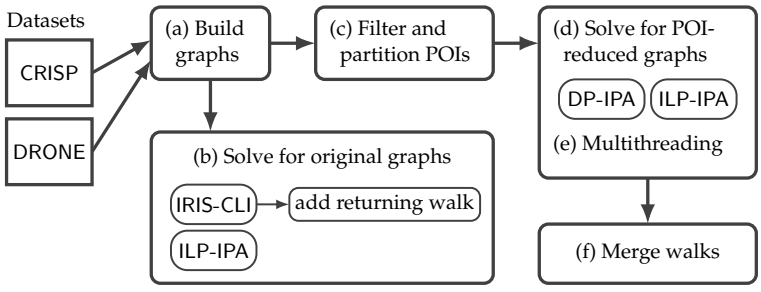


Figure 6.7: Overview of our experiment pipeline.

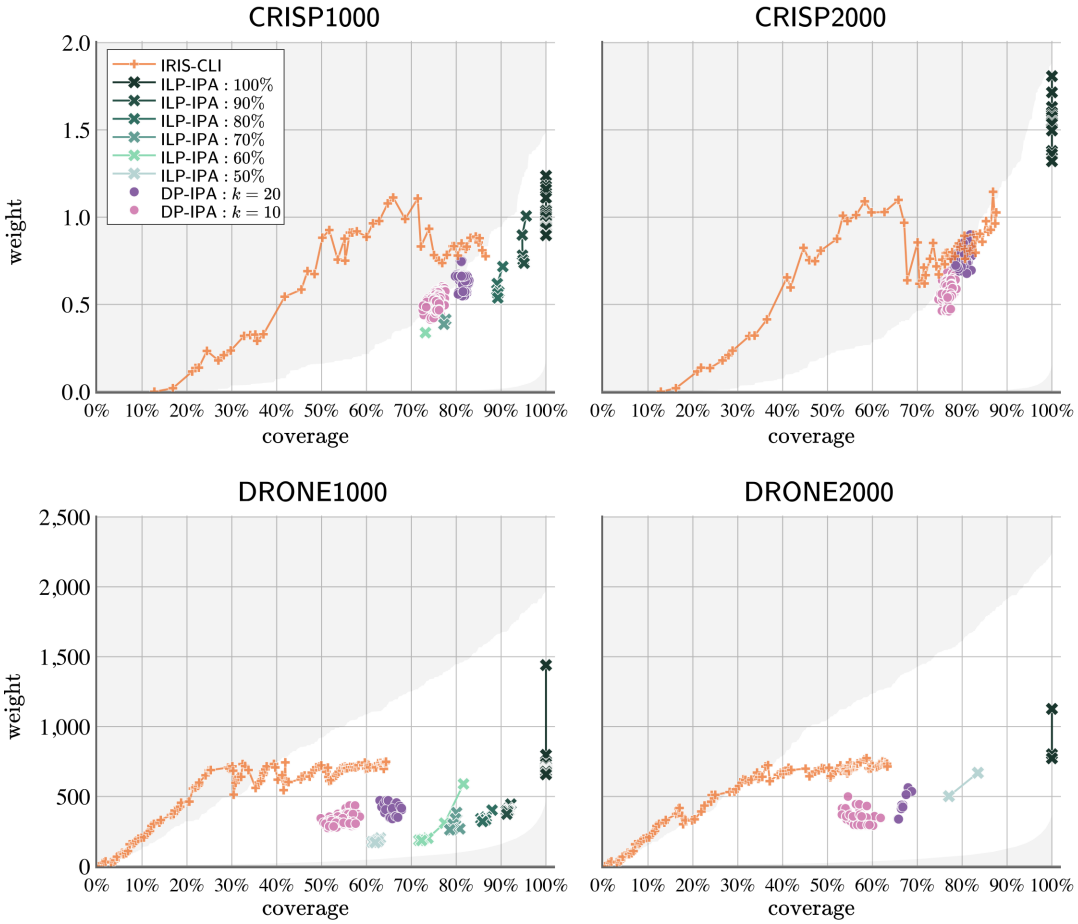


Figure 6.8: Performance of IRIS-CLI, ILP-IPA and DP-IPA (with MetricMD) on DRONE and CRISP benchmarks. Each data point represents a computed inspection plan; coverage is shown as a percentage of all POIs in the input graph. The area shaded in gray is outside the upper/lower bounds given in Section 6.3.3.

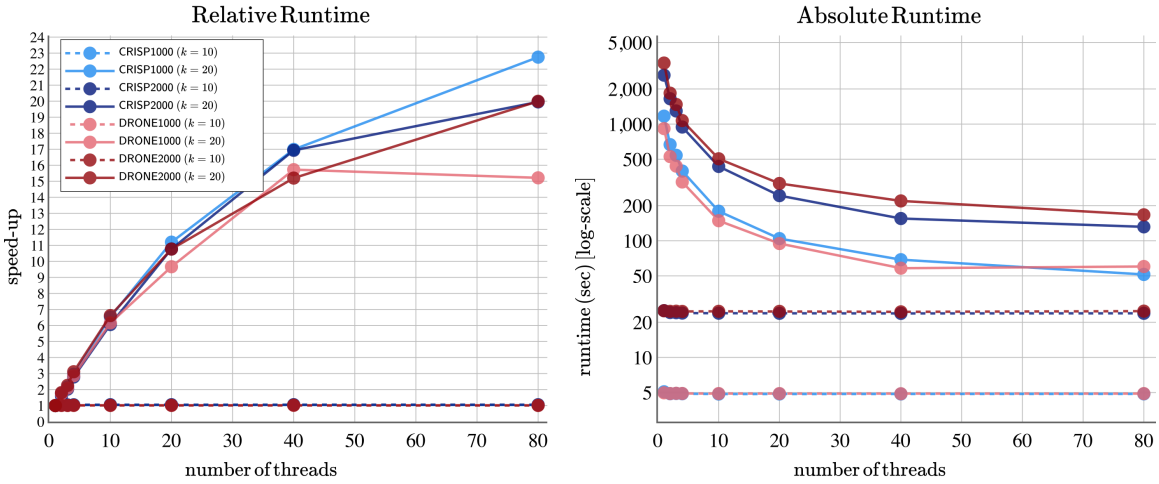


Figure 6.9: Relative (left) and absolute (right) runtimes for DP-IPA with 1-80 threads. The relative runtime (speed-up) is with respect to a single thread.

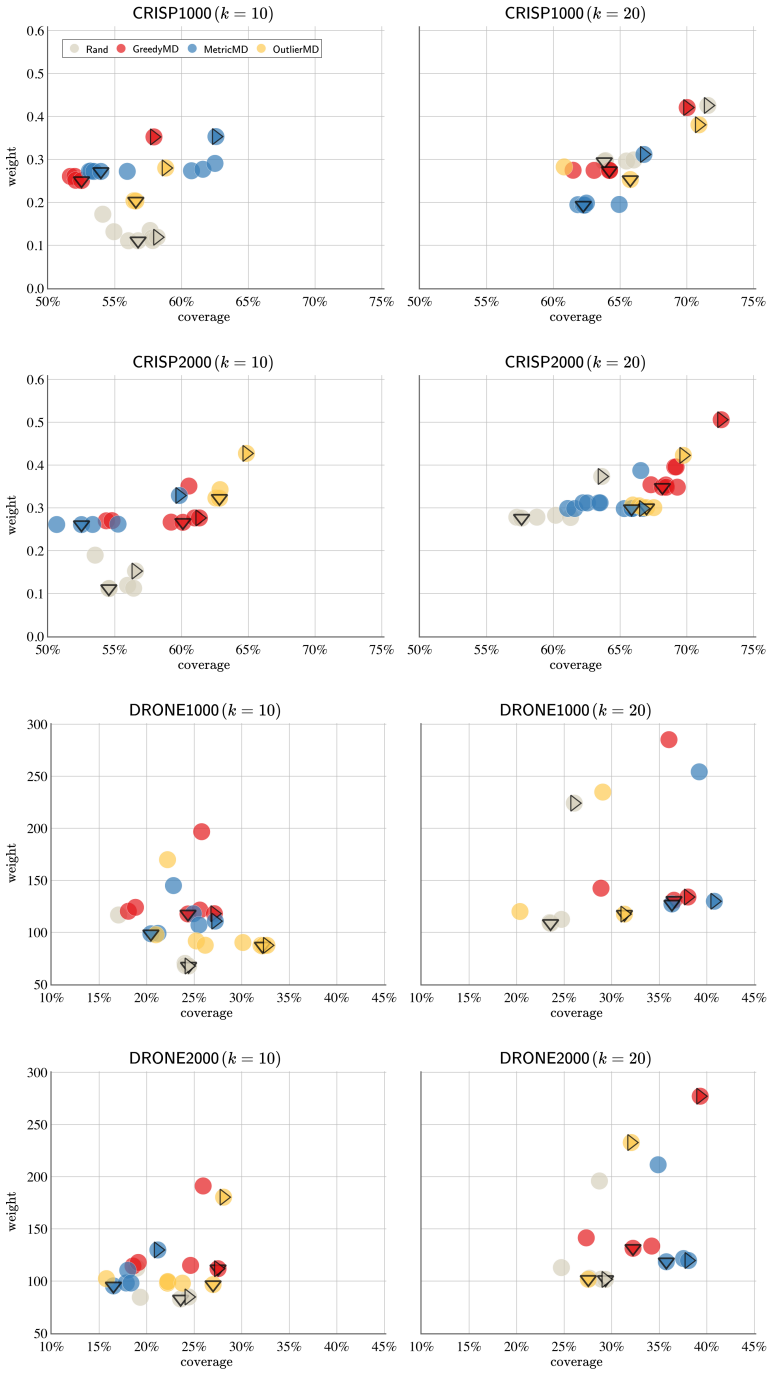


Figure 6.10: Results of color reduction experiments for Rand, GreedyMD, OutlierMD, and MetricMD. Each data point represents a GRAPH INSPECTION solution generated by DP-IPA. For each algorithm, the solution with the highest coverage (computed in the original, non-color-reduced graph) is marked with a rightward-pointing arrow, and the minimum weight solution is marked with a downward-pointing arrow.

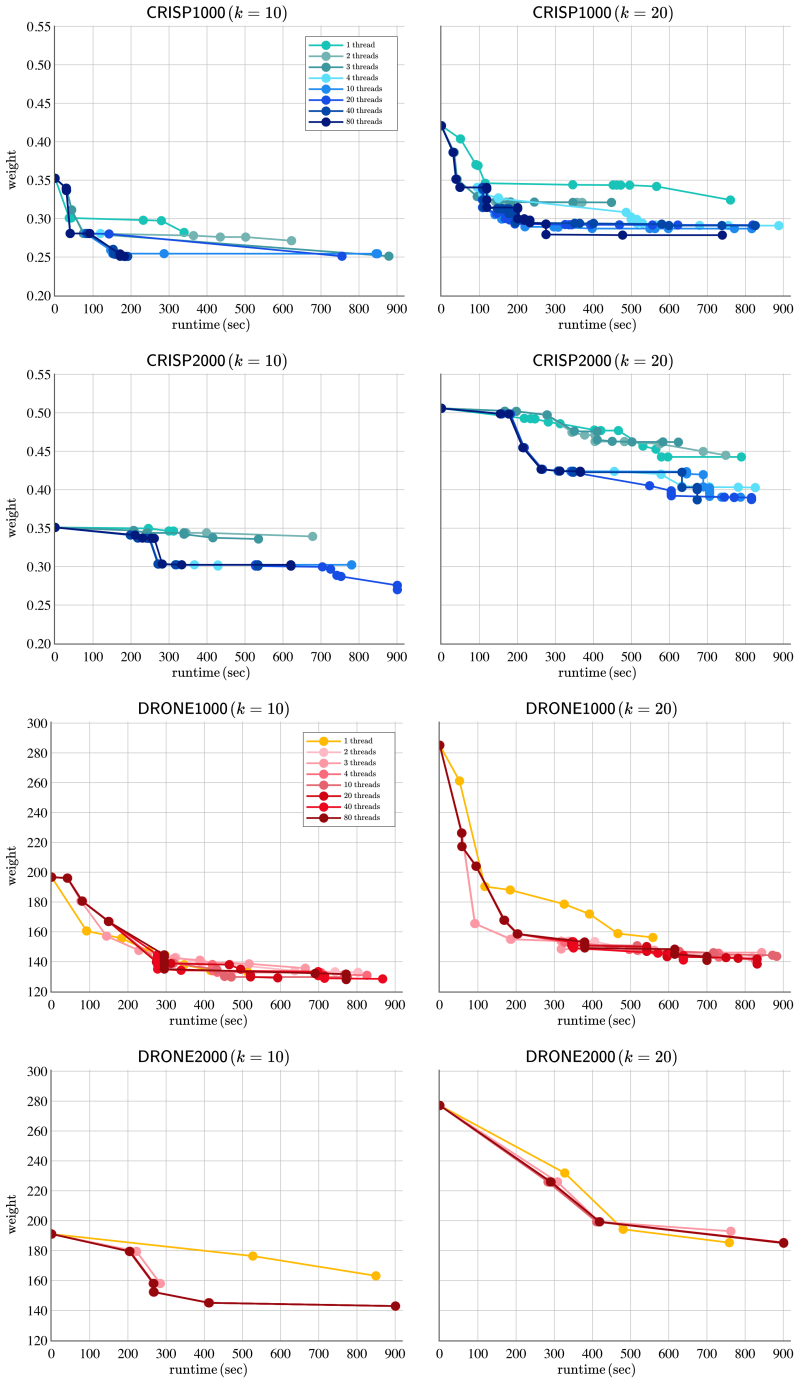


Figure 6.11: Results of multithreading experiments for ILP-IPA, executed on datasets CRISP (top two rows) and DRONE (bottom two rows), with $k = 10$ (left column) or $k = 20$ (right column).

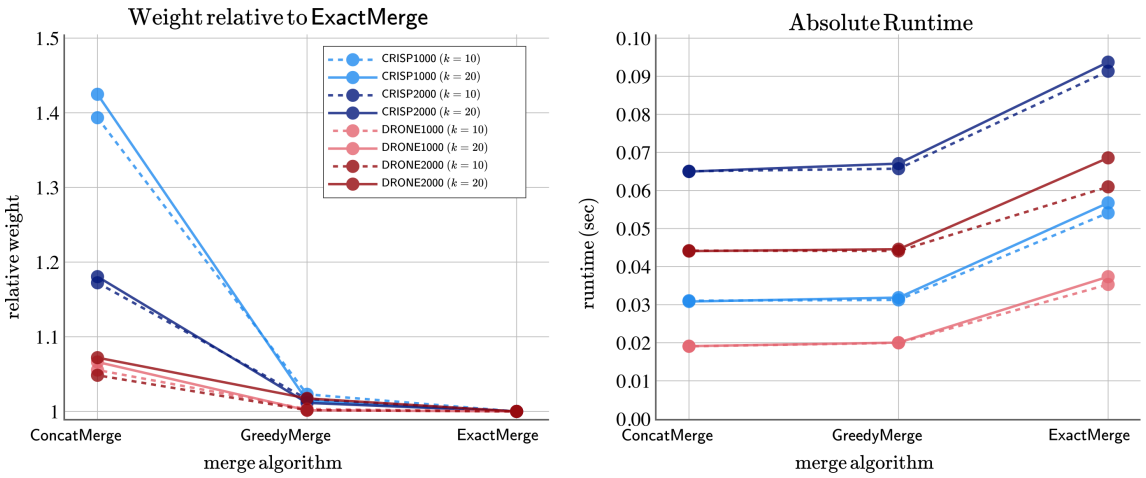


Figure 6.12: Experimental results comparing ConcatMerge, GreedyMerge, and ExactMerge. On the left, the weight of the combined walk is compared (relative to ExactMerge). On the right, runtimes are compared. In all experiments, three walks were combined.

Table 6.1: Corpus of test instances for GRAPH INSPECTION.

dataset		CRISP		DRONE	
n_{build}		1,000	2,000	1,000	2,000
name		CRISP1000	CRISP2000	DRONE1000	DRONE2000
number of vertices (n)		1,006	2,005	1,002	2,001
number of edges (m)		18,695	41,506	19,832	44,089
number of colors		4,200	4,200	3,204	3,254
number of colors at the starting vertex		535	540	10	10
number of colors at a vertex	min	0	0	0	0
	mean	183.39	175.71	22.67	19.16
	max	855	876	129	129
	stdev	179.02	170.75	24.61	22.41
edge weight	min	0.000002	0.000000	0.51	0.43
	mean	0.006971	0.005275	4.61	3.91
	max	0.060926	0.060926	18.51	18.51
	stdev	0.005354	0.004271	1.86	1.58
minimum spanning tree weight		1.109606	1.637212	1875.53	3118.65
diameter	unweighted	7	8	6	7
	weighted	0.136846	0.138467	48.24	49.73

Table 6.2: Comparison of solution weights generated by ILP-IPA, relative to the lowest weight produced by DP-IPA; both solvers had a 15-minute timeout.

		CRISP			DRONE		
n_{build}	k	mean	min	max	mean	min	max
1,000	10	1.03	1.00	1.16	1.04	1.00	1.13
	20	1.04	1.00	1.31	1.05	1.00	1.18
2,000	10	1.12	1.00	1.38	1.17	1.03	1.38
	20	1.16	1.01	1.34	1.19	1.02	1.51

REFERENCES

[1] P. L. ANDERSON, A. W. MAHONEY, AND R. J. WEBSTER, *Continuum reconfigurable parallel robots for surgery: Shape sensing and state estimation with uncertainty*, IEEE robotics and automation letters, 2 (2017), pp. 1617–1624.

[2] D. L. APPLEGATE, R. E. BIXBY, V. CHVATÁL, AND W. J. COOK, *The Traveling Salesman Problem: A Computational Study*, vol. 17, Princeton University Press, 2006.

[3] P. N. ATKAR, A. GREENFIELD, D. C. CONNER, H. CHOSET, AND A. A. RIZZI, *Uniform coverage of automotive surface patches*, The International Journal of Robotics Research, 24 (2005), pp. 883–898.

[4] A. BJÖRKLUND, T. HUSFELDT, AND N. TASLAMAN, *Shortest Cycle Through Specified Elements*, in Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Jan. 2012, pp. 1747–1753.

[5] P. CHENG, J. KELLER, AND V. KUMAR, *Time-optimal UAV trajectory planning for 3D urban structure coverage*, in 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2008, pp. 2750–2757.

[6] B. Y. CHO, T. HERMANS, AND A. KUNTZ, *Planning Sensing Sequences for Subsurface 3D Tumor Mapping*, in 2021 International Symposium on Medical Robotics (ISMR), Nov. 2021, pp. 1–7.

[7] B. Y. CHO AND A. KUNTZ, *Efficient and Accurate Mapping of Subsurface Anatomy via Online Trajectory Optimization for Robot Assisted Surgery*, in IEEE International Conference on Robotics and Automation (ICRA), May 2024, pp. 15478–15484.

[8] J. COHEN, G. F. ITALIANO, Y. MANOUSSAKIS, N. K. THANG, AND H. P. PHAM, *Tropical paths in vertex-colored graphs*, Journal of Combinatorial Optimization, 42 (2021), pp. 476–498.

[9] N. COHEN, *Several Graph problems and their Linear Program formulations*, research report, INRIA, 2019.

[10] B. COUËTOUX, E. NAKACHE, AND Y. VAXÈS, *The Maximum Labeled Path Problem*, Algorithmica, 78 (2017), pp. 298–318.

[11] V. DIMITRIJEVIĆ AND Z. ŠARIĆ, *An efficient transformation of the generalized traveling salesman problem into the traveling salesman problem on digraphs*, Information Sciences, 102 (1997), pp. 105–110.

[12] B. ENGLOT AND F. HOVER, *Planning complex inspection tasks using redundant roadmaps*, in Proceedings of The 15th International Symposium on Robotics Research (ISRR), Springer International Publishing, 2017, pp. 327–343.

[13] L. EULER, *Solutio problematis ad geometriam situs pertinentis*, Commentarii academiae scientiarum Petropolitanae, (1741), pp. 128–140.

[14] F. V. FOMIN, P. A. GOLOVACH, T. KORHONEN, K. SIMONOV, AND G. STAMOULIS, *Fixed-Parameter Tractability of Maximum Colored Path and Beyond*, in Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Society for Industrial and Applied Mathematics, Jan. 2023, pp. 3700–3712.

[15] M. FU, O. SALZMAN, AND R. ALTEROVITZ, *Computationally-efficient roadmap-based inspection planning via incremental lazy search*, in 2021 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2021, pp. 7449–7456.

[16] T. F. GONZALEZ, *Clustering to minimize the maximum intercluster distance*, Theoretical computer science, 38 (1985), pp. 293–306.

[17] D. HALPERIN, O. SALZMAN, AND M. SHARIR, *Algorithmic Motion Planning*, in Handbook of Discrete and Computational Geometry, Chapman and Hall/CRC, 3 ed., 2017.

[18] R. J. HARRIS, M. S. KAVURU, A. C. MEHTA, S. V. MEDENDORP, H. P. WIEDEMANN, T. J. KIRBY, AND T. W. BICE, *The impact of thoracoscopy on the management of pleural disease*, Chest, 107 (1995), pp. 845–852.

[19] C. HIERHOLZER AND C. WIENER, *Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren*, Mathematische Annalen, 6 (1873), pp. 30–32.

[20] S. KARAMAN AND E. FRAZZOLI, *Sampling-based algorithms for optimal motion planning*, The international journal of robotics research, 30 (2011), pp. 846–894.

[21] L. KOU, G. MARKOWSKY, AND L. BERMAN, *A fast algorithm for Steiner trees*, Acta informatica, 15 (1981), pp. 141–145.

[22] Y.-N. LIEN, E. MA, AND B. W.-S. WAH, *Transformation of the generalized traveling-salesman problem into the standard traveling-salesman problem*, Information Sciences, 74 (1993), pp. 177–189.

[23] R. W. LIGHT, *Pleural diseases*, Lippincott Williams & Wilkins, 2007.

[24] A. W. MAHONEY, P. L. ANDERSON, P. J. SWANEY, F. MALDONADO, AND R. J. WEBSTER, *Reconfigurable parallel continuum robots for incisionless surgery*, in 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, 2016, pp. 4330–4336.

[25] Y. MIZUTANI, D. C. SALOMAO, A. CRANE, M. BENTERT, P. G. DRANGE, F. REIDL, A. KUNTZ, AND B. D. SULLIVAN, *Leveraging fixed-parameter tractability for robot inspection planning*, arXiv preprint arXiv:2407.00251, 2024, (2024).

[26] M. NOPPEN, *The utility of thoracoscopy in the diagnosis and management of pleural disease*, in Seminars in respiratory and critical care medicine, vol. 31, 2010, pp. 751–759. tex.number: 06 tex.organization: Thieme Medical Publishers.

[27] P. C. POP, O. COSMA, C. SABO, AND C. P. SITAR, *A comprehensive survey on the generalized traveling salesman problem*, European Journal of Operational Research, 314 (2024), pp. 819–835.

[28] M. N. RICE AND V. J. TSOTRAS, *Exact graph search algorithms for generalized traveling salesman path problems*, in Experimental Algorithms, R. Klasing, ed., Berlin, Heidelberg, 2012, Springer Berlin Heidelberg, pp. 344–355.

[29] ——, *Parameterized algorithms for generalized traveling salesman problems in road networks*, in Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL'13, New York, NY, USA, Nov. 2013, Association for Computing Machinery, pp. 114–123.

[30] S. SAFRA AND O. SCHWARTZ, *On the complexity of approximating TSP with neighborhoods and related problems*, computational complexity, 14 (2006), pp. 281–307.

CHAPTER 7

ALGEBRAIC TECHNIQUES FOR INSPECTION PLANNING

In this chapter, we continue to study the Inspection Planning problem from robotics, but with a different approach. We investigate whether algorithms based on arithmetic circuits are a viable alternative to existing solvers for GRAPH INSPECTION. Specifically, we seek to address the high memory usage of existing solvers [14]. Aided by novel theoretical results enabling fast solution recovery, we implement a circuit-based solver for GRAPH INSPECTION, which uses only polynomial space, and test it on several realistic robotic motion planning datasets. While we demonstrate that circuit-based methods are not yet practically competitive for this application, our experimental evaluation of a suite of engineered algorithms for three key subroutines provides insights, which may guide future efforts to bring circuit-based algorithms from theory to practice.

This is joint work with Matthias Bentert[1], Daniel Coimbra Salomao[2], Alex Crane[2], Felix Reidl[3], and Blair D. Sullivan[2]. I am the lead author, designing and developing FPT, poly-space randomized algorithms using arithmetic circuits. I also designed computational experiments and wrote the majority of the manuscript. The latest version of our manuscript is available on arXiv [1].

In addition to the contents from the preprint, Section 7.3.3 details my exploratory work on characterizing a minimal circuit to reproduce a multilinear monomial, which we term as *certificate flow*.

---

[1]University of Bergen, Norway.

[2]University of Utah, USA.

[3]Birkbeck, University of London, UK.

7.1 Introduction

In GRAPH INSPECTION, we are given an edge-weighted, vertex-multi-colored graph and are asked to find a minimum-weight closed walk from a given starting vertex s that collects at least t colors. The colors allow us to model a "collection" problem, generalizing[4] the TRAVELING SALESMAN PROBLEM by allowing objects to be collectable at multiple vertices in the graph. GRAPH INSPECTION is motivated by robotic motion planning [5, 7], where a robot is tasked with inspecting "points of interest" by traveling a route in its configuration space. While exact solutions to this problem for a whole motion planning task may be prohibitively expensive to compute—given that the involved configuration space is immensely large and complicated—exact algorithms are nonetheless useful as Mizutani *et al.* [14] demonstrated by improving an existing planning heuristic [7] using integer linear programming (ILP) and dynamic programming (DP) exact solvers for GRAPH INSPECTION as subroutines. These approaches resulted in improved solution quality with comparable running times on simulated robotic tasks.

Mizutani *et al.* noted two limitations of the solvers which constrain their scalability. The ILP solver cannot solve problems on large networks[5] in part because the ILP itself scales with the number of edges in the network times the number of colors. In contrast, the DP solver runs in time linear in the network size but scales exponentially with the number of colors. Crucially, this is also true for the memory consumed by the DP and creates a sharp limit for its use case. The resulting need for a GRAPH INSPECTION solver which a) runs on large instances and b) has low space consumption was the starting point for our work here.

One theoretical remedy to dynamic programming algorithms with exponential space complexity are algebraic approaches which can "simulate" dynamic programming by constructing compact arithmetic circuits and testing the polynomials represented by these circuits for the presence of linear monomials [11, 12, 16, 8]. We can conceptualize this as a polynomial-time (and therefore polynomial-space) reduction from the source problem to

[4] Also of note is the GENERALIZED TRAVELING SALESMAN PROBLEM (GTSP) [15], for which solutions are simple paths rather than walks. Our results extend to GTSP restricted to complete graphs with edge-weights satisfying the triangle inequality.

[5] In their experiments, Mizutani *et al.* [14] were able to solve instances with roughly $2,000$ vertices and $40,000$ edges.

an instance of MULTILINEAR DETECTION, in which we are given an arithmetic circuit and are tasked with deciding whether a multilinear monomial exists. This latter problem is solvable using polynomial space (and running times comparable to the DP counterparts).

Algorithms based on MULTILINEAR DETECTION have been implemented for certain problems on synthetic or generic data with somewhat promising results [2, 10]. In this work, we took the opportunity to put the arithmetic circuit approach to the test in a very realistic setting: our test data is derived from two robotic inspection scenarios and our GRAPH INSPECTION solver can be used as a subroutine (similarly to Mizutani *et al.* [14]) to accomplish the relevant robotics tasks.

7.1.0.1 Theoretical contributions. Our main contribution is the introduction of *tree certificates* in the context of MULTILINEAR DETECTION. These objects allow us to circumvent an issue with existing results (see Section 7.3) and, more importantly, to directly recover a solution from the circuit without self-reduction. We present two randomized algorithms, one Monte-Carlo and one Las Vegas, which recover tree certificates for certain linear monomials of degree k in time $\tilde{\mathcal{O}}(2^k km)$ in circuits with m edges.

Next, in Section 7.4 we adapt the arithmetic circuit approach to reduce GRAPH INSPECTION parameterized by t to MULTILINEAR DETECTION on integer-weighted instances. Combining this with the aforementioned certificate recovery yields an FPT (defined in Section 7.2) algorithm which runs in $\tilde{\mathcal{O}}(2^t(\ell t^3 n^2 + t^3|\mathcal{C}|n))$ time and uses $\tilde{\mathcal{O}}(\ell t n^2 + t|C|n)$ space, where ℓ is an upper bound on the solution weight, $|\mathcal{C}|$ is the number of colors in the instance and t is the minimum number of colors we want to collect. One general challenge in using circuits is incorporating additional parameters like the solution weight ℓ and we explore several options to do so which might be of independent interest.

7.1.0.2 Engineering contributions. We present a C++ implementation of our theoretical algorithm with a modular design. This allows us to test different components of the algorithm, namely four different methods of circuit construction (Section 7.5), three different search strategies to find the optimal solution weight ℓ (Section 7.6), and both of the aforementioned solution recovery algorithms. Our engineering choices and experiments provide insights for several problems relevant for future implementations of arithmetic circuit algorithms, including (i) the avoidance of circuit reconstruction when searching for optimal solution weight, (ii) the discretization of non-integral weights, and (iii) the

importance of multithreading for this class of algorithms.

7.1.0.3 Experimental results. In Section 7.8 we experimentally evaluate each of the engineering choices described above, thereby demonstrating that careful engineering can yield significant running time improvements for arithmetic circuit algorithms. Additionally, we illustrate a trade-off between solution-quality and scalability by evaluating several scaling factors (used to produce integral edge weights). Finally, we demonstrate the large gap remaining between theory and practice: though the algebraic approach scales better in theory, it is not yet practically competitive with dynamic programming on instances with many nodes or more than $\sim$10 colors, which is too limiting for many practical applications.

7.1.0.4 Future directions: toward practicality. In this line of work, new theoretical insights and careful engineering yield progress toward the application of arithmetic-circuit-based techniques to a graph analysis problem with practical relevance in robotics. However, our experimental results demonstrate that further progress is needed before this class of algorithms becomes competitive with existing strategies in practice. From a theoretical perspective, new techniques which reduce the depth of constructed circuits, or new solving methods which reduce the dependence on that parameter, are desirable. Another interesting future direction would be hybrid methods that provide a trade-off between memory-intensive dynamic programming and time-intensive circuit evaluation. On the engineering side, the work by Kaski *et al.* [10] suggests that GPGPU implementations are feasible (albeit complicated) and the use of such hardware could provide a significant speedup for circuit-based algorithms.

7.2 Preliminaries

We refer to Section 1.3.1 and the textbook by Diestel [4] for standard graph-theoretic definitions and notation. Also, refer to Section 1.3.2 and the textbook by Cygan *et al.* [3] for parameterized complexity. Unless otherwise specified, all graphs $G = (V, E)$ in this work are undirected, with edges weighted by a function $w\colon E \to \mathbb{R}_{\geq 0}$ and vertices multi-colored by a function $\chi\colon V \to 2^{\mathcal{C}}$, where $\mathcal{C}$ is the *color set*. Given a vertex subset $S \subseteq V$, we use the notations $\chi(S)$ for $\bigcup_{v \in S} \chi(v)$. We can now define the subject of our study:

GRAPH INSPECTION

Input: An undirected graph $G = (V, E)$, a color set $\mathcal{C}$, an edge-weight function $w\colon E \to \mathbb{R}_{\geq 0}$, a vertex-coloring function $\chi\colon V \to 2^{\mathcal{C}}$, a vertex $s \in V$, and an integer t.

Problem: Find a minimum-weight closed walk $P = (v_0, v_1, \ldots, v_p)$ in G with $v_0 = v_p = s$ and $|\bigcup_{i=1}^{p} \chi(v_i)| \geq t$.

For the sake of simplicity, we may assume that G is connected, $t \leq |\mathcal{C}|$, and $\chi(s) = \emptyset$.

7.2.0.1 Polynomials and arithmetic circuits. A *monomial* over a set X of variables is a (commutative) product of variables from X. We call a monomial *multilinear* if no variable appears more than once in it. A polynomial is a linear combination of monomials with coefficients from $\mathbb{Z}_+$.

An *arithmetic circuit* $\mathcal{C}$ over X is a directed acyclic graph (DAG) for which every source is labelled either by a constant from $\mathbb{Z}_+$ (a *scalar*) or by a variable from X, and furthermore every internal node is labelled as either an *addition* or a *multiplication* node. The internal nodes and sinks of the DAG are called *gates* and *outputs*, respectively, of the circuit $\mathcal{C}$.

For a node $v \in \mathcal{C}$, we define $\mathcal{C}[v]$ to be the arithmetic induced by all nodes that can reach v in $\mathcal{C}$, including v. We further define $P_{\mathcal{C}[v]}(X)$ to be the polynomial that results from expanding the arithmetic expression of $\mathcal{C}[v]$ into a sum of products. For a circuit $\mathcal{C}$ with a single output $r \in V(\mathcal{C})$, we write $P_{\mathcal{C}}(X)$ for the polynomial $P_{\mathcal{C}[r]}(X)$.

We now formalize the problem of checking whether the polynomial representation of an output node includes a multilinear monomial.

MULTILINEAR DETECTION

Input: An arithmetic circuit $\mathcal{C}$ over a set X of variables and an integer k.

Problem: For each output node $r \in V(\mathcal{C})$, determine if $P_{\mathcal{C}[r]}(X)$ contains a multilinear monomial of degree at most k.

Prior work has established a randomized FPT algorithm which uses only polynomial space.

Lemma 7.1 ([12, 16])**.** *There is a randomized $\mathcal{O}^*(2^k)$-time and polynomial space algorithm for* MULTILINEAR DETECTION.

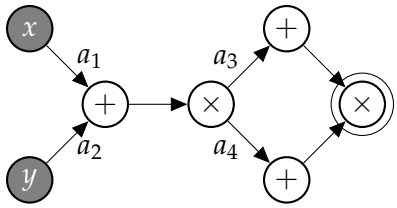
7.3 Engineering MULTILINEAR DETECTION

Before elaborating on our main algorithm, we characterize arithmetic circuits in terms of *certificates* and provide solution recovery algorithms.

Let $P_{\mathcal{C}}(X)$ be a polynomial represented by an (arithmetic) circuit $\mathcal{C}$, and let $P_{\mathcal{C}}(X, A)$ be a *fingerprint* polynomial derived from $\mathcal{C}$ as follows (as defined by Koutis and Williams): for every addition gate $v \in V(\mathcal{C})$ and input edge $uv \in E(\mathcal{C})$, we annotate the edge uv with a dedicated variable $a_{uv} \in A$. The semantic of this annotation is that the output of node u is multiplied by a_{uv} before it is fed to v. Koutis and Williams [12] showed that multilinear monomials can be detected using only polynomial space *if* the associated fingerprint polynomial has additional properties:

Lemma 7.2 ([12, 16])**.** *Let $\mathcal{C}$ be a connected arithmetic circuit over a set X of variables, and let k be an integer. Suppose that the coefficient of each monomial in the fingerprint polynomial $P_{\mathcal{C}}(X, A)$ is* 1*. Then, there exists a randomized $\mathcal{O}^*(2^k)$-time polynomial-space algorithm for* MULTILINEAR DETECTION *with $(\mathcal{C}, X, k)$. This is a one-sided error Monte Carlo algorithm with a constant success probability.*

To generalize this result to arbitrary circuits, Koutis and Williams defined $\mathcal{A}$-circuits[6] and claim that their associated fingerprint polynomial only contains monomials with coefficient one. However, our initial implementation showed that this claim does not hold, and in fact the following small fingerprint circuit is indeed an $\mathcal{A}$-circuit but does not have the claimed property:



The associated fingerprint polynomial is $P(X, A) = ((a_1x + a_2y) \cdot a_3)((a_1x + a_2y) \cdot a_4) = a_3a_4a_1^2x^2 + 2a_1a_2a_3a_4xy + a_3a_4a_1^2y^2$, in particular the one multilinear monomial $2a_1a_2a_3a_4xy$ has coefficient 2.

[6]These circuits have the properties that addition and multiplication gates alternate, addition gates have an out-degree of one, and that all scalar inputs are either 0 or 1.

In fact, the presence of multilinear monomials in the fingerprint polynomial is directly related to the presence of tree-shaped substructures in the circuit which the above circuit does not have. First, we need some machinery.

Given a circuit $\mathcal{C}$ with single output r such that $P_{\mathcal{C}}(X, A)$ contains a multilinear monomial, we define a *certificate* $\hat{\mathcal{F}}$ as a minimal sub-circuit of $\mathcal{C}$ with the same output node such that $P_{\hat{\mathcal{F}}}(X, A)$ contains a multilinear monomial, and every multiplication gate in $\hat{\mathcal{F}}$ takes the same inputs in $\mathcal{C}$. In other words, we require $N^-_{\hat{\mathcal{F}}}(v) = N^-_{\mathcal{C}}(v)$ for every multiplication gate $v \in V(\hat{\mathcal{F}})$. We say a certificate $\hat{\mathcal{F}}$ is a *tree certificate* if the underlying graph of $\hat{\mathcal{F}}$ is a tree. We can now state the lemma.

Lemma 7.3 (Tree Certificate Lemma)**.** *Let $\mathcal{C}$ be a circuit without scalar inputs. For a node $v \in V(\mathcal{C})$, the fingerprint polynomial $P_{\mathcal{C}[v]}(X, A)$ contains a multilinear monomial of coefficient* 1 *if and only if there exists a tree certificate for v.*

For example, it is clear to verify that the small circuit above does *not* contain a tree certificate since any certificate must include the cycle. The following lemma helps to show the existence of tree certificates by construction (proof in Section 7.3.1).

Lemma 7.4. *Let $\mathcal{C}$ be an arithmetic circuit. If every multiplication gate in $\mathcal{C}$ has at most* 1 non-variable *in-neighbor, then every certificate in $\mathcal{C}$ is a tree certificate.*

Unfortunately, for some of our circuit constructions, the condition imposed by Lemma 7.2 is too strong. However, we next prove that we solve MULTILINEAR DETECTION if *there exists* a multilinear monomial in $P_{\mathcal{C}}(X, A)$ with coefficient 1.

Lemma 7.5. *Let $\mathcal{C}$ be a connected arithmetic circuit over a set X of variables with m edges, and let k be an integer. Suppose* there exists *a multilinear monomial in the fingerprint polynomial $P_{\mathcal{C}}(X, A)$ whose coefficient is* 1 *if $P_{\mathcal{C}}(X)$ contains a multilinear monomial of degree at most k. Then, there exists a randomized $\tilde{\mathcal{O}}(2^k km)$-time $\tilde{\mathcal{O}}(m + k|X|)$-space algorithm for* MULTILINEAR DETECTION *with $(\mathcal{C}, X, k)$. This is a one-sided error Monte Carlo algorithm with a constant success probability (*1/4*).*

Proof. We use an algorithm by Koutis for ODD MULTILINEAR k-TERM [11], where we want to decide if the polynomial represented by an arithmetic circuit contains a multilinear

monomial with odd coefficient of degree at most k.

If $P_{\mathcal{C}}(X)$ does not contain a multilinear monomial of degree at most k, then the algorithm always decides correctly. Suppose $(\mathcal{C}, X, k)$ is a yes-instance. Then, by definition, if $P_{\mathcal{C}}(X, A)$ contains a multilinear monomial of coefficient 1, then $P_{\mathcal{C}}(X, A)$ contains a multilinear monomial with odd coefficient; notice that not *all* multilinear monomials have to have coefficient 1. The algorithm works in $\mathcal{O}(2^k(k|X| + T))$ time and $\mathcal{O}(k|X| + S)$ space, where T and S are the time and the space taken for evaluating the circuit over the integers modulo 2^{k+1}, respectively.

In our case, each node stores an $\mathcal{O}(\log k)$-size vector of integers up to 2^{k+1}, representing the coefficients of a polynomial with degree $\mathcal{O}(\log k)$. This requires $\tilde{\mathcal{O}}(k)$-bit information. The most time-consuming operation is multiplication of these vectors, which can be done in $\tilde{\mathcal{O}}(k)$ time with a Fast-Fourier-Transform style algorithm [16]. Hence, $T \in \tilde{\mathcal{O}}(km)$ and the running time is $\mathcal{O}(2^k(k|X| + T)) \subseteq \tilde{\mathcal{O}}(2^k(k|X| + km)) = \tilde{\mathcal{O}}(2^k km)$.

Storing the circuit and fingerprint polynomial requires $\tilde{\mathcal{O}}(m)$ space and each computation requires $\tilde{\mathcal{O}}(k)$ space at a time. Hence, $S \in \tilde{\mathcal{O}}(m + k)$ and $\mathcal{O}(k|X| + S) \subseteq \tilde{\mathcal{O}}(m + k|X|)$. □

7.3.1 Proofs on Tree Certificates

We start by stating the following property of a tree certificate.

Proposition 7.6. *Let $\hat{\mathcal{F}}$ be a tree certificate without scalar inputs. Then, for any node $v \in V(\hat{\mathcal{F}})$, we have that $P_{\hat{\mathcal{F}}[v]}(X, A)$ is a multilinear monomial (without any other terms) of coefficient 1. Moreover, every addition gate in $\hat{\mathcal{F}}$ has exactly one in-neighbor.*

Proof. We prove by induction on the number n of nodes in $\hat{\mathcal{F}}$ that $P_{\hat{\mathcal{F}}[v]}(X, A)$ contains exactly one monomial of coefficient 1 for all $v \in V(\hat{\mathcal{F}})$. For the base case with $n = 1$, the only node is a variable node, which is by definition a multilinear monomial of coefficient 1. For the inductive step, consider $P_{\hat{\mathcal{F}}[v]}(X, A)$ for an output node v. Notice that v must be either an addition gate or a multiplication gate.

If v is an addition gate, we know that there exists an in-neighbor of v whose fingerprint polynomial contains a multilinear monomial of coefficient 1. Due to the minimality of a certificate, v cannot have more than one in-neighbor. Let $u \in N^-_{\hat{\mathcal{F}}}(v)$ be v's in-neighbor.

Then, $\hat{\mathcal{F}}[u]$ is a tree certificate for u. From the inductive hypothesis, $P_{\hat{\mathcal{F}}[v]}(X, A) = a_{uv} \cdot P_{\hat{\mathcal{F}}[u]}(X, A)$, which is a multilinear monomial of coefficient 1.

If v is a multiplication gate, then for every in-neighbor u of v, $\hat{\mathcal{F}}[u]$ must be a tree certificate because otherwise $P_{\hat{\mathcal{F}}[v]}(X, A)$ cannot have a multilinear monomial. From the inductive hypothesis, $P_{\hat{\mathcal{F}}[u]}(X, A)$ is a multilinear monomial of coefficient 1 for every u. Notice that since the underlying graph of $\hat{\mathcal{F}}$ is a tree, for each distinct $u, u' \in N^-_{\hat{\mathcal{F}}}(v)$, $P_{\hat{\mathcal{F}}[u]}(X, A)$ and $P_{\hat{\mathcal{F}}[u']}(X, A)$ do not share any variables or fingerprints. Hence, we conclude that $P_{\hat{\mathcal{F}}[v]}(X, A) = \prod_{u \in N^-_{\hat{\mathcal{F}}}(v)} P_{\hat{\mathcal{F}}[u]}(X, A)$ is also a multilinear monomial of coefficient 1. □

Corollary 7.7. *Let $\hat{\mathcal{F}}$ be a tree certificate without scalar inputs. Then, for any node $v \in V(\hat{\mathcal{F}})$, $\hat{\mathcal{F}}[v]$ is also a tree certificate.*

Proof. From Proposition 7.6, for any node $v \in V(\hat{\mathcal{F}})$ we have that $P_{\hat{\mathcal{F}}[v]}(X, A)$ is a multilinear monomial of coefficient 1. Necessarily, $\hat{\mathcal{F}}[v]$ is a certificate as it should be minimal, and clearly the underlying graph of $\hat{\mathcal{F}}[v]$ is a tree. Thus $\hat{\mathcal{F}}[v]$ is a tree certificate. □

Here we prove Lemma 7.3.

Lemma 7.3 (Tree Certificate Lemma)**.** *Let $\mathcal{C}$ be a circuit without scalar inputs. For a node $v \in V(\mathcal{C})$, the fingerprint polynomial $P_{\mathcal{C}[v]}(X, A)$ contains a multilinear monomial of coefficient* 1 *if and only if there exists a tree certificate for v.*

To show this, we introduce new notation and then prove a stronger lemma. For a nonempty set of variables $\emptyset \neq X' \subseteq X$ and a (possibly empty) set of fingerprints $A' \subseteq A$, we write $\mu(X', A')$ for $(\prod_{x \in X'} x)\,(\prod_{a \in A'} a)$. By definition, $\mu(X', A')$ is a multilinear monomial of coefficient 1. Similarly, we write $\mu(X')$ for $\prod_{x \in X'} x$ in case fingerprints are irrelevant. Also, we say a tree certificate $\hat{\mathcal{F}}$ for $v \in V(\hat{\mathcal{F}})$ *encodes* (X', A') if $P_{\hat{\mathcal{F}}[v]}(X, A) = \mu(X', A')$.

Lemma 7.8. *Let $\mathcal{C}$ be a circuit without scalar inputs. For a node $v \in V(\mathcal{C})$, the fingerprint polynomial $P_{\mathcal{C}[v]}(X, A)$ contains a monomial $\mu(X', A')$ for some $\emptyset \neq X' \subseteq X$ and $A' \subseteq A$ if and only if there exists a tree certificate $\hat{\mathcal{F}}$ for v encoding (X', A').*

Proof. We will show the both directions of implications.

($\Rightarrow$) For a node $v \in V(\mathcal{C})$, assume that $P_{\mathcal{C}[v]}(X, A)$ contains a monomial $\mu(X', A')$ for some $\emptyset \neq X' \subseteq X$ and $A' \subseteq A$. We will show that $\mathcal{C}$ has a tree certificate $\hat{\mathcal{F}}$ for v encoding (X', A') by induction on the number n of nodes in $\mathcal{C}[v]$. It is trivial for the base case $n = 1$: since there are no scalar inputs, we have $n = 1$ if and only if v is a variable node representing $x \in X$. We have $P_{\mathcal{C}[v]}(X, A) = x = \mu(\{x\}, \emptyset)$, and $\mathcal{C}[v]$ encodes $(\{x\}, \emptyset)$. For the inductive step with $n > 1$, consider two cases.

Suppose node v is an addition gate. Then, there exists $u \in N_{\mathcal{C}}^{-}(v)$ such that $P_{\mathcal{C}[u]}(X, A)$ contains a monomial $\mu(X', A' \setminus \{a_{uv}\})$. From the inductive hypothesis, there exists a tree certificate $\hat{\mathcal{F}}$ for u encoding $(X', A' \setminus \{a_{uv}\})$. We see that the circuit $(V(\hat{\mathcal{F}}) \cup \{v\}, E(\hat{\mathcal{F}}) \cup \{uv\})$ forms a tree certificate for v encoding (X', A').

Suppose node v is a multiplication gate, and let $u_1, \ldots, u_\ell$ be the in-neighbors of v in $\mathcal{C}$. Since there are no scalar inputs, there must be a partition $(X_1, \ldots, X_\ell)$ of X' and a partition $(A_1, \ldots, A_\ell)$ of A' such that for each $1 \leq i \leq \ell$, $X_i \neq \emptyset$ and $P_{\mathcal{C}[u_i]}(X, A)$ contains a monomial $\mu(X_i, A_i)$. From the inductive hypothesis, there exists a tree certificate $\hat{\mathcal{F}}_i$ for u_i encoding (X_i, A_i) for each $1 \leq i \leq \ell$. If tree certificates $\hat{\mathcal{F}}_i$ are vertex-disjoint, then the circuit $(\{v\} \cup \bigcup_i V(\hat{\mathcal{F}}_i), \bigcup_i (E(\hat{\mathcal{F}}_i) \cup \{u_i v\}))$ forms a tree certificate for v encoding (X', A').

Now, assume towards a contradiction that tree certificates $\hat{\mathcal{F}}_1$ and $\hat{\mathcal{F}}_2$ (without loss of generality) share a node. Let $w \in V(\hat{\mathcal{F}}_1) \cap V(\hat{\mathcal{F}}_2)$ be a shared node first appearing in an arbitrary topological ordering of $(V(\hat{\mathcal{F}}_1) \cup V(\hat{\mathcal{F}}_2), E(\hat{\mathcal{F}}_1) \cup E(\hat{\mathcal{F}}_2))$. Observe that w cannot be a variable node because $X_1 \cap X_2 = \emptyset$.

From Corollary 7.7, $\hat{\mathcal{F}}_1[w]$ and $\hat{\mathcal{F}}_2[w]$ are tree certificates for w. Then, there exist some sets X_1', A_1', X_2', A_2' such that $\emptyset \neq X_1' \subseteq X_1$, $\emptyset \neq X_2' \subseteq X_2$, $A_1' \subseteq A_1$, $A_2' \subseteq A_2$, $P_{\hat{\mathcal{F}}_1[w]}(X, A) = \mu(X_1', A_1')$ and $P_{\hat{\mathcal{F}}_2[w]}(X, A) = \mu(X_2', A_2')$.

Consider the circuit $\hat{\mathcal{F}}_1'$ such that $V(\hat{\mathcal{F}}_1') = (V(\hat{\mathcal{F}}_1) \setminus V(\hat{\mathcal{F}}_1[w])) \cup V(\hat{\mathcal{F}}_2[w])$ and $E(\hat{\mathcal{F}}_1') = (E(\hat{\mathcal{F}}_1) \setminus E(\hat{\mathcal{F}}_1[w])) \cup E(\hat{\mathcal{F}}_2[w])$. Since w is the earliest "overlapping" node in the topological ordering, $V(\hat{\mathcal{F}}_1) \setminus V(\hat{\mathcal{F}}_1[w])$ and $V(\hat{\mathcal{F}}_2[w])$ do not share any nodes. Then, $\hat{\mathcal{F}}_1'$ is the tree certificate for u_1 encoding $((X_1 \setminus X_1') \cup X_2', (A_1 \setminus A_1') \cup A_2')$. Similarly, define $\hat{\mathcal{F}}_2' := (V(\hat{\mathcal{F}}_2) \setminus V(\hat{\mathcal{F}}_2[w]) \cup V(\hat{\mathcal{F}}_1[w]), E(\hat{\mathcal{F}}_2) \setminus E(\hat{\mathcal{F}}_2[w]) \cup E(\hat{\mathcal{F}}_1[w]))$. We know that $\hat{\mathcal{F}}_2'$ is the tree certificate for u_2 encoding $((X_2 \setminus X_2') \cup X_1', (A_2 \setminus A_2') \cup A_1')$.

For convenience, let $\tilde{X}_1 := (X_1 \setminus X_1') \cup X_2'$, $\tilde{A}_1 := (A_1 \setminus A_1') \cup A_2'$, $\tilde{X}_2 := (X_2 \setminus X_2') \cup X_1'$,

and $\tilde{A}_2 := (A_2 \setminus A_2') \cup A_1'$. Notice that by the inductive hypothesis, $P_{\mathcal{C}[u_1]}(X, A)$ contains monomials $\mu(X_1, A_1)$ and $\mu(\tilde{X}_1, \tilde{A}_1)$. Similarly, $P_{\mathcal{C}[u_2]}(X, A)$ contains monomials $\mu(X_2, A_2)$ and $\mu(\tilde{X}_2, \tilde{A}_2)$. Because v is a multiplication gate, we have

$$
\begin{aligned}
& P_{\mathcal{C}[v]}(X, A) \\
&= \prod_{i=1}^{\ell} P_{\mathcal{C}[u_i]}(X, A) \\
&= P_{\mathcal{C}[u_1]}(X, A) \cdot P_{\mathcal{C}[u_2]}(X, A) \cdot \prod_{i=3}^{\ell} P_{\mathcal{C}[u_i]}(X, A) \\
&= (\mu(X_1, A_1) + \mu(\tilde{X}_1, \tilde{A}_1) + \ldots) \cdot \\
&\quad (\mu(X_2, A_2) + \mu(\tilde{X}_2, \tilde{A}_2) + \ldots) \cdot \prod_{i=3}^{\ell} (\mu(X_i, A_i) + \ldots) \\
&= (\mu(X_1, A_1)\mu(X_2, A_2) + \mu(\tilde{X}_1, \tilde{A}_1)\mu(\tilde{X}_2, \tilde{A}_2) + \ldots) \cdot \\
&\quad \prod_{i=3}^{\ell} (\mu(X_i, A_i) + \ldots) \\
&= (\mu(X_1, A_1)\mu(X_2, A_2) + \mu(\tilde{X}_1, \tilde{A}_1)\mu(\tilde{X}_2, \tilde{A}_2)) \cdot \\
&\quad \prod_{i=3}^{\ell} \mu(X_i, A_i) + p(X, A),
\end{aligned}
$$

for some polynomial $p(X, A)$. This can be simplified as follows:

$$
\begin{aligned}
& \mu(\tilde{X}_1, \tilde{A}_1)\mu(\tilde{X}_2, \tilde{A}_2) \\
&= \mu(\tilde{X}_1 \cup \tilde{X}_2, \tilde{A}_1 \cup \tilde{A}_2) \\
&= \mu(((X_1 \setminus X_1') \cup X_2') \cup ((X_2 \setminus X_2') \cup X_1'), \\
&\quad ((A_1 \setminus A_1') \cup A_2') \cup ((A_2 \setminus A_2') \cup A_1')) \\
&= \mu(((X_1 \setminus X_1') \cup X_1') \cup ((X_2 \setminus X_2') \cup X_2'), \\
&\quad ((A_1 \setminus A_1') \cup A_1') \cup ((A_2 \setminus A_2') \cup A_2')) \\
&= \mu(X_1 \cup X_2, A_1 \cup A_2) \\
&= \mu(X_1, A_1)\mu(X_2, A_2)
\end{aligned}
$$

$$
\begin{aligned}
& P_{\mathcal{C}[v]}(X, A) \\
&= 2\mu(X_1, A_1)\mu(X_2, A_2) \cdot \prod_{i=3}^{\ell} \mu(X_i, A_i) + p(X, A) \\
&= 2 \cdot \prod_{i=1}^{\ell} \mu(X_i, A_i) + p(X, A) \\
&= 2\mu(\bigcup_{i=1}^{\ell} X_i, \bigcup_{i=1}^{\ell} A_i) + p(X, A) \\
&= 2\mu(X', A') + p(X, A)
\end{aligned}
$$

This result implies that $P_{\mathcal{C}[v]}(X, A)$ contains a monomial $\alpha\mu(X', A')$ for some integer $\alpha \geq 2$, contradicting our assumption that $P_{\mathcal{C}[v]}(X, A)$ contains $\mu(X', A')$ as a monomial.

($\Leftarrow$) Suppose that $\mathcal{C}$ has a tree certificate $\hat{\mathcal{F}}$ for $v \in V(\mathcal{C})$ encoding (X', A') for some $\emptyset \neq X' \subseteq X$ and $A' \subseteq A$. We will show that $P_{\mathcal{C}[v]}(X, A)$ contains the monomial $\mu(X', A')$.

By definition, we have $P_{\hat{\mathcal{F}}[v]}(X, A) = \mu(X', A')$. We iteratively construct $\mathcal{C}'$ as follows.

1. Initially, let $\mathcal{C}' \leftarrow (V(\mathcal{C}), E(\hat{\mathcal{F}}))$.
2. Add all the edges in $E(\mathcal{C}) \setminus E(\hat{\mathcal{F}})$ to $\mathcal{C}'$ that are not pointing to $V(\hat{\mathcal{F}})$. At this point, we still have $P_{\mathcal{C}'[v]}(X, A) = \mu(X', A')$.
3. Add an arbitrary edge $uw \in E(\mathcal{C}) \setminus E(\mathcal{C}')$ pointing to $w \in V(\hat{\mathcal{F}})$. Here, node w is an addition gate because otherwise edge uw must have been included in $\hat{\mathcal{F}}$. Notice that newly introduced terms in $P_{\mathcal{C}'[v]}(X, A)$ have a_{uw} as a factor. Recall that for every edge e such that $a_e \in A'$, we have $e \in E(\hat{\mathcal{F}})$. Since $uv \notin E(\hat{\mathcal{F}})$, $a_{uw} \notin A'$, and $P_{\mathcal{C}'[v]}(X, A)$ still have the monomial $\mu(X', A')$.
4. Repeat from Step 3 until we reach $\mathcal{C}' = \mathcal{C}$.

From the construction above, we have shown that $P_{\mathcal{C}[v]}(X, A)$ contains $\mu(X', A')$ as a monomial. □

The following proposition characterizes variable nodes in a certificate.

Proposition 7.9. *Let $\hat{\mathcal{F}}$ be a certificate for node v that is not a variable node. Then, every variable node in $\hat{\mathcal{F}}$ has out-degree* 1.

Proof. If there is a variable node x with out-degree 0, then x can be removed and $\hat{\mathcal{F}}$ is not minimal.

Assume towards a contradiction that a variable node x has out-degree at least 2. Then, $\hat{\mathcal{F}}$ contains a node u with two vertex-disjoint x-u paths. Since $\hat{\mathcal{F}}$ is minimal, multilinear monomials in $P_{\hat{\mathcal{F}}[v]}(X)$ require a monomial in $P_{\hat{\mathcal{F}}[u]}(X)$ using the two x-u paths P_1, P_2.

First, u cannot be a multiplication gate as the terms in $P_{\hat{\mathcal{F}}[u]}(X)$ using P_1 and P_2 contain x^2. Suppose u is an addition gate. Then, the terms in $P_{\hat{\mathcal{F}}[u]}(X)$ using P_1 and P_2 are in the form $\alpha_1 x + \alpha_2 x$ for some monomials α_1 and α_2. Then, having only one of the in-neighbors of u is sufficient, and thus $\hat{\mathcal{F}}$ is not minimal, a contradiction. □

Next, we prove Lemma 7.4.

Lemma 7.4. *Let $\mathcal{C}$ be an arithmetic circuit. If every multiplication gate in $\mathcal{C}$ has at most* 1 non-variable *in-neighbor, then every certificate in $\mathcal{C}$ is a tree certificate.*

Proof. Let $\hat{\mathcal{F}}$ be a certificate in $\mathcal{C}$. We prove by induction on the number n of nodes in $\hat{\mathcal{F}}$. It is clear for the base case $n = 1$. For the inductive step with $n > 1$, let v be an output node of $\hat{\mathcal{F}}$. We consider two cases (note that v cannot be a variable node).

Suppose node v is an addition gate. Then, it must have one in-neighbor u due to the minimality. First, we show that $\hat{\mathcal{F}}[u]$ is a certificate for u. Since $P_{\hat{\mathcal{F}}[v]}(X) = P_{\hat{\mathcal{F}}[u]}(X)$, we know that $P_{\hat{\mathcal{F}}[u]}(X)$ contains a multilinear monomial. Also for the same reason, if $\hat{\mathcal{F}}[u]$ is not minimal, then $\hat{\mathcal{F}}[v]$ is not minimal. Second, from the inductive hypothesis $\hat{\mathcal{F}}[u]$ is a tree certificate, and adding edge uv to $\hat{\mathcal{F}}[u]$ does not create a cycle in the underlying graph. Thus $\hat{\mathcal{F}}$ is a tree certificate.

Suppose node v is a multiplication gate. If v does not have a non-variable in-neighbor, then $\hat{\mathcal{F}}$ is clearly a tree certificate. Assume that the in-neighbors of v are variable nodes $X' \subseteq X$ and a non-variable node u. Since $P_{\hat{\mathcal{F}}[v]}(X) = \mu(X') \cdot P_{\hat{\mathcal{F}}[u]}(X)$, if $P_{\hat{\mathcal{F}}[v]}(X)$ contains a multilinear monomial $m_v(X)$, then $P_{\hat{\mathcal{F}}[u]}(X)$ contains a multilinear monomial $m_u(X) := m_v(X)/\mu(X')$. Here $m_u(X)$ may be scalar; recall that scalar terms are also considered multilinear.

First, we show that $\hat{\mathcal{F}}[u]$ is a certificate for u. As $P_{\hat{\mathcal{F}}[u]}(X)$ contains a multilinear monomial, it suffices to show that $\hat{\mathcal{F}}[u]$ is minimal. Assume not. Then, there exists a certificate $\hat{\mathcal{F}}'[u]$ as a sub-circuit of $\hat{\mathcal{F}}[u]$ such that $P_{\hat{\mathcal{F}}'[u]}(X)$ contains a multilinear monomial $m'_u(X)$. Now, the circuit $\hat{\mathcal{F}}'$ constructed from $\hat{\mathcal{F}}$ by replacing $\hat{\mathcal{F}}[u]$ with $\hat{\mathcal{F}}'[u]$ must have

the monomial $\mu(X') \cdot m'_u(X)$. From Proposition 7.9, $m'_u(X)$ does not contain any variables from X' because for each $x \in X'$, v is the only out-neighbor of x. Hence, $\mu(X') \cdot m'_u(X)$ is a multilinear monomial, contradicting that $\hat{\mathcal{F}}$ is minimal.

Second, from the inductive hypothesis $\hat{\mathcal{F}}[u]$ is a tree certificate. Also, from Proposition 7.9, $\hat{\mathcal{F}}[u]$ is disjoint from X'. Thus $\hat{\mathcal{F}}$ is a tree certificate. □

7.3.2 Solution Recovery for MULTILINEAR DETECTION

We say a fingerprint circuit $\mathcal{C}$ over (X, A) is *recoverable* with respect to an output node r and an integer k if it is verified that $P_{\mathcal{C}[r]}(X, A)$ contains a multilinear monomial of degree at most k with coefficient 1. Once we have determined that $\mathcal{C}$ is recoverable, we want to recover a solution by finding a tree certificate of $\mathcal{C}$ with the single output r.

Now we present two algorithms for finding a tree certificate: MonteCarloRecovery and LasVegasRecovery. The basic idea common in both algorithms is backtracking from the output node of a circuit. When seeing a multiplication gate, we keep all in-edges. For an addition gate, we use binary search to find exactly one in-edge that is included in a tree certificate. Specifically, for every addition gate v encountered during the solution recovery process, let E' be the set of in-edges of v, i.e., $E' = \{uv \colon u \in N^-_{\mathcal{C}}(v)\}$. Then, we say a partition (A, B) of E' is a *balanced partition of the in-edges of* v if $0 \leq |A| - |B| \leq 1$. We then run an algorithm for MULTILINEAR DETECTION with either $\mathcal{C} - A$ or $\mathcal{C} - B$ to decide which edges to keep.

Lemma 7.10 (MonteCarloRecovery)**.** *Let $\mathcal{C}$ be a connected recoverable circuit of m edges with respect to degree k and output r. Also assume that every tree certificate of $\mathcal{C}$ contains at most $\mathcal{O}(k)$ addition nodes. There exists a one-sided error Monte Carlo algorithm that finds a tree certificate of $\mathcal{C}$ in $\tilde{\mathcal{O}}(2^k km)$ time with a constant success probability.*

Proof. Consider Algorithm 10. This algorithm traverses all nodes in $\mathcal{C}$ from the given output node r to the variable nodes and removes nodes and edges that are not in a tree certificate. Whenever the algorithm sees an addition gate having a path to node r in the current circuit, it keeps exactly one in-neighbor. This operation is safe due to Proposition 7.6. Let $\hat{\mathcal{C}}$ be the resulting circuit. Every addition gate in $\hat{\mathcal{C}}$ has in-degree 1, and all the in-edges of a multiplication gate are kept if there is a path to node r in $\hat{\mathcal{C}}$. Assuming that

Algorithm 10: MonteCarloRecovery

Input: A recoverable circuit $\mathcal{C}$ with respect to k and output r, and a failure count threshold θ.

Output: A tree certificate.

```
   // Reversed traversal from the output node.
1  for v ∈ V(C) in topological ordering of the reverse graph of C do
2      if v ≠ r and N⁺(v) = ∅ then
           // Remove unlinked nodes.
3          Let C ← C − v.
4      else if v is an addition gate then
           // Perform binary search.
5          while deg⁻_C(v) > 1 do
6              Let A, B be a balanced partition of the in-edges of v.
7              Repeatedly solve MULTILINEAR DETECTION at most θ times with
                (C − A, k).
8              if C − A contains a tree certificate then
                   // Safe to remove A.
9                  Let C ← C − A.
10             else
                   // Should keep a vertex in A.
11                 Let C ← C − B.
12 return C
```

the algorithm solves MULTILINEAR DETECTION correctly, $\hat{\mathcal{C}}$ is a tree certificate.

The algorithm fails when Line 7 incorrectly concludes that $\mathcal{C} - A$ does not have a tree certificate. This happens with probability at most $(1-p)^\theta$, where p is a constant success probability of MULTILINEAR DETECTION (Lemma 7.5). By assumption, Algorithm 10 enters Line 5 no more than $\mathcal{O}(k)$ times. Let ck be this number. Then, the overall failure probability $f(\theta, k)$ is: $\sum_{i=0}^{ck-1}(1-(1-p)^\theta)^i \cdot (1-p)^\theta = (1-p)^\theta \cdot \frac{1-(1-(1-p)^\theta)^{ck}}{1-(1-(1-p)^\theta)} = 1-(1-(1-p)^\theta)^{ck}$. For a fixed success probability p' of the algorithm, we can find a value $\theta \in \mathcal{O}(\log k)$ such that $f(\theta, k) \leq p'$.

Finally, the expected running time of this algorithm is asymptotically bounded by the running time of Line 7 as other operations can be done in $\mathcal{O}(km)$ time. From Lemma 7.5, the total running time is $\mathcal{O}(k\theta \cdot 2^k m) = \mathcal{O}(2^k km \log k) = \tilde{\mathcal{O}}(2^k km)$ with a constant success probability. □

Lemma 7.11 (LasVegasRecovery)**.** *Let $\mathcal{C}$ be a connected recoverable circuit of m edges with respect to degree k and output r. Also assume that every tree certificate of $\mathcal{C}$ contains at most $\mathcal{O}(k)$ addition*

Algorithm 11: LasVegasRecovery

Input: A recoverable circuit $\mathcal{C}$ with respect to k and output r.
Output: A tree certificate.

```
   // Reversed traversal from the output node.
1  for v ∈ V(C) in topological ordering of the reverse graph of C do
2      if v ≠ r and N⁺(v) = ∅ then
           // Remove unlinked nodes.
3          Let C ← C − v.
4      else if v is an addition gate then
           // Perform binary search.
5          while deg⁻_C(v) > 1 do
6              Let A, B be a balanced partition of the in-edges of v.
               // Alternatively set A and B.
7              for X ∈ [A, B, A, B, ...] do
8                  Solve MULTILINEAR DETECTION with (C − X, k).
9                  if C − X contains a tree certificate then
                       // Safe to remove X.
10                     Let C ← C − X.
11                     break
12 return C
```

nodes. There exists a Las Vegas algorithm that finds a tree certificate of $\mathcal{C}$ with an expected running time of $\tilde{\mathcal{O}}(2^k km)$.

Proof. Consider Algorithm 11. This algorithm is identical to Algorithm 10 except for the inner loop starting from Line 5. Now, since Line 8 is a one-sided error Monte Carlo algorithm, if $\mathcal{C} - X$ contains a tree certificate, then there exists a tree certificate $\hat{\mathcal{F}}$ including some edge $uv \in A \cup B \setminus X$. The set X is safe to remove, and with the argument in the proof of Lemma 7.10, the algorithm correctly outputs a tree certificate of $\mathcal{C}$.

For the expected running time, note that by assumption, Algorithm 11 enters Line 5 no more than $\mathcal{O}(k)$ times. Again, the running time of the algorithm is asymptotically bounded by the running time of Line 8.

The expected number of executions of Line 8 is $\mathcal{O}(k \log n)$ because we perform binary search on the $\mathcal{O}(n)$ in-edges of an addition gate v. We know that the "correct" edge exists in either A or B, and the algorithm for MULTILINEAR DETECTION succeeds with a constant probability p. We expect to see one success for every $2/p$ runs. From Lemma 7.5, the total expected running time is $\tilde{\mathcal{O}}(k \log n \cdot \frac{2}{p} \cdot 2^k m) = \tilde{\mathcal{O}}(2^k km)$. □

7.3.3 Certificate Flow

This content is not included in the preprint.

Certificates help construct a minimal circuit to reproduce a multilinear monomial. But sometimes it is useful to track how a particular multilinear monomial is built. We therefore define a *certificate flow* to materialize this idea. In this section, we consider arithmetic circuits with scalar inputs, but any scalar input must represent a positive integer.

Let us recall $\mu(Y) := \prod_{y \in Y} y$, and by definition $\mu(\emptyset) = 1$. First, we show that the polynomial of any certificate contains only one multilinear monomial.

Lemma 7.12. *Let $\hat{\mathcal{F}}$ be a certificate for node v. Then, $P_{\hat{\mathcal{F}}[v]}(X)$ contains exactly one multilinear monomial.*

Proof. Suppose $P_{\hat{\mathcal{F}}[v]}(X)$ contains a multilinear monomial $\alpha_1 \mu(X')$ for some $X' \subseteq X$ and $\alpha_1 \in \mathbb{Z}_+$. Since $\hat{\mathcal{F}}$ is minimal, the set of the variable nodes in $\hat{\mathcal{F}}$ must form X'. Assume towards a contradiction that $P_{\hat{\mathcal{F}}[v]}(X)$ contains another multilinear monomial $\alpha_2 \mu(X'')$ for some $X' \neq X'' \subseteq X$ and $\alpha_2 \in \mathbb{Z}_+$. This implies that X'' is a proper subset of X'. Let x be a variable in $X' \setminus X''$. Since $\alpha_2 \mu(X'')$ does not contain x, there exists an edge wu for an addition gate u in an x-v path such that there are no paths from X'' to v using wu. Then, $P_{(\hat{\mathcal{F}}-wu)[v]}(X)$ also contains $\alpha_2 \mu(X'')$, contradicting that $\hat{\mathcal{F}}$ is minimal. □

The following is a straightforward consequence of Lemma 7.12.

Corollary 7.13. *Let $\hat{\mathcal{F}}$ be a certificate for node v. If $P_{\hat{\mathcal{F}}[v]}(X)$ contains a constant term, then $\hat{\mathcal{F}}$ does not contain any variable nodes.*

Before formalizing the certificate flow, we introduce several notations. Let $\mathcal{L}(X)$ be the set of multilinear monomials of coefficient 1 over X, that is, $\mathcal{L}(X) := \{\mu(Y) \mid Y \subseteq X\}$. We use the symbol $\prod$ in different contexts. For a collection C (a set, a vector, a tuple, etc.) of polynomials, we write $\prod C$ for $\prod_{c \in C} c$. For a set $\mathcal{C}$ of collections, we write $\prod \mathcal{C}$ for the Cartesian product of the sets in $\mathcal{C}$. Thus, $\prod \mathcal{C}$ is a set of tuples. We define $\mathrm{Mon}()$ as a function that takes as input a polynomial and outputs the set of monomials without coefficient. For example, $\mathrm{Mon}(2x^2 + xy + 3y + 9) = \{x^2, xy, y, 1\}$. For an arithmetic circuit, we say addition and multiplication gates except the output node are *internal nodes*.

Given a certificate $\hat{\mathcal{F}}$ for node v and variables X, a *certificate flow* on $\hat{\mathcal{F}}$ is a map $\phi : V(\hat{\mathcal{F}}) \cup E(\hat{\mathcal{F}}) \to 2^{\mathcal{L}(X)}$ satisfying the following conditions.

1. Sources: (1a) For every variable node x, $\phi(x) = \{x\}$. (1b) For every scalar input c, $\phi(c) = \{1\}$. We call $\{1\}$ the *constant flow*. Otherwise, we say a flow is *non-constant*.
2. Sink: (2) $\phi(v) = \{\mu(X)\}$.
3. Outgoing flow: For every node $u \in V(\hat{\mathcal{F}}) \setminus \{v\}$, let $N^+_{\hat{\mathcal{F}}}(u) = z_1, \dots, z_\ell$. (3a) If $\phi(u) = \{1\}$, then $\phi(uz_i) = \{1\}$ for all $1 \le i \le \ell$. (3b) Otherwise, $(\phi(uz_1), \dots, \phi(uz_\ell))$ is a partition of $\phi(u)$.
4. Incoming flow: For every node u except input nodes, let $N^-_{\hat{\mathcal{F}}}(u) = y_1, \dots, y_\ell$. (4a) If u is an addition gate, then $(\phi(y_1u), \dots, \phi(y_\ell u))$ is a partition of $\phi(u)$. If u is a multiplication gate, let $w_1, \dots, w_k$ be the in-neighbors of u such that $\phi(w_iu) \neq \{1\}$. (4b) If no such w_i exists, then $\phi(u) = \{1\}$. (4c) Otherwise, every $\phi(w_iu)$ has the same size d, and there exists $q_{ij} \in \mathcal{L}(X)$ for $1 \le i \le k$, $1 \le j \le d$ such that $\phi(w_iu) = \{q_{ij} \colon 1 \le j \le d\}$ and $\phi(u) = \{\prod_{i=1}^k q_{ij} \colon 1 \le j \le d\}$.

Intuitively, this can be seen as a "flow" from sources to the sink (v), where the amount of a flow is a set of multilinear monomials. We will later show that the "flow amount" $\phi(u) = \{q_1, q_2, \dots\} \subseteq \mathcal{L}(X)$ at node $u \in V(\hat{\mathcal{F}})$ means that $P_{\hat{\mathcal{F}}[u]}(X)$ contains a linear combination of $\{q_1, q_2, \dots\}$, or $\phi(u) \subseteq \mathrm{Mon}(P_{\hat{\mathcal{F}}[u]}(X))$. For a node or an edge z, we call the size of $\phi(z)$ the *width* of the flow. We also say a certificate flow has width k if the width of every flow amount is at most k. Figure 7.1 illustrates an example of a certificate flow.

We show basic properties of a certificate flow. For the rest of this section, we assume that $\hat{\mathcal{F}}$ is a certificate for node v, and ϕ is a certificate flow for $\hat{\mathcal{F}}$. We first observe that the set of variables is "conserved" at any node.

Observation 7.14. *For every internal node u, we have $\prod_{y \in N^-_{\hat{\mathcal{F}}}(u)} \prod \phi(yu) = \prod_{z \in N^+_{\hat{\mathcal{F}}}(u)} \prod \phi(uz)$.*

Proof. Let $y_1, y_2, \dots$ be the in-neighbors of u, and let $z_1, z_2, \dots$ be the out-neighbors of u. First, we will show that $\prod \phi(u) = \prod_{z \in N^+_{\hat{\mathcal{F}}}(u)} \prod \phi(uz)$. If $\phi(u)$ is the constant flow, i.e., $\phi(u) = \{1\}$, then from condition (3a), $\phi(uz) = \{1\}$ for each $z \in N^+_{\hat{\mathcal{F}}}(u)$. We have $\prod_{z \in N^+_{\hat{\mathcal{F}}}(u)} \prod \phi(uz) = 1 = \prod \phi(u)$. Otherwise, from (3b), $(\phi(uz_1), \phi(uz_2), \dots)$ is a parti-

tion of $\phi(u)$, and thus $\prod \phi(u) = \prod(\phi(uz_1), \phi(uz_2), \ldots) = (\prod \phi(uz_1)) \cdot (\prod \phi(uz_2)) \cdot \ldots = \prod_{z \in N^+_{\hat{\mathcal{F}}}(u)} \prod \phi(uz)$.

It remains to show that $\prod \phi(u) = \prod_{y \in N^-_{\hat{\mathcal{F}}}(u)} \prod \phi(yu)$. If u is an addition gate, then from (4a) $(\phi(y_1u), \phi(y_2u), \ldots)$ is a partition of $\phi(u)$. Hence, $\prod \phi(u) = \prod(\phi(y_1u), \phi(y_2u), \ldots) = (\prod \phi(y_1u)) \cdot (\prod \phi(y_2u)) \cdot \ldots = \prod_{y \in N^-_{\hat{\mathcal{F}}}(u)} \prod \phi(yu)$.

If u is a multiplication gate, let $w_1, w_2, \ldots$ be the in-neighbors of u such that $\phi(w_iu) \neq \{1\}$. If no such w_i exists, then $\phi(yu) = \{1\}$ for each $y \in N^-_{\hat{\mathcal{F}}}(u)$. In this case, we have $\prod_{y \in N^-_{\hat{\mathcal{F}}}(u)} \prod \phi(yu) = \prod_{y \in N^-_{\hat{\mathcal{F}}}(u)} 1 = 1$, and from (4b) $\phi(u) = 1$. Otherwise, from (4c) we have $\phi(u) = \{\prod_{i=1}^k q_{ij} \colon 1 \leq j \leq d\}$ for some q_{ij}. Then, $\prod \phi(u) = \prod_{j=1}^d \prod_{i=1}^k q_{ij} = \prod_{i=1}^k \prod_{j=1}^d q_{ij} = \prod_{i=1}^k \prod \phi(w_iu) = \prod_{y \in N^-_{\hat{\mathcal{F}}}(u)} \prod \phi(yu)$. □

For the sources and the sink, the following holds.

Observation 7.15. *We have* $\prod_{x \in X} \prod \phi(x) = \prod \phi(v)$.

Proof. From condition (1a), we have $\phi(x) = \{x\}$ for each $x \in X$. From condition (2), we have $\phi(v) = \{\mu(X)\}$. Hence, $\prod_{x \in X} \prod \phi(x) = \prod_{x \in X} x = \mu(X) = \prod \phi(v)$. □

These observations give us important insights; a certificate flow can be seen as a multi-commodity flow, where each commodity represents one variable. Specifically, each variable starts from the corresponding variable node and travels to the output node v. When the variable "enters" an internal node (whether an addition gate or a multiplication gate), it "chooses" one of the outgoing edges. Because of the preservation of variables (Observation 7.14), the same variable cannot use multiple out-edges. Furthermore, since $\hat{\mathcal{F}}$ is acyclic, the same variable cannot enter the same node twice. Hence, a variable $x \in X$ appears only along a certain x-v path. The fact that the incoming flow never shares the same variable ensures that the image of ϕ is indeed closed in $2^{\mathcal{L}(X)}$.

Now, we relate certificates to the certificate flow. First, we show that the flow amount at a node encodes information of the corresponding polynomial.

Proposition 7.16. *Given a certificate $\hat{\mathcal{F}}$ for node v on variables X and a certificate flow ϕ for $\hat{\mathcal{F}}$, for every node $u \in V(\hat{\mathcal{F}})$, we have $\phi(u) \subseteq \mathrm{Mon}(P_{\hat{\mathcal{F}}[u]}(X))$.*

Proof. We prove by induction on the number n of nodes in a sub-circuit $\hat{\mathcal{F}}[u]$ of $\hat{\mathcal{F}}$. For the

base case with $n = 1$, the only node u is either a variable node or a scalar input. If u is a variable node for x, then $\phi(u) = \{x\}$. We clearly have $\mathrm{Mon}(P_{\hat{\mathcal{F}}[u]}(X)) = \mathrm{Mon}(x) = \{x\} \supseteq \phi(u)$. If u is a scalar node, then $\phi(u) = \{1\}$ and $\mathrm{Mon}(P_{\hat{\mathcal{F}}[u]}(X)) = \{1\} \supseteq \phi(u)$.

For the inductive step, suppose the statement holds up to $n - 1$. Suppose u is an addition gate. By the condition of a certificate flow, $\phi(u) = \bigcup_{y \in N^-_{\hat{\mathcal{F}}}(u)} \phi(yu)$. From the inductive hypothesis, $\phi(yu) \subseteq \mathrm{Mon}(P_{\hat{\mathcal{F}}[y]}(X))$ for each $y \in N^-_{\hat{\mathcal{F}}}(u)$. Then, we have $\mathrm{Mon}(P_{\hat{\mathcal{F}}[u]}(X)) = \mathrm{Mon}(\sum_{y \in N^-_{\hat{\mathcal{F}}}(u)} P_{\hat{\mathcal{F}}[y]}(X)) = \bigcup_{y \in N^-_{\hat{\mathcal{F}}}(u)} \mathrm{Mon}(P_{\hat{\mathcal{F}}[y]}(X)) \supseteq \bigcup_{y \in N^-_{\hat{\mathcal{F}}}(u)} \phi(yu) = \phi(u)$.

Now, suppose u is a multiplication node. First, we observe that $\phi(u) \subseteq \{\prod \mathbf{q} \colon \mathbf{q} \in \prod_{y \in N^-_{\hat{\mathcal{F}}}(u)} \phi(yu)\}$. Recall that the first $\prod$ denotes the product of tuple elements, and the second denotes the Cartesian product of sets. This holds whether $\phi(u)$ is the constant flow or not. Second, by definition $\mathrm{Mon}(P_{\hat{\mathcal{F}}[u]}(X)) = \mathrm{Mon}(\prod_{y \in N^-_{\hat{\mathcal{F}}}(u)} P_{\hat{\mathcal{F}}[y]}(X)) = \{\prod \mathbf{q} \colon \mathbf{q} \in \prod_{y \in N^-_{\hat{\mathcal{F}}}(u)} \mathrm{Mon}(P_{\hat{\mathcal{F}}[y]}(X))\}$. Third, from the inductive hypothesis, for each $y \in N^-_{\hat{\mathcal{F}}}(u)$ we have $\phi(yu) \subseteq \mathrm{Mon}(P_{\hat{\mathcal{F}}[y]}(X))$. Putting these together, we have $\phi(u) \subseteq \{\prod \mathbf{q} \colon \mathbf{q} \in \prod_{y \in N^-_{\hat{\mathcal{F}}}(u)} \phi(yu)\} \subseteq \{\prod \mathbf{q} \colon \mathbf{q} \in \prod_{y \in N^-_{\hat{\mathcal{F}}}(u)} \mathrm{Mon}(P_{\hat{\mathcal{F}}[y]}(X))\} = \mathrm{Mon}(P_{\hat{\mathcal{F}}[u]}(X))$. □

Using this result, we can obtain several neat observations.

Observation 7.17. *Every flow amount in a certificate flow is nonempty.*

Proof. Consider from the sink, which has a nonempty flow by definition. If a node u is a multiplication gate, then its incoming flow is either the constant flow or has the same size as $\phi(u)$. Hence, every incoming flow is nonempty. If u is an addition gate, suppose there exists an in-neighbor w of u such that $\phi(wu) = \emptyset$. Then, from Proposition 7.16, $P_{(\hat{\mathcal{F}} - wu)[v]}(X)$ also contains a multilinear monomial, a contradiction to the minimality of $\hat{\mathcal{F}}$. Again, every incoming flow is nonempty. This then implies that all nodes and edges have a nonempty flow. □

The following observations characterize the scalar flow at addition gates.

Observation 7.18. *Let u be an addition gate such that $\phi(u) = \{1\}$. Then, u has only one in-neighbor.*

Proof. First, u must have at least one in-neighbor to be valid. Suppose u has at least two in-neighbors y_1, y_2. By construction, $(\phi(y_1u), \phi(y_2u), \ldots)$ is a partition of $\phi(u)$. But since

$\phi(u)$ contains only one element, either $\phi(y_1u)$ or $\phi(y_2u)$ must be empty, contradicting Observation 7.17. □

Observation 7.19. *Let u be an addition gate with an in-neighbor w such that $\phi(wu) = \{1\}$, Then, w is the only in-neighbor of u and hence $\phi(u) = \{1\}$.*

Proof. From Lemma 7.12, $P_{\hat{\mathcal{F}}[v]}(X)$ has exactly one multilinear monomial. Since $\hat{\mathcal{F}}$ is minimal, the multilinear monomial should appear in $(P_{\hat{\mathcal{F}}[u]}(X))^d \cdot q$ for some $d \in \mathbb{Z}_+$ and some multilinear monomial q.

Assume towards a contradiction that u has in-neighbors other than w. We consider removing the other in-edges of u from $\hat{\mathcal{F}}$. Define $\hat{\mathcal{F}}' := \hat{\mathcal{F}} - \left\{yu \colon y \in N^-_{\hat{\mathcal{F}}}(u) \setminus \{w\}\right\}$. Since $\phi(wu) = \{1\}$, $\phi(u)$ contains 1 as an element. Then, from Proposition 7.16, $P_{\hat{\mathcal{F}}'[u]}(X)$ contains some constant c. Now, $c^d q$ is a multilinear monomial that should appear in $P_{\hat{\mathcal{F}}'[v]}(X)$. Thus, $\hat{\mathcal{F}}$ is not minimal, a contradiction. Since w is the only in-neighbor of u, $\phi(u) = \phi(wu) = \{1\}$. □

Finally, we prove the existence of a certificate flow for any certificate. For simplicity, we first show how to construct a certificate flow for certificates without scalar inputs.

Lemma 7.20. *There exists a certificate flow ϕ for any certificate $\hat{\mathcal{F}}$ without scalar inputs.*

Proof. Consider constructing a certificate flow ϕ in the reversed topological ordering of $\hat{\mathcal{F}}$. We maintain the following properties during this process.

- (P1) For every entity $e \in V(\hat{\mathcal{F}}) \cup E(\hat{\mathcal{F}})$, $\phi(e)$ is a nonempty set of multilinear monomials without 1, consisting of distinct variables. For example, $\emptyset$, $\{1, x\}$ and $\{xy, yz\}$ are disallowed.
- (P2) For every processed node $u \in V(\hat{\mathcal{F}})$ that is not an input node, with processed in-neighbors, we have $\prod \phi(u) = \prod_{y \in N^-_{\hat{\mathcal{F}}}(u)} \prod \phi(yu)$.
- (P3) For every processed node $u \in V(\hat{\mathcal{F}})$ that is not the output node, we have $\prod \phi(u) = \prod_{z \in N^+_{\hat{\mathcal{F}}}(u)} \prod \phi(uz)$.
- (P4) For every processed node $u \in V(\hat{\mathcal{F}})$, $\phi(u) \subseteq \mathrm{Mon}(P_{\hat{\mathcal{F}}[u]}(X))$.
- (P5) For every processed edge $wu \in E(\hat{\mathcal{F}})$, $\phi(wu) \subseteq \mathrm{Mon}(P_{\hat{\mathcal{F}}[w]}(X))$.

- (P6) For every processed node u with processed in-edges, $\phi(u)$ and $\phi(yu)$ for each $y \in N^-_{\hat{\mathcal{F}}}(u)$ satisfy Condition (4a) or (4c) in the definition of a certificate flow.

(Base case) For the output node v, we set $\phi(v) = \{\mu(X)\}$, satisfying (P1) as $X \neq \emptyset$. Since $P_{\hat{\mathcal{F}}[v]}(X)$ contains a multilinear monomial $\alpha\mu(X)$ for some $\alpha \in \mathbb{Z}_+$, we have $\mu(X) \in \mathrm{Mon}(P_{\hat{\mathcal{F}}[v]}(X))$. Hence, $\phi(v) \subseteq \mathrm{Mon}(P_{\hat{\mathcal{F}}[v]}(X))$, which satisfies (P4).

(Processing in-edges) Once a node u has been processed, i.e., $\phi(u)$ is set, we examine the in-edges $y_1, \ldots, y_\ell$ of u. We will set $\phi(y_i u)$ for each $1 \leq i \leq \ell$ and prove Properties (P1), (P2), and (P5) hold.

Suppose u is an addition gate. Then, from $P_{\hat{\mathcal{F}}[u]}(X) = \sum_{y \in N^-_{\hat{\mathcal{F}}}(u)} P_{\hat{\mathcal{F}}[y]}(X)$, every monomial in $P_{\hat{\mathcal{F}}[u]}(X)$ must come from at least one in-neighbor. Formally, for all monomials $q \in \mathrm{Mon}(P_{\hat{\mathcal{F}}[u]}(X))$, there exists an in-neighbor $w_q \in N^-_{\hat{\mathcal{F}}}(u)$ such that $q \in \mathrm{Mon}(P_{\hat{\mathcal{F}}[y]}(X))$. Now, for each $q \in \phi(u)$, add q to $\phi(w_q u)$. Such an assignment exists because of Property (P4) $\phi(u) \subseteq \mathrm{Mon}(P_{\hat{\mathcal{F}}[u]}(X))$. This implies that $\phi(yu) \subseteq \mathrm{Mon}(P_{\hat{\mathcal{F}}[y]}(X))$ for each $y \in N^-_{\hat{\mathcal{F}}}(u)$, satisfying (P5) for yu. Notice that $(\phi(y_1 u), \ldots, \phi(y_\ell u))$ is a partition of $\phi(u)$, thus satisfying (P2) and (P6). From (P1) for u, $\phi(y_i u)$ contains distinct variables. Also, $\phi(y_i u)$ cannot be empty because if so, $P_{(\hat{\mathcal{F}} - y_i u)[v]}(X)$ also contains a multilinear monomial, violating the minimality of $\hat{\mathcal{F}}$. Hence, each edge $y_i u$ satisfies (P1).

Suppose u is a multiplication gate. Since we have $P_{\hat{\mathcal{F}}[u]}(X) = \prod_{y \in N^-_{\hat{\mathcal{F}}}(u)} P_{\hat{\mathcal{F}}[y]}(X)$, every monomial in $P_{\hat{\mathcal{F}}[u]}(X)$ is made up of the product of the monomials in $P_{\hat{\mathcal{F}}[y]}(X)$ for each in-neighbor $y \in N^-_{\hat{\mathcal{F}}}(u)$. Formally, $\mathrm{Mon}(P_{\hat{\mathcal{F}}[u]}(X)) = \{\prod \mathbf{q} \colon \mathbf{q} \in \prod_{y \in N^-_{\hat{\mathcal{F}}}(u)} \mathrm{Mon}(P_{\hat{\mathcal{F}}[y]}(X))\}$. Then for all monomials $q \in \mathrm{Mon}(P_{\hat{\mathcal{F}}[u]}(X))$, there must exist non-constant monomials $p_1(q), \ldots, p_\ell(q)$ such that $\prod_{i=1}^{\ell} p_i(q) = q$ and $p_i(q) \in \mathrm{Mon}(P_{\hat{\mathcal{F}}[y_i]}(X))$ for each $1 \leq i \leq \ell$. This is due to (P4) for u and the fact that $P_{\hat{\mathcal{F}}[y_i]}(X)$ does not contain any constants.

Let $\phi(u) = (q_1, \ldots, q_d)$. We set $\phi(y_i u) = \{p_i(q_1), \ldots, p_i(q_d)\}$ for each $1 \leq i \leq \ell$. From (P1) for u, all variables in $\phi(y_i u)$ are distinct. Furthermore, since each $p_i(q)$ partitions the variables in q, (P2) is also satisfied. Also, by construction $p_i(q_j) \in \mathrm{Mon}(P_{\hat{\mathcal{F}}[y_i]}(X))$ for each $1 \leq j \leq d$, and hence $\phi(y_i u) \subseteq \mathrm{Mon}(P_{\hat{\mathcal{F}}[y_i]}(X))$, satisfying (P5). Lastly, $|\phi(u)| = |\phi(y_i u)| = d$ for each i, and $\{\prod_{i=1}^{\ell} p_i(q_j) \colon 1 \leq j \leq d\} = \{q_j \colon 1 \leq j \leq d\} = \phi(u)$, which satisfies (P6).

(Processing nodes) When we process a node u, all the out-edges $z_1, z_2, \ldots$ should have been processed. We set $\phi(u) = \bigcup_{z \in N^+_{\hat{\mathcal{F}}}(u)} \phi(uz)$. It is clear to see that $\phi(u)$ satisfies (P3) by construction. From (P5) for all out-edges $z \in N^+_{\hat{\mathcal{F}}}(u)$, we have $\phi(uz) \subseteq \mathrm{Mon}(P_{\hat{\mathcal{F}}[u]}(X))$, and

(P4) $\phi(u) = \bigcup_{z \in N^+_{\hat{\mathcal{F}}}(u)} \phi(uz) \subseteq \mathrm{Mon}(P_{\hat{\mathcal{F}}[u]}(X))$ is also satisfied. So far, we have maintained (P2) and (P3), so we have $\prod_{y \in N^-_{\hat{\mathcal{F}}}(w)} \prod \phi(yw) = \prod_{z \in N^+_{\hat{\mathcal{F}}}(w)} \prod \phi(wz)$ for all processed nodes. Hence, the variables in $\{\phi(uz) \colon z \in N^+_{\hat{\mathcal{F}}}(u)\}$ are all distinct. Since $\phi(u)$ is clearly nonempty, we obtain Property (P1).

(Soundness of a flow) The previous arguments show that we can construct a flow ϕ with Properties (P1)-(P7). We verify that ϕ satisfies the conditions of a certificate flow. Conditions (1b) and (4b) are irrelevant here as $\hat{\mathcal{F}}$ does not contain scalar inputs.

Condition (1a): From (P4), for every variable node x, $\phi(x) \subseteq \mathrm{Mon}(P_{\hat{\mathcal{F}}[u]}(X)) = \{x\}$. From (P1), we have $\phi(x) = \{x\}$. Condition (2): By construction, $\phi(v) = \{\mu(X)\}$. Condition (3b): We know that $\phi(u) = \bigcup_{z \in N^+_{\hat{\mathcal{F}}}(u)} \phi(uz)$, and $1 \notin \phi(u)$. For $N^+_{\hat{\mathcal{F}}} = \{z_1, z_2, \ldots\}$, the variables in $\phi(uz_1), \phi(uz_2), \ldots$ are all distinct, and thus $(\phi(uz_1), \phi(uz_2), \ldots)$ is a partition of $\phi(u)$. Conditions (4a) and (4c) are directly implied from (P6). Hence, we have constructed a certificate flow ϕ for $\hat{\mathcal{F}}$. □

Constructing the scalar part of a certificate flow is straightforward.

Lemma 7.21. *There exists a certificate flow ϕ for any certificate $\hat{\mathcal{F}}$.*

Proof. If $P_{\hat{\mathcal{F}}[v]}(X)$ contains a constant term, then from Corollary 7.13 all the inputs are scalar inputs. We set $\phi(e) = \{1\}$ for all $e \in V(\hat{\mathcal{F}}) \cup E(\hat{\mathcal{F}})$.

Otherwise, let $S \subseteq V(\hat{\mathcal{F}})$ be the set of nodes unreachable from any variable nodes. We claim $\hat{\mathcal{F}}' := \hat{\mathcal{F}} - S$ is a certificate. Since $P_{\hat{\mathcal{F}}[v]}(X)$ contains a multilinear monomial with nonempty set of variables, $P_{\hat{\mathcal{F}}'[v]}(X)$ should also contain a multilinear monomial. And $\hat{\mathcal{F}}'$ should be minimal because otherwise $\hat{\mathcal{F}}$ is not minimal.

We construct a certificate flow ϕ' for $\hat{\mathcal{F}}'$ using Lemma 7.20. Then extend ϕ' to obtain ϕ by setting $\phi(e) = \{1\}$ for all $e \in S \cup (E(\hat{\mathcal{F}}) \setminus E(\hat{\mathcal{F}}'))$.

We claim that ϕ is a certificate flow for $\hat{\mathcal{F}}$. Condition (1a): Satisfied by ϕ'. Condition (1b): By construction $\phi(c) = \{1\}$ for each scalar input c. Condition (2): If $X = \emptyset$ (i.e., $P_{\hat{\mathcal{F}}[v]}(X)$ contains a constant term), then $\phi(v) = \{\phi(\emptyset)\} = \{1\}$. Otherwise, the condition is satisfied by ϕ'. Condition (3a): For every node $u \in V(\hat{\mathcal{F}}')$, $\phi(u) \neq \{1\}$. For a node $u \in S$, we have $\phi(u) = \{1\}$ and $\phi(uz) = \{1\}$ for all $z \in N^+_{\hat{\mathcal{F}}}(u)$. Condition (3b): Satisfied by ϕ'.

Condition (4a): Let u be an addition gate. From the minimality of a certificate, if

$u \in V(\hat{\mathcal{F}}')$, then u does not have an in-neighbor in S. The condition is satisfied by ϕ'. Otherwise, $u \in S$, and it should have exactly one in-neighbor w because having more than one scalar input is redundant. $(\phi(wu)) = (\{1\})$ is a partition of $\phi(u) = \{1\}$, as desired.

Condition (4b): This case happens only with a multiplication gate u in S. $\phi(u) = \{1\}$ by construction. Condition (4c): Now, this case is only with a multiplication gate u in $V(\hat{\mathcal{F}}')$ and satisfied by ϕ' as for any $y \in N^-_{\hat{\mathcal{F}}}(u) \cap S$, $\phi(yu) = \{1\}$. All conditions are satisfied. □

This lemma with Observation 7.17 directly implies Proposition 7.9, which we proved differently in Section 7.3.1. If a variable node has more than one out-neighbor, at least one flow amount must be empty to preserve the variable set, so this cannot happen.

One may ask, "given a certificate, can we find a certificate flow efficiently?" We answer this question affirmatively. First, we show that if we know the width of each flow amount, constructing a certificate flow is trivial. For simplicity, we also assume certificates do not contain scalar inputs as we can augment them later efficiently. We define the *flow profile* $\pi : E(\hat{\mathcal{F}}) \to \mathbb{Z}_{\geq 0}$ as a mapping intended to represent $|\phi(e)|$ for every edge $e \in E(\hat{\mathcal{F}})$. π is a *flow profile* if it satisfies the following conditions.

- (W1) For each variable node $x \in X$, $\pi(xz) = 1$ for all $z \in N^+_{\hat{\mathcal{F}}}(x)$.
- (W2) For each internal addition gate $u \in V(\hat{\mathcal{F}}) \setminus \{v\}$, we have $\sum_{y \in N^-_{\hat{\mathcal{F}}}(u)} \pi(yu) = \sum_{z \in N^+_{\hat{\mathcal{F}}}(u)} \pi(uz)$.
- (W3) For each internal multiplication gate $u \in V(\hat{\mathcal{F}}) \setminus \{v\}$, $\pi(yu) = \sum_{z \in N^+_{\hat{\mathcal{F}}}(u)} \pi(uz)$ for all $y \in N^-_{\hat{\mathcal{F}}}(x)$.
- (W4) For the output node v, $\pi(yv) = 1$ for all $y \in N^-_{\hat{\mathcal{F}}}(x)$.

Lemma 7.22. *Given a certificate $\hat{\mathcal{F}}$ for node v on variables X without scalar inputs. Let π be a flow profile on $\hat{\mathcal{F}}$. Then, there exists an $\mathcal{O}(|X| \cdot |E(\hat{\mathcal{F}})|)$-time algorithm to construct a certificate flow ϕ for $\hat{\mathcal{F}}$ such that $|\phi(e)| = \pi(e)$ for all $e \in E(\hat{\mathcal{F}})$.*

Proof. Consider the following algorithm. We process all nodes and their out-neighbors in the topological ordering of $\hat{\mathcal{F}}$ as follows.

- For every variable node x, set $\phi(x) = \{x\}$.

- For an addition gate u, set $\phi(u) = \bigcup_{y \in N^-_{\hat{\mathcal{F}}}(u)} \bigcup_{q \in \phi(yu)} q$.
- For a multiplication gate u, let $y_1, \ldots, y_\ell$ be the in-neighbors of u. By assumptions, we have $\phi(y_i u) \neq \{1\}$ for each i, and we know that from Condition (W3) there exists $d \in \mathbb{Z}_{\geq 0}$ such that $\pi(y_i u) = |\phi(y_i u)| = d$ for each $1 \leq i \leq \ell$. For each i, find $q_{ij} \in \mathcal{L}(X)$ such that $\phi(y_i u) = \{q_{ij} \colon 1 \leq j \leq d\}$ with some arbitrary ordering of $\phi(y_i u)$. Then, set $\phi(u) = \{\prod_{i=1}^{\ell} q_{ij} \colon 1 \leq j \leq d\}$.
- For any node u except the output node v, let $z_1, \ldots, z_\ell$ be the out-neighbors of u. Find an arbitrary partition $(Q_1, \ldots, Q_\ell)$ of $\phi(u)$ such that $\pi(uz_i) = |Q_i|$ for all $1 \leq i \leq \ell$. Set $\phi(uz_i) = Q_i$ for each i.

Notice that this construction with Conditions (W1)-(W3) satisfies Conditions (1a), (1b), (3a), (3b), (4b), and (4c) by default ((1b) and (4b) are not relevant as $\hat{\mathcal{F}}$ does not contain scalar inputs). We will show the remaining Conditions (2) and (4a) also hold. For Condition (4a), suppose towards a contradiction that there exists an addition gate $u \in V(\hat{\mathcal{F}})$ with in-neighbors $y_1, \ldots, y_\ell$ such that $(\phi(y_1 u), \ldots, \phi(y_\ell u))$ is not a partition of $\phi(u)$. Since $\bigcup_{i=1}^{\ell} \bigcup_{q \in \phi(y_i u)} q = \phi(u)$, there must be duplicates in $\{q \colon 1 \leq i \leq \ell,\ q \in \phi(y_i u)\}$. However, for each internal node, each variable flows to exactly one out-neighbor. And more than one copy of the same variable cannot enter the same node as $\hat{\mathcal{F}}$ is acyclic, a contradiction.

Hence, for each node, the set of variables is conserved as in Observation 7.14. Now, we know that $\prod \phi(v) = \mu(X)$. If v is an addition gate, then it must have only one in-neighbor due to the minimality of $\hat{\mathcal{F}}$. From (W4), regardless of the type of node v, we have $|\phi(v)| = 1$, leading to $\phi(v) = \{\mu(X)\}$, satisfying Condition (2).

The algorithm traverses all nodes in $\hat{\mathcal{F}}$, maintaining ϕ that requires $\mathcal{O}(|X| \cdot (|V(\hat{\mathcal{F}})| + |E(\hat{\mathcal{F}})|))$ space. The total running time is simplified to $\mathcal{O}(|X| \cdot |E(\hat{\mathcal{F}}|))$. □

One way to find a flow width $w_e \in \mathbb{Z}_+$ for all $e \in E(\hat{\mathcal{F}})$ is solving a system of linear Diophantine equations.

Corollary 7.23. *Given a certificate $\hat{\mathcal{F}}$ without scalar inputs. There exists a polynomial-time algorithm to find a flow width satisfying the flow width condition.*

Proof. We solve a system of linear Diophantine equations encoding the flow width condition (W1)-(W4). Let V_+ and $V_\times$ be the set of addition gates and the set of multiplication gates, respectively.

Note that from (W1), we may treat $\pi(xz)$ as a constant for all $x \in X$ for all $z \in N^+_{\hat{\mathcal{F}}}(x)$. From Proposition 7.9, there are $|X|$ such edges. Then, the system of equations for (W2)-(W4) consists of $(|E(\hat{\mathcal{F}})| - |X|)$ variables $\pi(e) \in \mathbb{Z}_+$ and $|V_+| + \sum_{u \in V_\times} \deg^-_{\hat{\mathcal{F}}}(u) \leq |E(\hat{\mathcal{F}})|$ constraints, whose coefficients are ± 1. Also, we have $\pi(e) \leq |X|$ due to Observation 7.14.

It is known that there is a polynomial-time algorithm to find a positive-integer solution to linear Diophantine equations [13]. Specifically, for a system of equations $A\mathbf{x} = \mathbf{b}$ for $A \in \mathbb{Z}^{m \times n}$ and $\mathbf{b} \in \mathbb{Z}^m$, we can find in polynomial time the Smith form $D \in \mathbb{Z}^{m \times n}$ such that $LAR = D$ for unimodular matrices[7] $L \in \mathbb{Z}^{m \times m}$, $R \in \mathbb{Z}^{n \times n}$. Then, we find $\mathbf{y} \in \mathbb{Z}^n$ such that $D\mathbf{y} = L\mathbf{b}$. Since D is a diagonal matrix, $\mathbf{y}$ can be found in polynomial time. Lastly, we set $\mathbf{x} = R\mathbf{y}$. It is clear to see that $A\mathbf{x} = AR\mathbf{y} = L^{-1}LAR\mathbf{y} = L^{-1}D\mathbf{y} = L^{-1}L\mathbf{b} = \mathbf{b}$.

In our case, if $\mathbf{x}$ has a unique solution, then it must be a non-negative integer solution. Otherwise, there are infinitely many solutions, and $\mathbf{y}$ contains at least one free variable. We know that there exists a solution $\mathbf{x}$ whose elements are between 0 and $|X|$, inclusive, and we can determine those values sequentially in polynomial time. □

So far, we have not found a certificate having different flow widths, thus leaving here the following question.

Open Question 7.24. *Given a certificate $\hat{\mathcal{F}}$ without scalar inputs. Is there always a* unique *flow profile?*

If the flow profile is always unique to the certificate, then we may just solve a system of linear equations to obtain the flow profile.

The following lemma gives an algorithm to find a certificate flow for a given certificate.

Lemma 7.25. *There exists a polynomial-time algorithm to construct a certificate flow ϕ for a given certificate $\hat{\mathcal{F}}$.*

Proof. Consider the following algorithm. First, we construct $\hat{\mathcal{F}}'$ by removing the constant

[7] A unimodular matrix is a square matrix with determinant ± 1.

flow. Let $S \subseteq V(\hat{\mathcal{F}})$ be the set of nodes unreachable from any variable nodes, and let $\hat{\mathcal{F}}' := \hat{\mathcal{F}} - S$. If $\hat{\mathcal{F}}'$ is empty, then output $\phi(e) = \{1\}$ for all $e \in V(\hat{\mathcal{F}}) \cup E(\hat{\mathcal{F}})$. This is correct because in that case, there are no variable nodes in $\hat{\mathcal{F}}$.

Otherwise, find the flow profile π for $\hat{\mathcal{F}}'$ by solving a system of linear Diophantine equations (Corollary 7.23). If $\hat{\mathcal{F}}'$ is a certificate for v, then there exists a certificate flow ϕ for $\hat{\mathcal{F}}'$ by Lemma 7.20, and ϕ should satisfy $|\phi(e)| = \pi(e)$ for all $e \in E(\hat{\mathcal{F}}')$. Once we find the variables $\pi(e)$ for all $e \in E(\hat{\mathcal{F}}')$, we can construct ϕ in polynomial time from Lemma 7.22. Finally, we augment ϕ by setting $\phi(e) = \{1\}$ for $e \in S \cup (E(\hat{\mathcal{F}}) \setminus E(\hat{\mathcal{F}}'))$ as proven by Lemma 7.21.

The correctness of this algorithm is due to Lemmas 7.20 to 7.22 and corollary 7.23. The overall running time is polynomial in $|E(\hat{\mathcal{F}})|$. □

In the beginning of Section 7.3, we introduced the notion of the tree certificate. The following lemma shows a relation between the width of a certificate flow and a tree certificate.

Lemma 7.26. *Let $\hat{\mathcal{F}}$ be a certificate without scalar inputs. A certificate flow ϕ has width* 1 *if and only if $\hat{\mathcal{F}}$ is a tree certificate.*

Proof. ($\Rightarrow$) If ϕ has width 1, then every node has at most one out-neighbor. The underlying graph of $\hat{\mathcal{F}}$ has to be a tree. Hence, $\hat{\mathcal{F}}$ is a tree certificate.

($\Leftarrow$) If $\hat{\mathcal{F}}$ is a tree certificate, then from Proposition 7.6, every addition gate in $\hat{\mathcal{F}}$ has in-degree 1. Then, any certificate flow ϕ for $\hat{\mathcal{F}}$ has width 1. □

We have shown that every certificate admits a certificate flow, but the converse is not true; a certificate does not guarantee the circuit being a certificate. Given a circuit and a corresponding certificate flow, when can we conclude that $\mathcal{F}$ is a certificate? One such characterization is the empty flow.

Observation 7.27. *Let $\mathcal{F}$ be a scalar-free circuit with an output node v. If there exists a certificate flow ϕ on $\mathcal{F}$ for v containing the empty flow at a node or an edge, then $\mathcal{F}$ is not a certificate.*

Proof. If a flow amount at a node is empty, then its incident edges also must be the empty flow. Hence, there exists an edge e such that $\phi(e) = \emptyset$. It is clear to see that $\mathcal{F} - e$ admits

the same flow without e, and then $P_{\mathcal{F}[v]}(X)$ contains a multilinear monomial. $\mathcal{F}$ is not a certificate as it is not minimal. □

Another characterization is the existence of a *bad weak cycle*, which we define here. A *bad weak cycle* C is a weak cycle[8] in $\mathcal{F}$ such that all the nodes in C except the sources of C (nodes with no incoming edges in the DAG C) are addition gates. Now, we have the following result.

Observation 7.28. *If a circuit $\mathcal{F}$ contains a bad weak cycle, then $\mathcal{F}$ is not a certificate.*

Proof. Let C be a bad weak cycle in $\mathcal{F}$ with nodes $u_1, \ldots, u_\ell$. First, observe that if C includes any variable nodes or the output node, then $\mathcal{F}$ is not a certificate. By the property of cycles, every node in C is incident to at least 2 edges. From Proposition 7.9, any variable node in a certificate cannot have more than one out-neighbor. Also, if C includes the output node v, then v must be an addition gate due to the definition of the bad weak cycle. A certificate for v should not have more than one in-neighbors.

Now, suppose towards a contradiction that $\mathcal{F}$ is a certificate. Then, there exists a flow profile $\pi : E \to \mathbb{Z}_{\geq 0}$ representing a certificate flow for $\mathcal{F}$. Without loss of generality, let $e_1 = u_1u_2$ such that $d := \pi(e_1)$ is minimized, and set the parity $p_1 = 1$. For each $2 \leq i \leq \ell$, if $u_iu_{i+1} \in E(\mathcal{F})$, then let $e_i = u_iu_{i+1}$ and $p_i = 1$. Otherwise, i.e., $u_{i+1}u_i \in E(\mathcal{F})$, let $e_i = u_{i+1}u_i$ and $p_i = -1$. For brevity, we set $u_{\ell+1} = u_1$, $e_0 = e_\ell$, and $p_0 = p_\ell$.

We create a new flow profile π' by setting $\pi'(e_i) = \pi(e_i) - p_i \cdot d$. We claim that π' is a valid flow profile. For $1 \leq i \leq \ell$ such that $p_{i-1} = p_i$, we know that u_i is an addition gate, and then $\pi'(e_{i-1}) - \pi(e_{i-1}) = \pi'(e_i) - \pi(e_i) = \pm d$; both incoming flow width and outgoing flow width either decrease or increase by d. So, the condition at u_i holds. Otherwise, $p_{i-1} \neq p_i$, and we have $\pi'(e_{i-1}) + \pi'(e_i) = \pi(e_{i-1}) + \pi(e_i)$. If u_i is the tail of e_{i-1} and e_i, then u_i is an addition gate, and the changes in incoming flow widths cancel out. If u_i is the head of e_{i-1} and e_i, then the changes in outgoing flow widths also cancel out.

Since $\pi'(e_1) = \pi(d_1) - d = 0$, there is a flow amount with the empty flow at edge e_1. By Observation 7.27, $\mathcal{F}$ is not a certificate, a contradiction. □

[8] A cycle if we ignore the directionality of the edges.

Remark 7.29. *We can extend this result to the case where a weak cycle includes multiplication gates if they have the same parity.*

Open Question 7.30. *Let $\mathcal{F}$ be a scalar-free circuit with an output node v. Identify all structures (or all obstructions) of $\mathcal{F}$ such that if there exists a certificate flow on $\mathcal{F}$ for v with no empty flow, then $\mathcal{F}$ is necessarily a certificate.*

To conclude this line of work, we propose another view of certificates. Our initial thought was to identify whether a circuit $\mathcal{F}$ is a certificate for node $v \in V(\mathcal{F})$ just by looking at the polynomial $P_{\mathcal{F}[v]}(X)$. Unfortunately, this is impossible. Consider a polynomial $(x+y)^2$. If the output is a multiplication gate merging two $(x+y)$ instances, then the circuit may be a certificate. However, the same polynomial can be built by gluing the terms x^2, y^2, $2xy$ by an addition gate. In this case, an edge from x^2, for example, to the addition gate is redundant, and the circuit cannot be a certificate.

The following is one way to define a class of functions to which polynomials evaluated on a certificate belong.

Definition 7.31. For a set of variables $X' \subseteq X$, a function $f(X)$ belongs to the class $\mathsf{Cert}[X']$ of functions defined recursively as follows.

- (O1) If $f(X) \in \mathbb{Z}_+$, then $f \in \mathsf{Cert}[\emptyset]$.
- (O2) If $f(X) = x$ for some $x \in X$, then $f \in \mathsf{Cert}[\{x\}]$.
- For some functions $f_1 \in \mathsf{Cert}[Y_1]$ and $f_2 \in \mathsf{Cert}[Y_2]$ for $Y_1, Y_2 \subseteq X$:
  - (O3) If $f(X) = f_1(X) + f_2(X)$, then $f \in \mathsf{Cert}[Y_1]$ and $f \in \mathsf{Cert}[Y_2]$.
  - (O4) If $f(X) = f_1(X) \cdot f_2(X)$ and $Y_1 \cap Y_2 = \emptyset$, then $f \in \mathsf{Cert}[Y_1 \cup Y_2]$.

Observation 7.32. *Let $f \in \mathsf{Cert}[Y]$ be a function for some $Y \subseteq X$. Then, $\mu(Y) \in \mathrm{Mon}(f(X))$.*

Proof. We prove by induction on the minimum number of operations to construct f. For the base case, with (O1) we have $Y = \emptyset$. Clearly, $\mu(Y) = 1 \in \mathrm{Mon}(f(X)) = \{1\}$, and there is no $Y' \subset Y$. With (O2), we have $Y = \{x\}$. $\mu(Y) = x \in \mathrm{Mon}(f(X)) = \{x\}$. If $Y' \subset Y$, then $Y' = \emptyset$. We see $\mu(Y') = 1 \notin \mathrm{Mon}(f(X))$.

For the inductive step, suppose f was constructed by (O3) $f(X) = f_1(X) + f_2(X)$ for some $f_1 \in \mathsf{Cert}[Y_1]$, $f_2 \in \mathsf{Cert}[Y_2]$. Without loss of generality, we may assume $Y = Y_1$. From the inductive hypothesis, we have $\mu(Y_1) \in \mathrm{Mon}(f_1(X))$. Hence, $\mu(Y) = \mu(Y_1) \in \mathrm{Mon}(f_1(X)) \subseteq \mathrm{Mon}(f(X))$.

Lastly, suppose f was constructed by (O4) $f(X) = f_1(X) \cdot f_2(X)$ for some $f_1 \in \mathsf{Cert}[Y_1]$, $f_2 \in \mathsf{Cert}[Y_2]$ such that (Y_1, Y_2) is a partition of Y. From the inductive hypothesis, we have $\mu(Y_1) \in \mathrm{Mon}(f_1(X))$ and $\mu(Y_2) \in \mathrm{Mon}(f_2(X))$. We know that $\mathrm{Mon}(f(X)) = \{q_1 q_2 \mid q_1 \in \mathrm{Mon}(f_1(X)), q_2 \in \mathrm{Mon}(f_2(X))\}$, and thus $\mu(Y_1) \cdot \mu(Y_2) \in \mathrm{Mon}(f(X))$. Since $Y_1 \cap Y_2 = \emptyset$, $\mu(Y_1) \cdot \mu(Y_2) = \mu(Y_1 \cup Y_2) = \mu(Y) \in \mathrm{Mon}(f(X))$. □

Lemma 7.33. *Let $\hat{\mathcal{F}}$ be a certificate for node v on variables X. Then, $P_{\hat{\mathcal{F}}[v]}(X)$ is in $\mathsf{Cert}[X]$.*

Proof. First, observe that there exists an "equivalent" certificate $\hat{\mathcal{F}}'$ for v such that $P_{\hat{\mathcal{F}}[v]}(X) = P_{\hat{\mathcal{F}}'[v]}(X)$ and every node in $\hat{\mathcal{F}}'$ has at most 2 in-neighbors. This can be constructed by adding extra addition and multiplication gates (e.g., $a + b + c$ can be expressed as $a' + c$ with a new addition gate $a' = a + b$).

Let ϕ be a certificate flow for $\hat{\mathcal{F}}'$. We show that for every node $u \in V(\hat{\mathcal{F}}')$ and for every multilinear monomial $q \in \phi(u)$, we have $P_{\hat{\mathcal{F}}'[u]}(X) \in \mathsf{Cert}[\mu^{-1}(q)]$. We prove by induction on the number n of the nodes in $\hat{\mathcal{F}}'[u]$ for a node u. For the base case with $n = 1$, if u is a scalar input, then $P_{\hat{\mathcal{F}}'[u]}(X) \in \mathbb{Z}_+$, and $\phi(u) = \{1\}$. From (O1), $P_{\hat{\mathcal{F}}'[u]}(X) \in \mathsf{Cert}[\emptyset] = \mathsf{Cert}[\mu^{-1}(1)]$. If u is a variable node for $x \in X$, then $P_{\hat{\mathcal{F}}'[u]}(X) = x$, and $\phi(u) = \{x\}$. From (O2), $P_{\hat{\mathcal{F}}'[u]}(X) \in \mathsf{Cert}[\{x\}] = \mathsf{Cert}[\mu^{-1}(x)]$.

For the inductive step, first consider the case where u has only one in-neighbor y. Then, $P_{\hat{\mathcal{F}}'[u]}(X) = P_{\hat{\mathcal{F}}'[y]}(X)$ and $\phi(u) \subseteq \phi(y)$, and from the inductive hypothesis, for all $q \in \phi(y)$, $P_{\hat{\mathcal{F}}'[y]}(X) \in \mathsf{Cert}[\mu^{-1}(q)]$. Clearly, for all $q \in \phi(u)$, we have $P_{\hat{\mathcal{F}}'[u]}(X) \in \mathsf{Cert}[\mu^{-1}(q)]$.

Now, consider u has two in-neighbors y_1 and y_2. Suppose u is an addition gate. Then, $P_{\hat{\mathcal{F}}'[u]}(X) = P_{\hat{\mathcal{F}}'[y_1]}(X) + P_{\hat{\mathcal{F}}'[y_2]}(X)$. Let $q \in \phi(u)$. We know that $\phi(u) = \phi(y_1 u) \cup \phi(y_2 u)$. Without loss of generality, we may assume $q \in \phi(y_1 u)$. Then, $q \in \phi(y_1)$ by construction. Similarly, since $\phi(y_2 u)$ is nonempty, there must exist $q' \in \phi(y_2 u) \subseteq \phi(y_2)$. From the inductive hypothesis, we have $P_{\hat{\mathcal{F}}'[y_1]}(X) \in \mathsf{Cert}[\mu^{-1}(q)]$ and $P_{\hat{\mathcal{F}}'[y_2]}(X) \in \mathsf{Cert}[\mu^{-1}(q')]$. From (O3), $P_{\hat{\mathcal{F}}'[u]}(X) \in \mathsf{Cert}[\mu^{-1}(q)]$.

Next, suppose u is a multiplication gate. Then, $P_{\hat{\mathcal{F}}'[u]}(X) = P_{\hat{\mathcal{F}}'[y_1]}(X) \cdot P_{\hat{\mathcal{F}}'[y_2]}(X)$. Let

$q \in \phi(u)$. We know that there exist multilinear monomials q_1 and q_2 such that $q = q_1 \cdot q_2$, $q_1 \in \phi(y_1 u) \subseteq \phi(y_1)$, and $q_2 \in \phi(y_2 u) \subseteq \phi(y_2)$. From the inductive hypothesis, we have $P_{\hat{\mathcal{F}}'[y_1]}(X) \in \mathsf{Cert}[\mu^{-1}(q_1)]$ and $P_{\hat{\mathcal{F}}'[y_2]}(X) \in \mathsf{Cert}[\mu^{-1}(q_2)]$. Notice that $\mu^{-1}(q_1) \cap \mu^{-1}(q_2) = \emptyset$. From (O4), $P_{\hat{\mathcal{F}}'[u]}(X) \in \mathsf{Cert}[\mu^{-1}(q_1) \cup \mu^{-1}(q_2)] = \mathsf{Cert}[\mu^{-1}(q_1 \cdot q_2)] = \mathsf{Cert}[\mu^{-1}(q)]$. □

7.4 Algorithm for GRAPH INSPECTION

The following describes a high-level algorithm ALG-IPA (ALGebraic Inspection Planning Algorithm) for GRAPH INSPECTION. There are three key subroutines: (1) circuit construction, (2) search, and (3) solution recovery. We designed and engineered several approaches to each, described in Sections 7.3, 7.5, and 7.6. This algorithm requires an input graph to be complete and metric and its edge weights to be integral.

- Algorithm ALG-IPA:

Input: A complete metric graph $G = (V, E)$, a color set $\mathcal{C}$, an edge-weight function $w\colon E \to \mathbb{Z}_{\geq 0}$, a vertex-coloring function $\chi\colon V \to 2^{\mathcal{C}}$, a vertex $s \in V$ such that $\chi(s) = \emptyset$, and a failure count threshold $\theta \in \mathbb{N}$.
Output: A minimum-weight closed walk in G, starting at s and collecting at least t colors.

(1) *Finding bounds.* Using the algorithms from [14], find lower (ℓ_{lo}) and upper bounds (ℓ_{hi}) for the solution weight. If the lower bound is fractional, then round it up to the nearest integer.

(2) *Search for the optimal weight.* Find the minimum weight $\tilde{\ell}$ such that there exists a closed walk from s with weight $\tilde{\ell}$, collecting at least t colors. We run the following steps iteratively.

   (a) *Construction of an arithmetic circuit.* As part of the search, construct an arithmetic circuit for a target weight ℓ ($\ell_{\mathrm{lo}} \leq \ell \leq \ell_{\mathrm{hi}}$).

   (b) *Evaluation of the arithmetic circuit.* Solve MULTILINEAR DETECTION for the constructed arithmetic circuit. If the output for ℓ contains a multilinear monomial, then we can immediately conclude that ℓ is feasible and update ℓ_{hi}. Otherwise, we repeatedly solve MULTILINEAR DETECTION for θ times until we conclude that ℓ is infeasible and update ℓ_{lo}.

(3) *Solution recovery.* Recover and output a solution walk with weight $\tilde{\ell}$. This can be done by reconstructing an arithmetic circuit for $\tilde{\ell}$, obtaining a tree certificate $\hat{\mathcal{F}}$ for it, and constructing a walk from vertex s in G based on $\hat{\mathcal{F}}$.

ALG-IPA can be used to solve any instance of GRAPH INSPECTION, given polynomial-time pre- and post-processing. The solution quality incurs a penalty based on rounding errors.

Let $\lambda \in \mathbb{R}$ be a *scaling factor*. We perform the following preprocessing steps in our implementation. Given a graph $G = (V, E)$, first create the transitive closure of G by computing all-pairs shortest paths. Remove all vertices $v \in V \setminus \{s\}$ that are unreachable from s or have no colors (i.e., $\chi(v) = \emptyset$). For every edge e, update its edge weight to $\lambda \cdot w(e)$ and round to the nearest integer[9]. Again, compute all-pairs shortest paths to make G a metric graph[10].

Once we obtain a solution walk W from ALG-IPA, simply replace every edge uv in W with any shortest u-v path in G. The resulting walk becomes a solution for GRAPH INSPECTION.

We prove later that ALG-IPA is a randomized FPT algorithm with respect to t, requiring only polynomial space. The algorithm's performance depends on the details of each subroutine (Sections 7.3, 7.5, and 7.6), and we defer a formal analysis of ALG-IPA to Section 7.7.

7.5 Circuit Construction

Here, we present four constructions for multilinear detection: NaiveCircuit, StandardCircuit, CompactCircuit, and SemiCompactCircuit. Each circuit consists of the following nodes:

- *Variables*: Variable node x_c for each color $c \in \mathcal{C}$.
- *Internal nodes*: We conceptually create t computational layers corresponding to the degree of a polynomial. Each layer contains two types of nodes: *transmitters* and *receivers*. A transmitter, denoted by $T_{t',v,d}$ or $T_{t',v,d,i}$, is a gate that transfers information to the next layer and is identified by layer $1 \leq t' \leq t$, vertex $v \in V \setminus \{s\}$, the weight d of a walk from s, and any optional index i.

[9]For accuracy, round weights on the transitive closure.

[10]This step is necessary because rounding may turn G into a non-metric graph.

A receiver, denoted by R_* (indices vary with construction types), is a gate that receives information from the previous layer and sends it to the transmitter in the same layer.

- *Output nodes*: The construction of output nodes (sinks that matter) depends on the search algorithm, but it is in common that they aggregate information from the transmitters in the last layer, i.e., layer t.

  If the search algorithm is UnifiedSearch (see Section 7.6.3), then there are addition nodes O_ℓ for every target value ℓ ($\ell_{\mathrm{lo}} \leq \ell \leq \ell_{\mathrm{hi}}$) as output nodes, and the edge from $T_{t,v,d}$ in layer t to O_ℓ exists if $\ell = d + w(v, s)$.

  If the search algorithm is either StandardBinarySearch or ProbabilisticBinarySearch (Sections 7.6.1 and 7.6.2), then a target weight ℓ is given when creating a circuit. In this case, an addition node O_ℓ is the only output node, and the edge from $T_{t,d,i}$ in layer t to O_ℓ exists if $\ell_{\mathrm{lo}} \leq d + w(v, s) \leq \ell$.

- *Auxiliary nodes*: CompactCircuit and SemiCompactCircuit (Sections 7.5.3 and 7.5.4) have another set of nodes located in between variable nodes and computational layers.

Observation 7.34. *There are $|\mathcal{C}|$ variable nodes and at most $(\ell_{\mathrm{hi}} - \ell_{\mathrm{lo}} + 1)$ output nodes for all constructions.*

Now we present how we construct computational layers. We then analyze the size of each circuit and its correctness. We argue that a construction is *correct* when the following conditions are met: (1) the circuit contains a tree certificate for MULTILINEAR DETECTION with degree at most t and the output node corresponding to objective ℓ if and only if there exists a solution walk with weight at most ℓ in an input graph for ALG-IPA, and (2) every tree certificate contains at most $\mathcal{O}(t)$ addition nodes. In the following arguments, we write k for $|\mathcal{C}|$, and set $w(v, v) = 0$.

7.5.1 NaiveCircuit

In this construction, transmitters are indexed by a pair of a vertex and one of its colors, that is, we split each color of a vertex into a distinct entity.

For the first layer, create a multiplication gate $T_{1,v,w(s,v),c}$ as a transmitter for every vertex v for every color $c \in \chi(v)$ if $w(s,v) \leq \ell$. Every transmitter in the first layer has only one input x_c.

In other layers $1 < t' \leq t$, proceed as follows. For every vertex v for every color $c \in \chi(v)$, and for every transmitter $T_{t'-1,v',d',c'}$ in the previous layer, let $d = d' + w(v',v)$. We continue only if $d \leq \ell$ and $c \neq c'$.

First, create a new multiplication gate r as a receiver taking $T_{t'-1,v',d',c'}$ and x_c as input. Next, create an addition gate $T_{t',v,d,c}$ as a transmitter if this does not exist. Finally, add an edge from r to the transmitter $T_{t',v,d,c}$. Notice that each receiver has 2 in-neighbors, and each transmitter may have at most $k(n-2)$ in-neighbors.

Lemma 7.35. NaiveCircuit *is correct and creates a circuit of* $\mathcal{O}(\ell_{\text{hi}} t k^2 n^2)$ *nodes and* $\mathcal{O}(\ell_{\text{hi}} t k^2 n^2)$ *edges.*

Proof. For each layer, there are $\mathcal{O}(\ell_{\text{hi}} kn)$ addition gates $T_{t',v,d,c}$ and for each of them, there are $\mathcal{O}(kn)$ multiplication gates r that link between layers. Hence, there are $\mathcal{O}(\ell_{\text{hi}} t k^2 n^2 + k + \ell_{\text{hi}}) = \mathcal{O}(\ell_{\text{hi}} t k^2 n^2)$ nodes in total. By construction, there are $\mathcal{O}(\ell_{\text{hi}} t k^2 n^2 + 2\ell_{\text{hi}} t k^2 n^2 + \ell_{\text{hi}} t k^2 n) = \mathcal{O}(\ell_{\text{hi}} t k^2 n^2)$ edges.

To show the correctness, suppose there is a solution walk $(s, v_1, v_2, \ldots, s)$ with weight ℓ. Then, there must be a corresponding sequence $(v_1, c_1), \ldots, (v_t, c_t)$ such that $\{v_i\}$ is an ordered (not necessarily distinct) vertex sets, and $\{c_i\}$ is a distinct set of colors with $c_i \in \chi(v_i)$. Such a distinct set of colors exists because the solution walk collects at least t colors. Let d_i be the distance from s to v_i in this walk, i.e., $d_1 = w(s, v_1), d_2 = d_1 + w(v_1, v_2), \ldots, d_i = d_{i-1} + w(v_{i-1}, v_i)$. Then, we construct a tree certificate as follows: pick T_{i,v_i,d_i,c_i} for every computational layer $1 \leq i \leq t$, and connect T_{t,v_t,d_t,c_t} to the output node O_ℓ. Observe that there is a path from T_{1,v_1,d_1,c_1} to O_ℓ including all T_{i,v_i,d_i,c_i}, because by assumption $w(s,v_1) + w(v_1,v_2) + \ldots + w(v_t,s) = d_t + w(v_t,s) = \ell$. We extend this path by adding all in-neighbors of any multiplication gates to construct $\hat{\mathcal{C}}$. This includes distinct t colors $\{c_i\}$, so $\hat{\mathcal{C}}$ is a tree certificate for MULTILINEAR DETECTION.

Now, suppose there exists a tree certificate $\hat{\mathcal{C}}$ for MULTILINEAR DETECTION. It is clear to see from construction that every monomial in $P_{\mathcal{C}[T_{t',v,d,c}]}(X)$ has degree t' and every monomial in $P_{\mathcal{C}[O_\ell]}(X)$ has degree t. Since the underlying graph of $\hat{\mathcal{C}}$ is a tree, $\hat{\mathcal{C}}$ includes

exactly one node T_{i,v_i,d_i,c_i} for each layer $1 \leq i \leq t$. For the same reason, the set $\{c_i\}$ is distinct. Also, we know that $\hat{\mathcal{C}}$ includes t addition gates, each of which has only one in-neighbor. Consider a walk $(s, v_1, v_2, \ldots, v_t, s)$. This walk collects at least t colors, and its total weight is ℓ. Hence, this is a solution walk. □

7.5.2 StandardCircuit

Intuitively, this type of circuit is built by switching addition and multiplication nodes in the computational layers of NaiveCircuit, as described in more detail below. The indexing scheme is also the same and the first layer is identical to the first layer of NaiveCircuit.

For layers $1 < t' \leq t$, proceed as follows. For each vertex v, for each color $c \in \chi(v)$, and for each transmitter $T_{t'-1,v',d',c'}$ in the previous layer, let $d = d' + w(v', v)$. We continue only if $d \leq \ell$ and $c \neq c'$.

First, create an addition gate $R_{t',v,d,c}$ as a receiver if it does not exist. Next, create a multiplication gate $T_{t',v,d,c}$ as a transmitter if it does not exist. Lastly, add the following edges: $(T_{t'-1,v',d',c'}, R_{t',v,d,c})$, $(R_{t',v,d,c}, T_{t',v,d,c})$, and $(x_c, T_{t',v,d,c})$. Note that in this construction, a transmitter has at most 2 in-neighbors, and a receiver has possibly $k(n-2)$ in-neighbors.

Lemma 7.36. StandardCircuit *is correct and creates a circuit of* $\mathcal{O}(\ell_{\text{hi}} tkn)$ *nodes and* $\mathcal{O}(\ell_{\text{hi}} tk^2n^2)$ *edges.*

Proof. For each layer, there are $\mathcal{O}(\ell_{\text{hi}} kn)$ addition gates with in-degree $\mathcal{O}(kn)$ and $\mathcal{O}(\ell_{\text{hi}} kn)$ multiplication gates with in-degree 2. There are $\mathcal{O}(\ell_{\text{hi}})$ output nodes with in-degree kn. Hence, in total there are $\mathcal{O}(\ell_{\text{hi}} tkn)$ nodes and $\mathcal{O}(\ell_{\text{hi}} tk^2n^2 + 2\ell_{\text{hi}} tkn + \ell_{\text{hi}} kn) = \mathcal{O}(\ell_{\text{hi}} tk^2n^2)$ edges.

For the correctness, if there is a solution walk $(s, v_i, v_2, \ldots, s)$ of weight ℓ, then there is a path including T_{i,v_i,d_i,c_i} and the output node O_ℓ, as defined in the proof of Lemma 7.35. This path and the set of collected colors $\{c_i\}$ induce a tree certificate.

If there is a tree certificate $\hat{\mathcal{C}}$, then $\hat{\mathcal{C}}$ must include t addition gates and T_{i,v_i,d_i,c_i} for each layer $1 \leq i \leq t$. The walk $(s, v_1, \ldots, v_t, s)$ will be a solution walk. □

7.5.3 CompactCircuit

This is designed to have an asymptotically smaller number of nodes than the others. In this construction, we do not split vertex colors. We have transmitters $T_{t',v,d}$, receivers

$R_{t',v,d}$, and auxiliary nodes $a_{t',v}$ for layer t', vertex $v \in V \setminus \{s\}$, and weight d.

First, construct auxiliary nodes. For each $1 \leq t' \leq t$ and for each vertex v, add an addition gate $a_{t',v}$ and take as input $\{x_c : c \in \chi(v)\}$.

For $t' = 1$, add an edge from a_v to $T_{1,v,w(s,v)}$ if $w(s,v) \leq \ell$. For $1 < t' \leq t$, for each vertex v, and for each transmitter $T_{t'-1,v',d'}$ in the previous layer, let $d = d' + w(v',v)$. Notice that it is possible that $v = v'$. Intuitively, this means collecting a new color without moving. We add the following edges if $d \leq \ell$: $(T_{t'-1,v',d'}, R_{t',v,d})$, $(R_{t',v,d}, T_{t',v,d})$, and $(a_{t',v}, T_{t',v,d})$. Figure 7.2 illustrates an example of this construction type.

Lemma 7.37. CompactCircuit *is correct and creates a circuit of* $\mathcal{O}(\ell_{\mathrm{hi}} tn + k)$ *nodes and* $\mathcal{O}(\ell_{\mathrm{hi}} tn^2 + tkn)$ *edges.*

Proof. In addition to k variable nodes, there are $\mathcal{O}(tn)$ addition nodes $a_{t',v}$, $\mathcal{O}(\ell_{\mathrm{hi}} tn)$ addition nodes $T_{t',v,d}$, $\mathcal{O}(\ell_{\mathrm{hi}} tn)$ multiplication nodes $R_{t',v,d}$, and $\mathcal{O}(\ell_{\mathrm{hi}})$ output nodes. The number of nodes is $\mathcal{O}(k + tn + 2\ell_{\mathrm{hi}} tn + \ell_{\mathrm{hi}}) = \mathcal{O}(\ell_{\mathrm{hi}} tn + k)$. To obtain the number of edges, we count in-degrees of those nodes. Each of $a_{t',v}$ has $\mathcal{O}(k)$ in-neighbors, each of $T_{t',v,d}$ has 2 in-neighbors, and each of $R_{t',v,d}$ has $\mathcal{O}(n)$ in-neighbors. For the output nodes, if the search strategy is UnifiedSearch, there are $\mathcal{O}(\ell_{\mathrm{hi}})$ nodes with in-degree $\mathcal{O}(n)$. Otherwise, there is 1 node with in-degree $\mathcal{O}(\ell_{\mathrm{hi}} n)$. In either case, there will be $\mathcal{O}(\ell_{\mathrm{hi}} n)$ edges to the output. The total number of edges is $\mathcal{O}(tkn + 2\ell_{\mathrm{hi}} tn + \ell_{\mathrm{hi}} tn^2 + \ell_{\mathrm{hi}} n) = \mathcal{O}(\ell_{\mathrm{hi}} tn^2 + tkn)$.

To show the correctness, suppose there is a solution walk of weight ℓ. Since the instance is complete and metric, there exists a solution walk $W = sv_1, \ldots, v_p, s$ of weight at most ℓ with no repeated vertices other than s such that at every vertex v in $V(W) \setminus \{s\}$ collects at least one new color. Then, we create a sequence $(v_1, c_1), \ldots, (v_t, c_t)$ as follows. First, pick exactly t colors $C \subseteq \bigcup_{v \in V(W)} \chi(v)$ so that we still collect at least one new color at every vertex in $V(W) \setminus \{s\}$. Let $C_v \subseteq C$ be the newly collected colors at vertex v. Then, when we see a new vertex v in $V(W) \setminus \{s\}$, append $\{(v,c) : c \in C_v\}$ to the sequence. Now, we have $\{v_i\}$ as an ordered (not necessarily distinct) vertex sets, and $\{c_i\}$ is a distinct set of colors with $c_i \in \chi(v_i)$. We write d_i for the distance from s to v_i in the walk W, i.e., $d_1 = w(s, v_1), d_2 = d_1 + w(v_1, v_2), \ldots$ with setting $w(v_i, v_i) = 0$. By assumption, we have $d_t + w(v_t, s) \leq \ell$.

We construct a tree certificate as follows. Let $S \subseteq V(\mathcal{C})$ be a set of nodes such that

$S = \{T_{i,v_i,d_i} : 1 \le i \le t\} \cup \{R_{i,v_i,d_i} : 1 < i \le t\} \cup \{a_{i,v_i} : 1 \le i \le t\} \cup \{O_{\ell'}\}$, where $\ell' = d_t + w(v_t, s) \le \ell$ for UnifiedSearch and $\ell' = \ell$ for the others. Let $\mathcal{C}' := \mathcal{C}[S]$ and observe that the underlying graph of $\mathcal{C}'$ is a tree. Then, we add variable nodes $\{c_i : 1 \le i \le t\}$ and edges $\{c_i a_{i,v_i} : 1 \le i \le t\}$ to $\mathcal{C}'$. It is clear to see that $\mathcal{C}'$ contains $2t$ addition gates. Also, its underlying graph remains a tree because c_i is distinct. By construction, $P_{\mathcal{C}'[O_{\ell'}]}(X)$ is a multilinear monomial representing the color set $\{c_i\}$ of size t. Observe that $\mathcal{C}'$ is a tree certificate for MULTILINEAR DETECTION.

Conversely, suppose there exists a tree certificate $\hat{\mathcal{C}}$ for MULTILINEAR DETECTION with respect to weight ℓ. Every multiplication gate in $\hat{\mathcal{C}}$ has the same in-neighbors as in $\mathcal{C}$, and from Proposition 7.6, every addition gate in $\hat{\mathcal{C}}$ has degree 1 in $\hat{\mathcal{C}}$. This leaves us one structure: t variable nodes $\{c_i\}$, their out-neighbors $\{a_{i,v_i}\}$ such that $c_i \in \chi(v_i)$, multiplication gates T_{i,v_i,d_i} in layers $1 \le i \le t$, accompanied addition gates R_{i,v_i,d_i} for $i > 1$, and the output node O_ℓ. Consider a walk $(s, v_1, v_2, \ldots, v_t, s)$. This walk collects t colors, and its total weight is at most ℓ. Hence, this is a solution walk. □

7.5.4 SemiCompactCircuit

This construction is similar to StandardCircuit, but instead of splitting vertex colors into vertex-color tuples, we keep track of vertex-multiplicity tuples. Here, the multiplicity means how many colors are collected at the same vertex.

First, add addition gates $a_v^{(i)}$ as auxiliary nodes that takes as input $\{x_c : c \in \chi(v)\}$ for every vertex $v \in V \setminus \{s\}$ and every multiplicity $1 \le i \le \min\{t, |\chi(v)|\}$.

The first layer contains multiplication gates $T_{1,v,d,1}$ as transmitters that takes $a_v^{(1)}$ as the only input. The other layers ($1 < t' \le t$) contain addition gates $R_{t',v,d,i}$ as receivers and multiplication gates $T_{t',v,d,i}$ as transmitters. $T_{t',v,d,i}$ takes two inputs, $R_{t',v,d,i}$ and $a_v^{(i)}$.

For every vertex $v \in V \setminus \{s\}$ add the following edges: (1) $(T_{t'-1,v,d,i}, R_{t',v,d,i+1})$ for $i < \min\{t, |\chi(v)|\}$, and (2) $(T_{t'-1,v',d',i}, R_{t',v,d'+w(v',v),1})$ for $v' \ne v$, $d' + w(v', v) \le \ell$, and $1 \le i \le \min\{t, |\chi(v')|\}$. The former represents collecting another color at the same vertex, thus keeping the same walk length and incrementing the multiplicity by one. The latter represents moving to another vertex, and the multiplicity is reset to one.

Lemma 7.38. SemiCompactCircuit *is correct and creates a circuit of* $\mathcal{O}(\ell_{\text{hi}} t^2 n + k)$ *nodes and* $\mathcal{O}(\ell_{\text{hi}} t^2 n^2 + tkn)$ *edges.*

Proof. We have $\mathcal{O}(tn)$ auxiliary nodes $a_v^{(i)}$ with in-degree $\mathcal{O}(k)$. The first layer contains $\mathcal{O}(n)$ nodes with in-degree 1. For each layer $1 < t' \leq t$, there are $\mathcal{O}(\ell_{\mathrm{hi}} n)$ addition gates $R_{t',v,d,1}$ with in-degree $\mathcal{O}(tn)$, $\mathcal{O}(\ell_{\mathrm{hi}} tn)$ addition gates $R_{t',v,d,i}$ with $i > 1$ and in-degree 1, $\mathcal{O}(\ell_{\mathrm{hi}} tn)$ multiplication gates with in-degree 2. There are $\mathcal{O}(\ell_{\mathrm{hi}})$ output nodes and $\mathcal{O}(\ell_{\mathrm{hi}} tn)$ edges to the output. Hence, there are $\mathcal{O}(\ell_{\mathrm{hi}} t^2 n + k)$ nodes and $\mathcal{O}(tkn + \ell_{\mathrm{hi}} t^2 n^2 + \ell_{\mathrm{hi}} t^2 n + 2\ell_{\mathrm{hi}} t^2 n + \ell_{\mathrm{hi}} tn) = \mathcal{O}(\ell_{\mathrm{hi}} t^2 n^2 + tkn)$ edges in total.

To show the correctness, suppose there is a solution walk of weight ℓ. Since there exists a solution walk $W = (s, v_1, \ldots, s)$ of weight at most ℓ with no repeated vertices other than s such that every vertex v in $V(W) \setminus \{s\}$ collects at least one new color. Then, we create a sequence $(v_1, c_1, \mu_1), \ldots, (v_t, c_t, \mu_2)$ as follows. First, pick exactly t colors $C \subseteq \bigcup_{v \in V(W)} \chi(v)$ so that we still collect at least one new color at every vertex in $V(W) \setminus \{s\}$. Let $C_v \subseteq C$ be the newly collected colors at vertex v. Then, when we see a new vertex v in $V(W) \setminus \{s\}$, append $(v, c_{v,1}, 1), (v, c_{v,2}, 2), \ldots$ to the sequence, where $C_v = \{c_{v,1}, c_{v,2}, \ldots\}$. Now, we have $\{v_i\}$ as an ordered (not necessarily distinct) vertex sets, $\{c_i\}$ is a distinct set of colors with $c_i \in \chi(v_i)$, and $\{\mu_i\}$ represents how many colors are collected at vertex v_i so far. We write d_i for the distance from s to v_i in the walk W, i.e., $d_1 = w(s, v_1), d_2 = d_1 + w(v_1, v_2), \ldots, d_i = d_{i-1} + w(v_{i-1}, v_i)$, with setting $w(v_i, v_i) = 0$. By assumption, we have $d_t + w(v_t, s) \leq \ell$.

We construct a tree certificate as follows. Let $S \subseteq V(\mathcal{C})$ be a set of nodes such that $S = \{T_{i,v_i,d_i,\mu_i} : 1 \leq i \leq t\} \cup \{R_{i,v_i,d_i,\mu_i} : 1 < i \leq t\} \cup \{a_{v_i}^{(\mu_i)} : 1 \leq i \leq t\} \cup \{O_{\ell'}\}$, where $\ell' = d_t + w(v_t, s) \leq \ell$ for UnifiedSearch and $\ell' = \ell$ for the others. Let $\mathcal{C}' := \mathcal{C}[S]$ and observe that the underlying graph of $\mathcal{C}'$ is a tree. By construction, the pair (v_i, μ_i) is unique in the sequence. Then, we add variable nodes $\{c_i : 1 \leq i \leq t\}$ and edges $\{c_i a_{v_i}^{\mu_i} : 1 \leq i \leq t\}$ to $\mathcal{C}'$. It is clear to see that $\mathcal{C}'$ contains $2t$ addition gates. Also, its underlying graph remains a tree because c_i is distinct. By construction, $P_{\mathcal{C}'[O_{\ell'}]}(X)$ is a multilinear monomial representing the color set $\{c_i\}$ of size t. $\mathcal{C}'$ is a tree certificate for MULTILINEAR DETECTION.

Conversely, suppose there exists a tree certificate $\hat{\mathcal{C}}$ for MULTILINEAR DETECTION with respect to weight ℓ. Every multiplication gate in $\hat{\mathcal{C}}$ has the same in-neighbors as in $\mathcal{C}$, and from Proposition 7.6, every addition gate in $\hat{\mathcal{C}}$ has degree 1 in $\hat{\mathcal{C}}$. This leaves us one structure: t variable nodes $\{c_i\}$, their out-neighbors $\{a_{v_i}^{(\mu_i)}\}$ such that $c_i \in \chi(v_i)$, multiplication gates T_{i,v_i,d_i,μ_i} in the computational layers $1 \leq i \leq t$, accompanied addition gates R_{i,v_i,d_i,μ_i} for $i > 1$, and the output node O_ℓ. Consider a walk $(s, v_1, v_2, \ldots, v_t, s)$. This

walk collects t colors, and its total weight is at most ℓ. This is a solution walk. □

7.6 Search Strategies

In this section, we describe three search algorithms. We prove that each algorithm correctly finds the optimal weight with probability $1-(1-p)^\theta$, where p is the constant success probability from Lemma 7.2, and θ is the failure count threshold greater than p^{-1}. Also, we analyze the expected running time under simplistic assumptions: the optimal weight $\tilde{\ell}$ is uniformly distributed between ℓ_{lo} and ℓ_{hi}; a single run of construction and evaluation of a circuit takes (non-decreasing) $f(\ell)$ time; and the evaluation results in True if $\tilde{\ell} \le \ell$ with probability p and False otherwise. We write $T(\ell_{\mathrm{lo}}, \ell_{\mathrm{hi}})$ for the expected running time with lower and upper bounds ℓ_{lo} and ℓ_{hi}, respectively. We also define $\ell_{\mathrm{diff}} := \ell_{\mathrm{hi}} - \ell_{\mathrm{lo}} + 1$.

7.6.1 StandardBinarySearch

We first implemented the standard binary search. Given ℓ_{lo} and ℓ_{hi}, we examine the middle value $\ell = \lfloor(\ell_{\mathrm{lo}} + \ell_{\mathrm{hi}})/2\rfloor$ to see if ℓ is feasible. If ℓ is feasible, then we update ℓ_{hi} to ℓ. On the other hand, if the circuit is evaluated only to False for θ' times, then we set ℓ_{lo} to $\ell+1$. Here θ' is a number in $\mathcal{O}(\theta \log\log(\ell_{\mathrm{diff}}))$ such that $(1-(1-p)^{\theta'})^{\lfloor \log_2(\ell_{\mathrm{diff}})\rfloor} \ge 1-(1-p)^\theta$; such a number must exist. The algorithm terminates when $\ell_{\mathrm{lo}} = \ell_{\mathrm{hi}}$, and this is our output.

Lemma 7.39. *Algorithm* StandardBinarySearch *correctly finds the optimal weight in expected running time* $\tilde{\mathcal{O}}(\theta \cdot f(\ell_{\mathrm{hi}}))$ *with probability* $1-(1-p)^\theta$.

Proof. We have $\theta' \in \tilde{\mathcal{O}}(\theta)$. It is known that binary search requires at most $\log_2(\ell_{\mathrm{diff}})$ evaluations, and each evaluation takes at most $\theta' f(\ell_{\mathrm{hi}})$ time. Hence, the running time is $\mathcal{O}(\theta' f(\ell_{\mathrm{hi}}) \log(\ell_{\mathrm{diff}})) \subseteq \tilde{\mathcal{O}}(\theta \cdot f(\ell_{\mathrm{hi}}))$.

The algorithm succeeds when all the $\log_2(\ell_{\mathrm{diff}})$-many evaluations succeed. This probability is

$$(1-(1-p)^{\theta'})^{\log_2(\ell_{\mathrm{diff}})} \ge 1-(1-p)^\theta$$

by our choice of θ'. □

7.6.2 ProbabilisticBinarySearch

Algorithm 12: ProbabilisticBinarySearch

Input: An instance $\mathcal{I}$ of GRAPH INSPECTION, bounds of the optimal weight $\ell_{\text{lo}} \leq \ell_{\text{hi}}$, a failure count threshold θ', and a probability p'.
Output: Optimal weight.

```
// Maintain midpoints as a stack.
Create an empty stack S.
while ℓ_lo < ℓ_hi do
    if S is empty then
        Push (⌊(ℓ_lo + ℓ_hi)/2⌋, 0) to S.
    Pop the top element (ℓ, c) from S.
    Create a circuit C of I for ℓ.
    Solve MULTILINEAR DETECTION with (C, k) and get result out.
    if out = True then
        Let ℓ_hi ← ℓ.
    else
        if c + 1 ≥ θ' then
            Let ℓ_lo ← ℓ + 1.                          // reject ℓ
            Clear S.
        else
            Push (ℓ, c + 1) to S.
            if ℓ < ℓ_hi − 1 then
                // Randomly go higher.
                With probability p', push (⌊(ℓ + 1 + ℓ_hi)/2⌋, 0) to S.
return ℓ_hi
```

The performance of the previous algorithm degrades when the optimal value is close to ℓ_{lo} because concluding that the value ℓ is infeasible requires θ evaluations of a circuit. To mitigate this penalty, ProbabilisticBinarySearch evaluates each circuit once at a time and randomly goes *higher* before concluding that the value is infeasible. We set this probability[11] p' to $1 - p/2$. We maintain midpoints and failure counts as a stack. Algorithm 12 gives the details.

Lemma 7.40. *Algorithm* ProbabilisticBinarySearch *correctly finds the optimal weight in expected running time* $\tilde{\mathcal{O}}((\theta + \ell_{\text{diff}}) \cdot f(\ell_{\text{hi}}))$ *with probability* $1 - (1 - p)^{\theta}$.

[11] This is the probability that the algorithm gives an incorrect output, assuming that the evaluated value is feasible with probability $1/2$.

Proof. The algorithm fails when for any feasible ℓ, it observes no successes and θ' failures. For a fixed ℓ, this probability is at most $(1-p)^{\theta'}$, and there are at most ℓ_{diff} possible values to check. By the same argument for StandardBinarySearch, by setting $\theta' \in \tilde{\mathcal{O}}(\theta)$ such that $(1-(1-p)^{\theta'})^{\ell_{\text{diff}}} \geq 1-(1-p)^{\theta}$, we can achieve success probability $1-(1-p)^{\theta}$.

To argue the running time, let $T(\tilde{\ell}, \ell)$ be the expected number of runs of an MULTI-LINEAR DETECTION solver for $\ell < \ell_{\text{hi}}$, where the optimal weight is $\tilde{\ell}$. This is sufficient as Algorithm 12 never evaluates ℓ_{hi}.

We consider three cases. If ℓ is feasible, that is, $\ell \geq \tilde{\ell}$, then unless the algorithm fails, it will eventually find that ℓ is feasible because whenever the algorithm search for a higher value, ℓ is always in the stack. Hence, $T(\tilde{\ell}, \ell) \leq p^{-1}$. Next, if $\ell = \tilde{\ell} - 1$, then the algorithm must try θ' evaluations to conclude that ℓ is infeasible. We have $T(\tilde{\ell}, \ell) = \theta'$. Lastly, if $\ell < \tilde{\ell} - 1$, the algorithm moves higher with probability p'. If it goes to another infeasible value (lucky case), it will never come back to ℓ. If it moves to a feasible value, it will then come back after finding that that value is feasible. Notice that the latter case only happens at most $\log_2(\ell_{\text{hi}} - \ell)$ times because when we come back from the higher part, the remaining search space will be shrunk into half. Hence, $T(\tilde{\ell}, \ell) \leq (p')^{-1} + \log_2(\ell_{\text{hi}} - \ell)$.

Putting these together, we sum over all possible $\tilde{\ell}, \ell$, assuming each of $\tilde{\ell}$ appears with probability ℓ_{diff}^{-1}. The expected running time is

$$\begin{aligned}
&\frac{f(\ell_{\text{hi}})}{\ell_{\text{diff}}} \sum_{\tilde{\ell}=\ell_{\text{lo}}}^{\ell_{\text{hi}}} \sum_{\ell=\ell_{\text{lo}}}^{\ell_{\text{hi}}-1} T(\tilde{\ell}, \ell) \\
\leq\ &\frac{f(\ell_{\text{hi}})}{\ell_{\text{diff}}} \left(\ell_{\text{diff}} \theta' + \ell_{\text{diff}}^2 (p^{-1} + (p')^{-1} + \log_2(\ell_{\text{diff}})) \right) \\
\in\ &\tilde{\mathcal{O}}((\theta + \ell_{\text{diff}}) \cdot f(\ell_{\text{hi}})),
\end{aligned}$$

as desired. Note that p and p' are constants. □

7.6.3 UnifiedSearch

The previous approach aims to reduce the number of evaluations for infeasible circuits, but in most cases, evaluating circuits with large ℓ (more time consuming than the evaluation of a circuit with small ℓ) multiple times is unavoidable.

We resolve this by tweaking a circuit to have multiple output nodes. Now, we assume that a circuit returns $\mathsf{out}(\ell) \in \{\mathsf{True}, \mathsf{False}\}$ for each $\ell_{\text{lo}} \leq \ell \leq \ell_{\text{hi}}$. As instructed in Sec-

tion 7.5 and illustrated in Figure 7.2, we create $(\ell_{\text{hi}} - \ell_{\text{lo}} + 1)$ output nodes connecting from the last layer of internal nodes. Other nodes are the same as the other search strategies.

The algorithm UnifiedSearch is shown in Algorithm 13. Once the output for ℓ is evaluated to True, it is safe to update $\ell_{\text{hi}} \leftarrow \ell$. We can then construct a smaller circuit for the new ℓ_{hi}, which may speed up the circuit evaluation.

Algorithm 13: UnifiedSearch

Input: An instance $\mathcal{I}$ of GRAPH INSPECTION, bounds of the optimal weight $\ell_{\text{lo}} \leq \ell_{\text{hi}}$, and a failure count threshold θ.
Output: Optimal weight.

```
for i ← 1 to θ do
    if ℓ_lo = ℓ_hi then
        return ℓ_hi
    Create a multi-output circuit C of I for all ℓ between ℓ_lo and ℓ_hi, inclusive.
    Solve MULTILINEAR DETECTION with (C, k) and obtain results
     out(ℓ) : ℓ ↦ {False, True}.
    Let ℓ_hi ← min{ℓ : out(ℓ) = True}.
return ℓ_hi
```

Lemma 7.41. *Algorithm* UnifiedSearch *correctly finds the optimal weight in expected running time* $\mathcal{O}(\theta \cdot f(\ell_{\text{hi}}))$ *with probability* $1 - (1-p)^{\theta}$.

Proof. Since there are at most θ circuit evaluations, the overall running time is bounded by $\theta \cdot f(\ell_{\text{hi}})$. Assuming the circuit construction is correct, we have $\text{out}(\ell) = \text{False}$ for every $\ell < \tilde{\ell}$. The algorithm fails only when $\text{out}(\tilde{\ell})$ is evaluated to False θ times consecutively. This happens with probability at most $(1-p)^{\theta}$, which completes the proof. □

7.7 Proof of Main Theorem

Now we are ready to formally state our main result.

Theorem 7.42. *If the edge weights are restricted to integers, and there exists a solution with weight at most* $\ell \in \mathbb{N}$, *then* GRAPH INSPECTION *can be solved in randomized* $\tilde{\mathcal{O}}(2^t(\ell t^3 n^2 + t^3|\mathcal{C}|n))$ *time and with* $\tilde{\mathcal{O}}(\ell t n^2 + t|\mathcal{C}|n)$ *space, with a constant success probability.*

Proof. Consider running ALG-IPA with CompactCircuit, UnifiedSearch, and MonteCarloRecovery with appropriate preprocessing. We set scaling factor $\lambda = 1$, assuming input weights

are integral, and let θ be a constant that controls the overall success probability. The preprocessing steps described in Section 7.4 take $\tilde{\mathcal{O}}(n^2)$ time to create a complete, metric graph. Step (1) is optional if we know an upper bound ℓ for the solution weight.

In step (2), as shown by Lemma 7.38, CompactCircuit creates a circuit $\mathcal{C}$ of $\mathcal{O}(\ell t^2 n + |\mathcal{C}|)$ nodes and $\mathcal{O}(\ell t n^2 + t|\mathcal{C}|n)$ edges such that if there exists a solution walk with weight at most ℓ in G, then $\mathcal{C}$ contains a tree certificate for MULTILINEAR DETECTION with degree at most t. This implies that for an output node $O_{\ell'}$ for every $\ell' \leq \ell$ the fingerprint polynomial $P_{\mathcal{C}[O_{\ell'}]}(X, A)$ contains a multilinear monomial with coefficient 1 if ℓ' is feasible. Note that $\mathcal{C}$ contains $\mathcal{O}(k)$ addition gates.

By Lemma 7.5 and Lemma 7.41, we can find the optimal weight $\tilde{\ell}$ in $\tilde{\mathcal{O}}(2^t t(\ell t n^2 + t|\mathcal{C}|n))$ time and $\tilde{\mathcal{O}}(\ell t n^2 + t|\mathcal{C}|n + t|\mathcal{C}|) = \tilde{\mathcal{O}}(\ell t n^2 + t|\mathcal{C}|n)$ space with a constant probability. Lastly, in Step (3) we use MonteCarloRecovery to find a solution walk with weight $\tilde{\ell}$. From Lemma 7.10, we can find a tree certificate of $\mathcal{C}$ in $\tilde{\mathcal{O}}(2^t t \cdot t(\ell t n^2 + t|\mathcal{C}|n)) = 2^t(\ell t^3 n^2 + t^3|\mathcal{C}|n)$ time with a constant probability. As stated in the proof of Lemma 7.37, once we find a tree certificate of $\mathcal{C}$, we can reconstruct a walk W collecting at least t colors with weight $\tilde{\ell}$ on G.

The postprocessing step is to replace each edge uv in W with any of the shortest u-v paths in the original graph for GRAPH INSPECTION to construct a solution walk $\tilde{W}$. Since G is complete and metric, the weight of $\tilde{W}$ is also $\tilde{\ell}$, and if the edge weight is restricted to (non-negative) integers, this weight is optimal in the original instance. Also, replacing edges in W with shortest paths in the original graph is safe because it is enough to collect colors at the vertices appeared in W.

The algorithm fails only when either UnifiedSearch or MonteCarloRecovery fails. Since both subroutines have a constant success probability, the overall success probability is also constant. □

7.8 Experimental Results

To show the practicality of our algebraic methods, we conducted computational experiments on the real-world instances used by Fu *et al.* [7, 6]. We built RRGs (Rapidly-exploring Random Graphs [9]) using IRIS-CLI, originating from the CRISP and DRONE datasets. For each dataset, we created small (roughly 50 vertex) and large (roughly 100 vertex) instances

and sampled $k \in \{10, 12\}$ dispersed POIs (Points Of Interest) using an algorithm from Mizutani *et al.* [14]. See Section 7.8.5 for details. Throughout the experiments, we set $t = k$, i.e., algorithms try to collect all colors (POIs) in the given graph and parallelized using 80 threads.

We first verify the effect of algorithmic choices—circuit types, search strategies, and solution recovery strategies—to see how they perform on various instances. We tested two scaling factors for each dataset: $\lambda_{\text{small}} = 50$, $\lambda_{\text{large}} = 100$ for CRISP and $\lambda_{\text{small}} = 0.1$, $\lambda_{\text{large}} = 0.5$ for DRONE. We measured running times by using the average measurement from three different seeds for a pseudorandom number generator. Next, we tested several scaling factors, as practical instances have real-valued edge weights, to observe trade-offs between running time/space usage and accuracy.

We implemented our code with C++ (using C++17 standard). We ran all experiments on Rocky Linux release 8.8 on identical hardware, equipped with 80 CPUs (Intel(R) Xeon(R) Gold 6230 CPU @ 2.10 GHz) and 191000 MB of memory.

Our code and data to replicate all experiments are available at `https://osf.io/4c92e/?view_only=e3d38d9356c04d60b32c2f45ccc19853`.

7.8.1 Choice of Circuit Types

To assess the four circuit type we proposed in Section 7.5, we ran our algorithm with all the circuit types, using the UnifiedSearch search strategy. We first measured the number of edges in the constructed arithmetic circuit as this number is a good estimator for running time and space usage.

Figure 7.3 (left) plots the number of edges of the circuit for each instance. By construction, StandardCircuit always gives a smaller circuit than NaiveCircuit. CompactCircuit has asymptotically the smallest circuit, but in some configurations, especially in DRONE, CompactCircuit results in a larger circuit than StandardBinarySearch. We observed that DRONE has a lower bound higher than that of CRISP, and CompactCircuit has to maintain low-weight walks that are "below" lower bounds, thus creating more edges. Lastly, SemiCompactCircuit has about the same size as StandardCircuit with DRONE and is much smaller with CRISP.

Next, we measured the search time to obtain the optimal weight[12], using the search strategy UnifiedSearch. Figure 7.3 (right) plots the average search time for each instance. For both datasets, search time is closely related to the number of edges in the circuit. With the CRISP dataset with larger k ($k = 12$), CompactCircuit recorded the best search time. This was not true with DRONE; for that dataset, CompactCircuit is even slower than NaiveCircuit. It is also surprising for us that StandardCircuit performed best on most of the DRONE instances. This suggests that the circuit size is not the only factor for determining the running time.

7.8.2 Choice of Search Strategies

For evaluating search strategies, we ran the algorithm with each strategy with the circuit type SemiCompactCircuit and measured the search time (as done in the previous experiment). Figure 7.4 (left) plots the average search time for various instances. We observe that UnifiedSearch is the fastest with a few exceptions with DRONE, which matches our expectation. ProbabilisticBinarySearch is more unstable; it is faster than StandardBinarySearch with CRISP except $n = 100, k = 12, \lambda = \lambda_{\text{large}}$, but it is the slowest among three with DRONE.

7.8.3 Choice of Solution Recovery Strategies

For each recovery strategy, we measured the time for solution recovery after finding the optimal weight, using circuit type SemiCompactCircuit. As shown in Figure 7.4 (right), LasVegasRecovery performed better than MonteCarloRecovery with all test instances. Moreover, unlike MonteCarloRecovery, it is guaranteed that LasVegasRecovery always succeeds. We conclude that LasVegasRecovery has clear advantages over MonteCarloRecovery.

7.8.4 Choice of Scaling Factors

Finally, we evaluated the effect of varied scaling factors (λ) by running our algorithm using subroutines SemiCompactCircuit, UnifiedSearch and LasVegasRecovery with fixed $k = 12$.

We first measured how the weight of the walk obtained by ALG-IPA is close to the

[12]The time for constructing a circuit was negligible (less than 1 second).

optimal weight. Figure 7.5 (left) shows the ratio of the weight by ALG-IPA to the optimal (the lower, the better). This ratio was less than 1.2 for all test instances, and with large enough scaling factors (100 for CRISP, 0.25 for DRONE), the ratio converges within 1.04. Figure 7.5 (middle) plots the peak memory usage for ALG-IPA. We observed the almost linear growth of memory usage with respect to the scaling factor. This is due to the fact that the size of an arithmetic circuit is proportional to the solution weight in integers.

Figure 7.5 (right) shows the overall running time (including search time and solution recovery time) for each instance. The precise running times depend on the structure of an instance[13] as well as randomness, but our experiment demonstrates that running time increases with the scaling factor. Taken together, the plots in Figure 7.5 illustrate a trade-off between fidelity (influenced by rounding errors) and computational resources (time and space).

7.8.5 Preprocessing Results

We present results of preprocessing on our GRAPH INSPECTION instances. Because k (the number of colors present in the graph) is small, this preprocessing is quite effective. In particular, removing colorless vertices significantly reduces the size of the graphs.

7.9 Conclusion

In this chapter we present a novel approach for solution recovery for MULTILINEAR DETECTION and applied these findings to design a solver for GRAPH INSPECTION, thereby addressing real-world applications in robotic motion planning. Using a modular design, we tested variants of different algorithmic subroutines, namely search strategy, circuit design, and solution recovery. Some of our findings are unambiguous and should easily translate to other implementations based on arithmetic circuits: First, we can recommend that parameter search (like the solution weight ℓ in our case) is best conducted by constructing a single circuit with multiple outputs for all candidate values. Second, optimizing the circuit design towards size is a clear priority as otherwise the number of circuit edges

[13]It is observable that, for example, DRONE $n = 50$ takes longer than DRONE $n = 100$ when $\lambda = 1.0$. This is because the original graph size is not the only indicator of running time. Especially, the time for solution recovery heavily depends on the structure of optimal and suboptimal solutions. In this particular case, DRONE $n = 100$ is faster because it can prune many suboptimal branches in the search space early in the solution recovery step.

quickly explodes. Finally, of the two novel solution recovery algorithms for MULTILINEAR DETECTION, LasVegasRecovery is the clear winner.

Many questions remain open. While our solver is not competitive with the existing solvers on realistic instances of GRAPH INSPECTION, it is quite plausible that it can perform well in other problem settings where memory is much more of a concern than running time. One drawback of the current circuit construction is the reliance on the transitive closure of the input graph. A more efficient encoding of the underlying metric graph could substantially reduce the size of the resulting circuit. Further, there might be preprocessing rules for arithmetic circuits that reduce their size without affecting their semantic. Finally, on the theoretical side, we would like to see whether other problems can be solved using the concept of tree certificates.

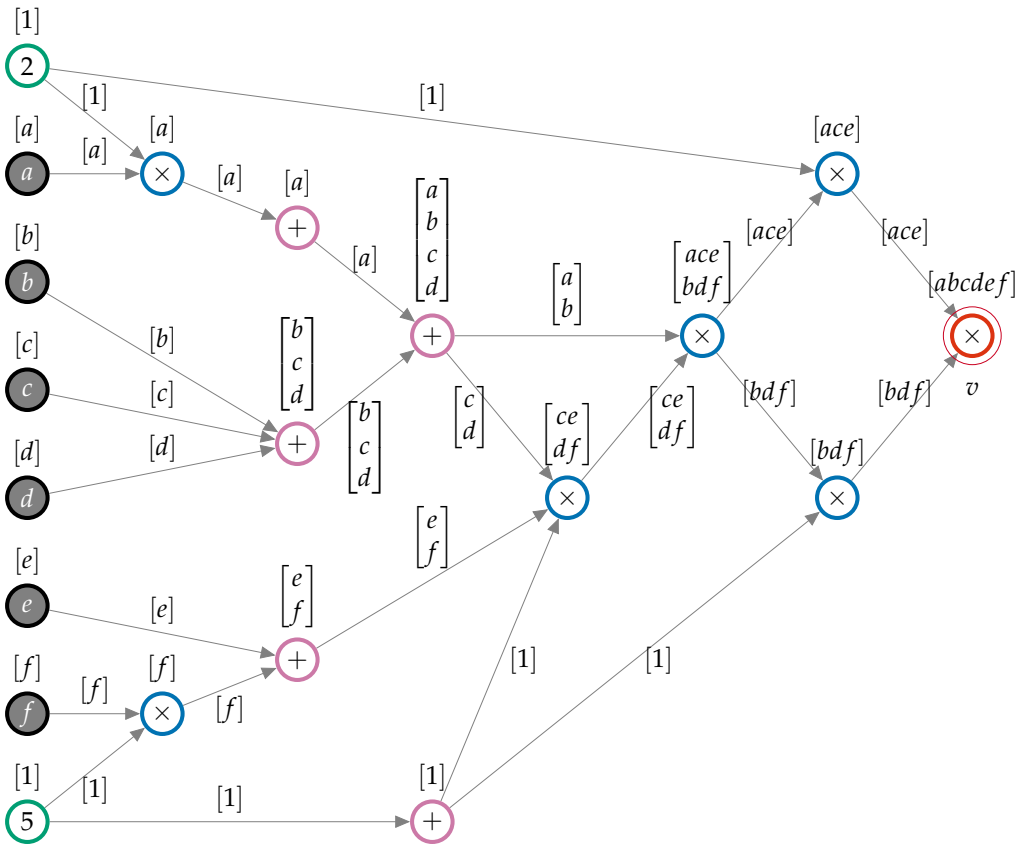


Figure 7.1: An illustration of the certificate flow for a certificate $\hat{\mathcal{F}}$ on variables $X := \{a, b, c, d, e, f\}$ such that $P_{\hat{\mathcal{F}}[v]}(X) = 2 \cdot 5^3(2a + b + c + d)^4(e + 5f)^2$, which contains the multilinear monomial $2 \cdot 5^3 \cdot (2 \cdot 4!) \cdot (5 \cdot 2!)abcdef = 120000abcdef$. The nodes in $\hat{\mathcal{F}}$ consist of variables (gray), scalar inputs (green), addition gates (purple), internal multiplication gates (blue), and the output multiplication gate v (red). The *flow amount* at all nodes and edges is depicted as a column vector of multilinear monomials in the figure.

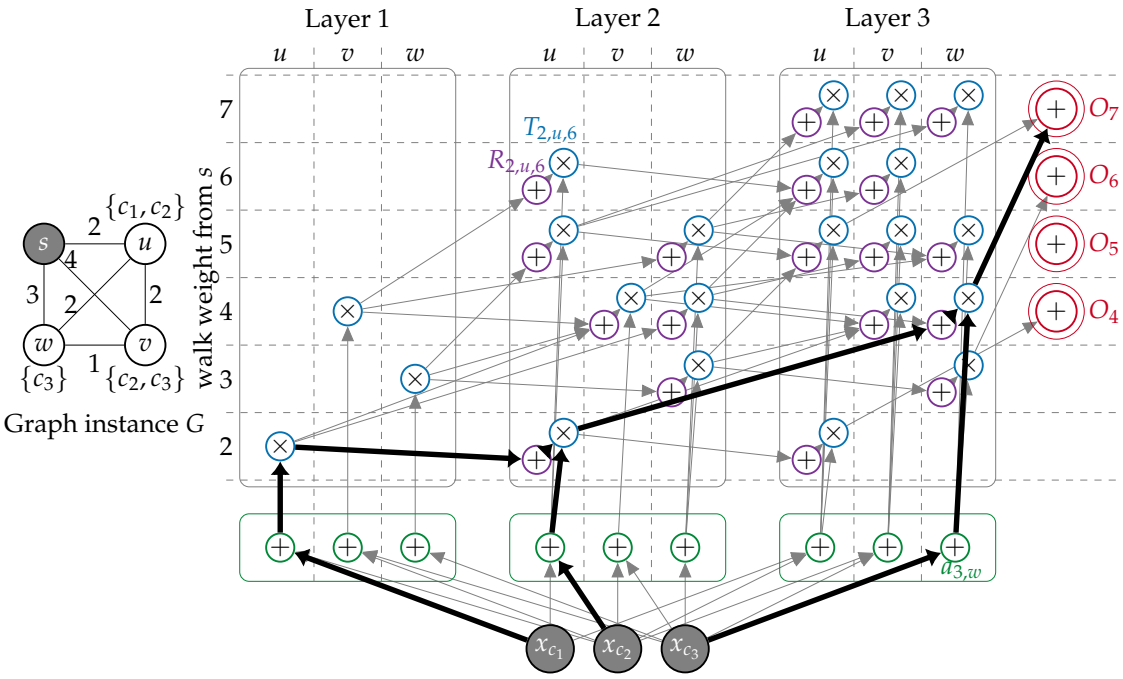


Figure 7.2: An example of an arithmetic circuit (CompactCircuit with UnifiedSearch), encoding the graph instance G with colors $\mathcal{C} = \{c_1, c_2, c_3\}$, illustrated on the left, with $t = 3$. The circuit consists of variable nodes (gray) for each color in $\mathcal{C}$, auxiliary nodes (green), receivers (purple), transmitters (blue), and output nodes (red). Notice that each receiver/transmitter pair is identifiable by a layer, index ($V \setminus \{s\}$ for CompactCircuit), and walk weight from the starting vertex s. A tree certificate, corresponding to walk (s, u, w, s) with weight 7, is highlighted in bold.

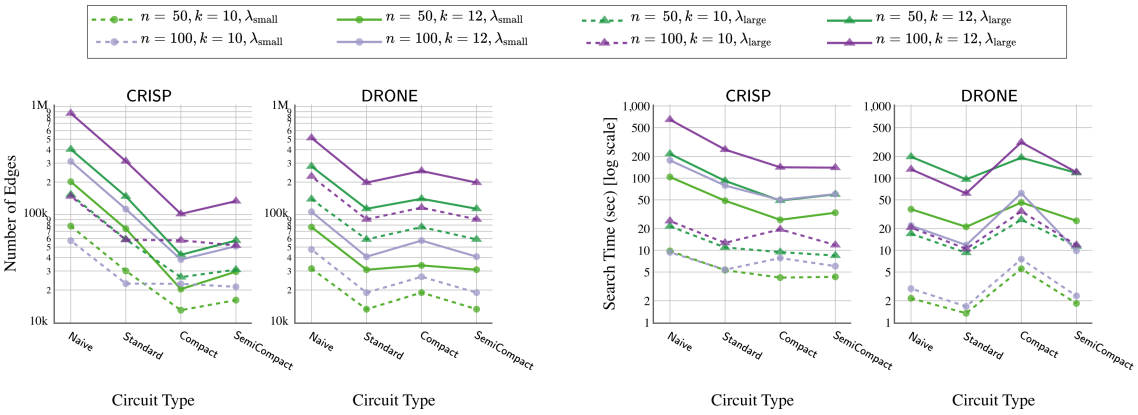


Figure 7.3: The number of edges in the circuit (left) and average search time (right) for each circuit type.

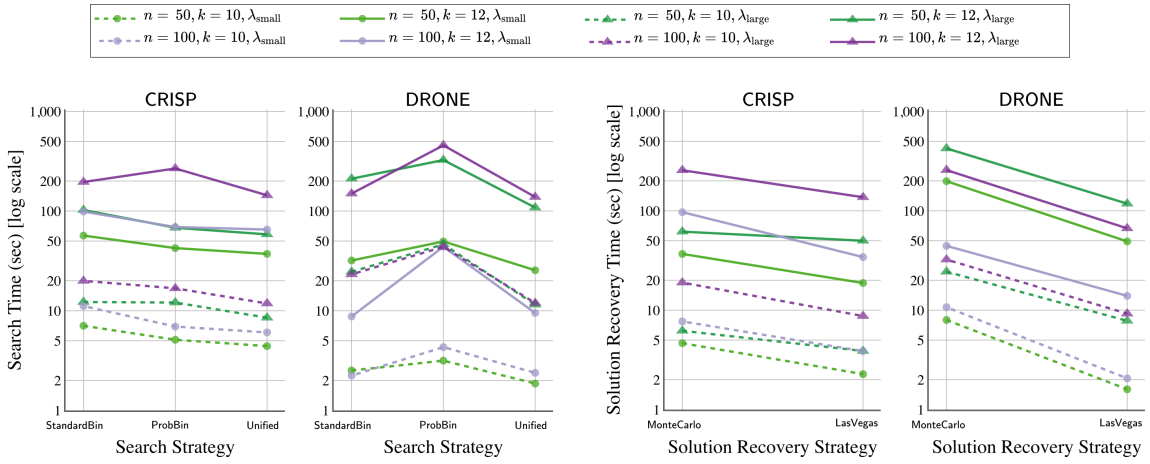


Figure 7.4: Runtime of each subroutine: search strategies (left) and solution recovery strategies (right).

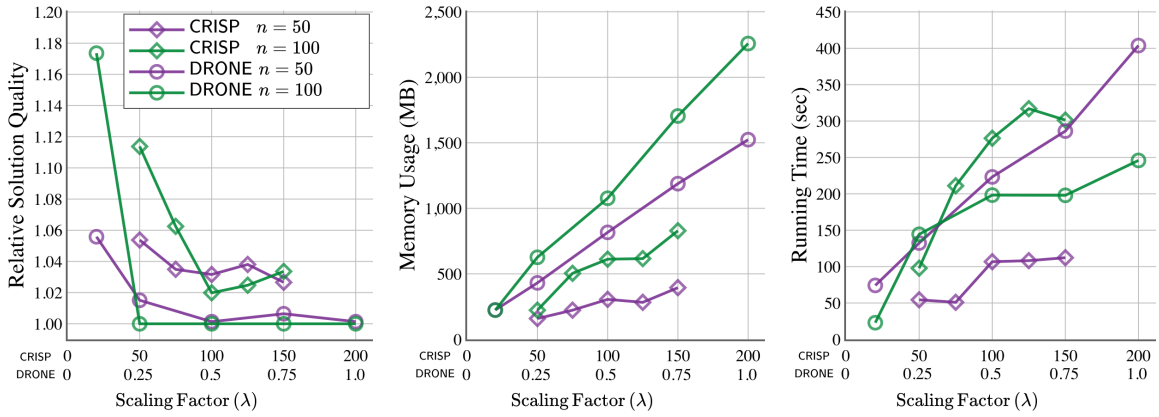


Figure 7.5: The average ratio of the weight obtained by ALG-IPA to the optimal weight (left), the average peak memory usage (middle), and the average overall running time (right) for different scaling factors.

Table 7.1: Instance sizes before and after preprocessing. n_{build} is the parameter given to the software of Fu *et al.* [7] during instance construction, and can be thought of as a "target" number of vertices for the constructed graph. k is the number of colors remaining after performing color reduction. n and m are the number of vertices and edges in the color-reduced instance, while n' and m' are the same statistics after preprocessing.

Dataset	n_{build}	k	n	m	n'	m'
CRISP	50	10	70	509	15	105
		12			17	136
	100	10	113	951	20	190
		12			24	276
DRONE	50	10	64	329	12	66
		12			14	91
	100	10	119	878	16	120
		12			19	171

REFERENCES

[1] M. BENTERT, D. C. SALOMAO, A. CRANE, Y. MIZUTANI, F. REIDL, AND B. D. SULLIVAN, *Graph Inspection for Robotic Motion Planning: Do Arithmetic Circuits Help?*, 2024. arXiv:2409.08219 [cs].

[2] A. BJÖRKLUND, P. KASKI, L. KOWALIK, AND J. LAURI, *Engineering motif search for large graphs*, in Proceedings of the 17th Workshop on Algorithm Engineering and Experiments (ALENEX), SIAM, 2015, pp. 104–118.

[3] M. CYGAN, F. V. FOMIN, L. KOWALIK, D. LOKSHTANOV, D. MARX, M. PILIPCZUK, M. PILIPCZUK, AND S. SAURABH, *Parameterized Algorithms*, Springer, 2015.

[4] R. DIESTEL, *Graph Theory*, vol. 173 of Graduate Texts in Mathematics, Springer-Verlag, Berlin, third ed., 2005.

[5] B. ENGLOT AND F. HOVER, *Planning complex inspection tasks using redundant roadmaps*, in Proceedings of The 15th International Symposium on Robotics Research (ISRR), Springer International Publishing, 2017, pp. 327–343.

[6] M. FU, A. KUNTZ, O. SALZMAN, AND R. ALTEROVITZ, *Asymptotically optimal inspection planning via efficient near-optimal search on sampled roadmaps*, The International Journal of Robotics Research, 42 (2023), pp. 150–175.

[7] M. FU, O. SALZMAN, AND R. ALTEROVITZ, *Computationally-efficient roadmap-based inspection planning via incremental lazy search*, in 2021 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2021, pp. 7449–7456.

[8] S. GUILLEMOT AND F. SIKORA, *Finding and counting vertex-colored subtrees*, Algorithmica, 65 (2013), pp. 828–844.

[9] S. KARAMAN AND E. FRAZZOLI, *Sampling-based algorithms for optimal motion planning*, The international journal of robotics research, 30 (2011), pp. 846–894.

[10] P. KASKI, J. LAURI, AND S. THEJASWI, *Engineering motif search for large motifs*, in Proceedings of the 17th International Symposium on Experimental Algorithms (SEA), Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, pp. 28:1–28:19.

[11] I. KOUTIS, *Faster Algebraic Algorithms for Path and Packing Problems*, in Proceedings of the 35th International Colloquium of Automata, Languages and Programming (ICALP), Springer, 2008, pp. 575–586.

[12] I. KOUTIS AND R. WILLIAMS, *LIMITS and Applications of Group Algebras for Parameterized Problems*, ACM Transactions on Algorithms, 12 (2016), pp. 31:1–31:18.

[13] F. LAZEBNIK, *On Systems of Linear Diophantine Equations*, Mathematics Magazine, 69 (1996), pp. 261–266. Publisher: [Mathematical Association of America, Taylor & Francis, Ltd.].

[14] Y. MIZUTANI, D. C. SALOMAO, A. CRANE, M. BENTERT, P. G. DRANGE, F. REIDL, A. KUNTZ, AND B. D. SULLIVAN, *Leveraging fixed-parameter tractability for robot inspection planning*, arXiv preprint arXiv:2407.00251, 2024, (2024).

[15] P. C. POP, O. COSMA, C. SABO, AND C. P. SITAR, *A comprehensive survey on the generalized traveling salesman problem*, European Journal of Operational Research, 314 (2024), pp. 819–835.

[16] R. WILLIAMS, *Finding paths of length k in $O^*(2^k)$ time*, Information Processing Letters, 109 (2009), pp. 315–318.

CHAPTER 8

CONCLUSION AND FUTURE WORK

This dissertation exhibits various applications of parameterized algorithms, with the goal of creating algorithms that will not be confined to "theoryland". As listed in Section 1.2, we implemented our algorithms and verified their practicality with real-world datasets (bioinformatics in Chapter 2, robotics in Chapters 6 and 7, etc.). Smaller kernels result in exponentially faster running time (Chapter 2), and we showed parameterized approaches can outperform traditional search algorithms in practice (Chapter 6). The parameterized preprocessing introduced in Chapter 3 involves a collection of nontrivial technical tools such as color coding, matroids, and vertex separators, achieving an important step towards further applications and generalization.

Structural parameters (as studied in Chapters 4 and 5) are very powerful when input instances have bounded widths. Interestingly, the trade-offs between efficient algorithms and the "strength" of parameters can also be explained by a connection to logic. It is known that model checking in monadic second-order logic (MSO MODEL CHECKING) in graphs of bounded clique-width is FPT [2], and similarly, model checking in first-order logic (FO MODEL CHECKING) in graphs of bounded twin-width is FPT [1][1]. Twin-width has become an emerging area of research in this decade, continuously producing fruitful theoretical results. Such a situation has increased the importance of twin-width solvers for real-world instances (see Chapter 5).

As detailed in each chapter, there are several open problems and future directions of research. Here I would like to highlight potential future work that would have impact and promising results in real-world applications.

First, our work on algebraic techniques for Inspection Planning (Chapter 7) is missing a direct comparison to the state-of-the-art solver for GRAPH INSPECTION. We plan

[1]Given a witness is provided.

additional experiments so that we can more precisely estimate when the algebraic solver can benefit from having polynomial memory footprint. We will also need code-level performance optimization to make the running time more competitive. In addition to the main work, I want to extend the results on general certificates (Section 7.3.3). One major open problem is whether we can directly detect a general certificate. I am also interested in implementing massively parallel multilinear detection using GPGPUs. Is there any chance an algebraic solver can then outperform the DP solver in terms of running time?

Second, our work on ML-based twin-width solvers (Section 5.6) is incomplete. For methodology, we have seen what works well for supervised learning, and next plan to implement and evaluate unsupervised learning. There are also several important open questions in the theory of twin-width. I am especially curious about new bounds on twin-width as they might guide and speed up twin-width solvers. For example, the maximum twin-width of a planar graph is still open (either 7 or 8) although the recent work by Jedelský [3] strongly suggests 7 is the right answer. Another open question is whether there is an n-vertex graph of twin-width at least $\lceil n/2 \rceil$.

For antler decompositions (Chapter 3), we are looking for general results applicable to other problems. We are also investigating decompositions of directed graphs, aiming to an application for DIRECTED FEEDBACK VERTEX SET. The tight OCC for ODD CYCLE TRANSVERSAL still needs more generalization. At least for a tri-partition (B, C, R) defined similarly to the OCC, we want to allow some edges between B (bipartite part) and R (remainder part), as in the original antler decomposition for FEEDBACK VERTEX SET. Then, the part C is no longer a vertex separator, introducing several technical challenges.

A missing piece in the study of the happy set problems (Chapter 4) is the parameterized complexity of DENSEST k-SUBGRAPH parameterized by modular-width. I suspect that it is W[1]-hard, but we know little about hardness reductions involving modular-width. It would be nice if we can establish a framework to show limitations of modular-width. Additionally, little work exists on kernelization with respect to modular-width.

Bringing FPT tools into practical applications is a promising course of research, and I am excited to bridge the gap between theory and practice.

REFERENCES

[1] E. BONNET, E. J. KIM, S. THOMASSÉ, AND R. WATRIGANT, *Twin-width I: tractable FO model checking*, in 2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS), 2020, pp. 601–612.

[2] B. COURCELLE, J. A. MAKOWSKY, AND U. ROTICS, *Linear Time Solvable Optimization Problems on Graphs of Bounded Clique-Width*, Theory of Computing Systems, 33 (2000), pp. 125–150.

[3] J. JEDELSKÝ, *Twin-width of planar graphs*, PhD thesis, Masarykova univerzita, Fakulta informatiky, 2024.