ABSTRACT

VAN DER POEL, ANDREW JOHN. Approaches for Graphs Near Structural Classes. (Under the direction of Blair D. Sullivan.)

Many real-world networks arise from processes which imply that the graph should have a particular structure. Unfortunately, often noise and errors in the data prevent this structure from existing, confounding, and in some cases, preventing analysis. One approach to resolving this discrepancy is through graph modification – the addition and deletion of nodes and edges until the resulting graph has some desired property. Modification problems are well-studied in graph theory, and are of particular interest to algorithm designers for graphs which are close to a structural class, since they naturally lend themselves to parameterization. Recently, they have also been used in the structural rounding technique of Demaine et al. for extending approximation algorithms.

Historically, most work on graph modification has focused on finding a (smallest) set of edits, but algorithmically it can also be interesting to use structure implied by an edit set to solve other classically hard problems. These settings naturally lend themselves to both exact and approximate variants of problems.

This dissertation focuses on expanding the frontier of graph modification algorithms, and does so in several directions, mixing approximation and parameterization. First, we give an exact algorithm for the parameterized version of the problem of finding an edge edit set whose deletion yields a path forest, significantly reducing the best-known complexity. We also give two novel algorithms which exactly enumerate all maximal induced complete bipartite subgraphs for graphs which are near-bipartite (have a small vertex edit set to some bipartite graph) and provide an empirical comparison with the state-of-the-art approach on general graphs. Finally, we develop a suite of algorithms for "lifting" solutions to Vertex Cover within the structural rounding framework (extending solutions on a subgraph to apply to the entire graph). We experimentally evaluate the effectiveness of these approaches on both synthetic and real-world graphs.

Overall, this work yields several novel algorithms involving graph modification in various ways. When it is applicable, we show these algorithms theoretically and practically outperform competitors.

Approaches for Graphs Near Structural Classes

by
Andrew John van der Poel

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2019

APPROVED BY:

Steffen Heber	Carla D. Savage
Matthias F. Stallmann	Blair D. Sullivan Chair of Advisory Committee

DEDICATION

To my parents, for always encouraging my curiosity.

BIOGRAPHY

Drew van der Poel grew up on the banks of the Hudson River in Kingston, NY. Prior to arriving at North Carolina State University, he double majored in Computer Science and Mathematics and minored in Business at the College of Saint Rose in Albany, NY.

ACKNOWLEDGEMENTS

I would like to thank Dr. Blair D. Sullivan for her years of advising and assistance with shaping this work. I deeply appreciate the opportunity I was given in her lab and the patience and respect I was always shown. Many thanks to Dr. Felix Reidl and Dr. Kyle Kloster for their guidance within the lab and, more importantly, their friendship outside. Likewise, my fellow PhD students Dr. Michael P. O'Brien, Timothy D. Goodrich, Brian Lavallee, and Michael Breen-McKay were always willing to lend a helping hand and work through problems together, while being the source of many laughs. You have all made my time as PhD student incredibly rewarding and enjoyable.

Thank you to my committee members, Dr. Steffen Heber, Dr. Carla D. Savage, and Dr. Matthias F. Stallmann, for your insights, feedback, and questions regarding this dissertation. It is an honor to have such talented faculty as part of my committee. I would also like to thank faculty members from Saint Rose, Dr. Jim Teresco, Dr. Amina Eladdadi, and Dr. Mary Ann McLoughlin for encouraging me to pursue this degree and serving as great ambassadors for higher education.

Finally, to my tremendous friends and family who have been an incredible support network during my time in Raleigh. The frequent messages and many visits from friends in New York and elsewhere were always welcomed and appreciated; I never felt that home was 600 miles away. Further, many of my fondest memories from this time come from playing sports with my soccer, volleyball, basketball, softball, and ultimate frisbee teams and groups in Raleigh; thank you to the many terrific folks who helped provide a much needed distraction from PhD life! However, my favorite distractions were the countless phone calls with my parents and grandparents; you all have always been there for me and words cannot express my gratitude for this. Problem solving has never been more fun than practicing Math Olympiad with Pop Pop at the kitchen table ("you gotta think outside the box!"). And to my lovely wife, Kathryn, your support and encouragement have been paramount throughout this entire process; I could not have done it without you. This closes a wonderful chapter in our lives, and I look forward to continuing to write the future ones together.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

1.1 Motivation

Data from a wide array of fields can be modeled as interactions in a network. Viewing data through this lens has naturally led to network analysis and graph algorithms being extremely useful tools in many different domains. Unfortunately, many graph problems are computationally difficult to solve. However, when networks have specific structure, they are much easier to work with, and common tasks may become much faster when restricted to particular classes of graphs. In the real-world, data simply does not have the structure which is necessary to take advantage of the aforementioned structure-based algorithms. Quite often this can be attributed to noise and errors within the data, which prevent the domain networks from belonging to any rigid class.

A classic approach to dealing with the unstructured data of the real-world is *graph modification*: deleting or adding vertices and/or edges to an input graph so that the resulting graph has a particular structure. In general, if the amount of modification needed is small, we are able to obtain better algorithms. To gain insight towards graph modification and small edit sets, consider the following example. In order for zoologists to infer phylogeny, the data must be in the form of a tree. However, due to errors in the data, raw zoological graphs are often not trees. One approach for inferring phylogeny is thus to delete edges from the graph so that the remaining graph is a tree. When the number of errors, and in turn required deletions, is known to be small, faster modification algorithms can be designed and inferences are more likely to be meaningful. To further understanding, we note that this concept is similar to detecting outliers in a statistical distribution. However, while finding outliers can be done relatively easily, it is computationally hard to find errors while preserving as much data as possible (i.e. finding the minimum number of modifications needed to obtain desired

structure).

Designing new efficient graph modificiation algorithms for graphs near structural classes is the main idea of this dissertation. We work with structural classes as opposed to spectral or other types of graph classes because they naturally capture desired features of real-world data. Editing algorithms perform best on graphs near structural classes, as opposed to ones which are not “close" to having structure, because the number of modifications is small. Further, advances in this space apply to a much larger set of graphs than when working with the more rigid, traditional classes. We consider three types of graph modification problems in a naturally advancing progression. The first exploits being close to a structural class to find an edit set via a parameterized algorithm. This is similar to the zoological example mentioned above, and is in many ways the “traditional” type of modification problem. Second, we consider the case where we are given an edit set, rather than being tasked with finding one. In this setting we leverage the structure implied by a small edit set to exactly solve a problem on the entire graph (both the edit set and the rest of the graph). We further extend this idea to approximately solving problems on the entire graph. Rather than solving a problem exactly, approximation algorithms give theoretical guarantees on the size of a solution it produces relative to the optimal solution.

Our work demonstrates that designing exact and approximation algorithms which both find edit sets and take advantage of the structure implied by edit sets are relevant, realistic, and robust approaches for utilizing graph-based techniques to solve real-world problems.

1.2 Research Directions and Outline

This work expands the algorithmic frontier of graph modification in a number of ways. We made progress in each direction by focusing on specific problems, which I now describe, along with an outline of how the document proceeds. Chapter 2 describes the necessary background on graph theory, computational complexity, and algorithmic methods to understand the research. In Chapter 3, we solve a parameterized problem where we want to find an edit set. Namely, the k-Co-Path Set problem asks if there exists a set of k edges whose removal yields a network where all connected components are paths? We solve this problem by giving a novel, fastest-known algorithm, which utilizes kernelization, branching, and the Cut & Count technique.

We then work on a second parameterized problem, but in the setting where we are given an edit set and aim to leverage the implied structure. In Chapter 4, we are given a graph with a set of nodes whose removal leaves a bipartite graph and describe two novel algorithms which enumerate all maximal induced complete bipartite subgraphs in such a graph. We also provide an open-source implementation and experimentally evaluate our approaches on a large suite of synthetic and real-world graphs.

In Chapter 5, we again operate in the case where we are given an edit set, but here the problem is motivated by structural rounding, a new perspective in the field of theoretical computer science. Given an edit set and a vertex cover on the rest of the graph (a set of vertices where every edge with

both endpoints outside the edit set has an endpoint in the set), we augment that solution to apply to the entire graph and provide an approximation factor on the augmentation. This is a fundamental part of the structural rounding framework which is used to obtain approximation algorithms on graphs near structural classes. We developed a suite of new algorithms which solve this problem and implemented and experimentally evaluated their effectiveness in practice, both on real-world and synthetic graphs.

Finally, the contributions of this work and future research directions are summarized in Chapter 6. Each chapter begins with an executive summary of its content, and my contributions to any collaborative work.

CHAPTER

2

BACKGROUND

Executive Summary

The algorithms we develop rely on knowledge of parameterized complexity, graph modification, and odd cycle transversals. In this chapter we explain these concepts, in addition to providing terminology and notation used throughout the document.

2.1 Terminology

In general we let $G = G(V,E)$ be the graph with vertex set V and edge set E. Unless otherwise noted, we assume $|V| = n$ and $|E| = m$; we let $N(v)$ denote the set of neighbors of a vertex v, and let $\deg(v) = |N(v)|$. We also assume that the graphs we work with are finite and undirected. We use $G[S]$ to represent the subgraph induced on S. If $S \subseteq V$, then $G[S]$ is subgraph of G with vertex set S and any edges with both endpoints in S. If $S \subseteq E$, $G[S] = G(V,S)$. We use $\setminus$ to represent the set subtraction operation; that is for sets A, B, $A \setminus B$ is the set of elements in A which are not in B. A graph is said to be *bipartite* if its vertices can be partitioned in to two disjoint independent sets. Chapters 4 and 5 pertain to graphs which are related to bipartite graphs.

2.2 Parameterized Complexity

Parameterized complexity provides a more granular view of computational problems than traditional "P vs. NP". The aspect of parameterized complexity which is most relevant in this document is the complexity class FPT.

Definition 1. *Given an instance of a problem with size n and a positive integer k, an algorithm is in the class FPT if it runs in time $O(f(k)n^{O(1)})$, where f is an arbitrary function that only depends on k. Such an algorithm is referred to as an fpt-algorithm.*

A problem which admits an fpt-algorithm is said to be fixed-parameter-tractable (FPT). The computational complexity of fpt-algorithms is commonly written in O^* notation, where $O(f(k)n^{O(1)})$ is written as $O^*(f(k))$, suppressing the polynomial factors. We say an algorithm is *linear-fpt* if the complexity is $O(f(k)n)$. In Chapters 3, and 4 we provide novel parameterized algorithms. These approaches are parameterized by the size of the edit set, which follows from graph modification.

Parameterized complexity was first introduced by Downey and Fellows [Dow12]. Many graph problems are known to be in FPT, including VERTEX COVER [Cyg15], FEEDBACK VERTEX SET [Che08a], and subgraph enumeration [Dam06]. These algorithms all make use of the *natural parameter*, which corresponds to the size of the solution or what is being enumerated. We use the natural parameter in this dissertation. It is also possible to solve problems when using a structural parameter, such as treewidth [Lok11] and cliquewidth [Fom10]. In Chapter 3 we relate the natural parameter to the treewidth of the instances we run our algorithm on, and the runtimes can be stated in terms of either.

There exists an extensive toolbox of algorithmic techniques which can be used when designing parameterized approaches. Kernelization is a series of pre-processing steps which creates a smaller, equivalent instance, the *kernel*, by handling the simpler parts of the input. VERTEX COVER in particular has seen kernelization heavily used [AK04; AK07; Jan13]. We make use of kernelization in our algorithm in Chapter 3, in addition to a branching strategy. Branching, another common FPT technique, uses the parameter to create bounded-depth search trees, as the parameter allows early pruning when exhaustively searching for solutions. This strategy can be seen in [Cyg15] and [Nas10], the latter of which is tailored for graph modification. A third FPT tool is used in Chapter 3, the Cut&Count framework [Cyg11a]. This is a general framework for parameterized problems on graphs with bounded treewidth, where it is cleverly ensured that any non-solutions are seen an even number of times. This enables doing the total count modulo two, and using additional techniques effectively bounds the likelihood of a false negative.

2.3 Graph Modification

Graph modification is the process of editing a graph so that it has some desired structure or property. These edits can be done in the form of vertex/edge additions/deletions. In this dissertation we restrict ourselves to the deletion case where the edit set is homogenous (completely contained in either V or E). Here we aim to find an edit set $X \subseteq V$ (E) such that $G[V$ $(E) \setminus X]$ has some property. In Chapter 3, $X \subseteq E$ and the property is being acyclic and having maximum degree 2 (e.g. being a path forest). In Chapters 4 and 5, $X \subseteq V$ and $G[V \setminus X]$ is a bipartite graph.

Graph modification is an actively pursued research field [Gra04; Sha04; Zha16] and has applications in computational biology [Sha02], machine learning [Ban04], and pattern matching [Gao10],

amongst other fields. As noted in [Dem19], a significant amount of work has been spent solving problems where the aim is to find a maximum induced subgraph which satisfies some property [Kri79; Lew78; Yan78]. This is an alternative lens through which to view graph modification, but yields equivalent, complementary results. When the property of interest exists in infinitely many graphs and does not exist in infinitely many graphs, and is satisfied in every induced subgraph of a graph satisfying it, then deleting vertices to arrive at such a graph is NP-hard [Lew80], and furthermore necessitates exponential running time assuming the Exponential Time Hypothesis [Kom18].

Using parameterized approaches for modification problems has been well studied. Dabrowski et al. showed that editing to a specified degree sequence is W[1]-complete in general, but is in FPT on planar graphs [Dab17]. There are also parameterized algorithms for the NP-complete problems of editing to threshold and chain graphs [Dra15], star forests [Dra15], multipartite cluster graphs [Fom14], and H-free graphs where H has bounded indegree and is finite [Dra16]. We regularly work in the context of vertex-deleting to bipartite graphs, which we describe in the next section.

2.4 Odd Cycle Transversals

In Chapters 4 and 5, we make use of a given odd cycle transversal (OCT set), a set of nodes O, such that $G[V \setminus O]$ is bipartite. Thus, OCT sets are a form of "near-bipartiteness" which arises naturally in many applications [Gül04; Pan04; Sch17]. Although finding a minimum size OCT set is NP-hard, the problem of deciding if an OCT set with size k exists is fixed parameter tractable, with algorithms in [Lok09] and [Iwa14] running in times $O(3^k kmn)$ and $O(4^k n)$, respectively. Other algorithms for OCT include a $O(\sqrt{log(n)})$-approximation [Aga05], a randomized polynomial kernelization algorithm based on matroids [Kra14], and a subexponential algorithm for planar graphs [Lok12]. Our algorithms only require a valid OCT set (not a minimal or optimal one), so any of these approaches or one of several high-performing heuristics may be used to pre-process the data. Recent implementations [Goo18] of a heuristic ensemble alongside algorithms from [Aki16; Hüf05] alleviate concerns about finding an OCT decomposition creating a barrier to usability for our algorithms.

CHAPTER

3

EDGE EDITING TO PATH FORESTS

Executive Summary

The k-Co-Path Set problem asks, given a graph G and a positive integer k, whether one can delete k edges from G so that the remainder is a path forest, a collection of disjoint paths. We give a randomized linear-fpt-algorithm with complexity $O^*(1.588^k)$ for deciding k-Co-Path Set, significantly improving the previously best known $O^*(2.17^k)$ of Feng, Zhou, and Wang (2015). Our main tool is a new $O^*(4^{t\,w(G)})$ algorithm for Co-Path Set using the Cut&Count framework, where $t\,w(G)$ denotes treewidth. In general graphs, we combine this with a branching algorithm which refines a $6k$-kernel into reduced instances, which we prove have bounded treewidth.

This chapter was advised by Blair D. Sullivan and the results were published in the proceedings of the 2016 International Symposium on Parameterized and Exact Computation (IPEC) [Sul16].

3.1 Introduction

We study parameterized versions of Co-Path Set [Che08b; Zha14a], an NP-complete problem asking for the minimum number of edges whose deletion from a graph results in a path forest, a collection of disjoint paths (also a graph which is acyclic and has maximum degree 2); the deleted edges are called a *co-path set* (see Figure 3.1). Specifically, we are concerned with k-Co-Path Set, which uses the natural parameter of the number of edges deleted.

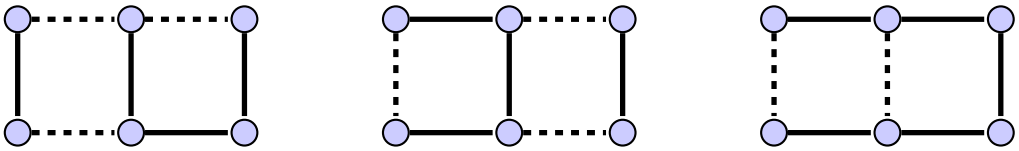

Figure 3.1 Three co-path sets (dashed edges), including one of minimum size (rightmost).

k-CO-PATH SET **Input:** A graph $G=(V,E)$ and a non-negative integer k. **Parameter:** k **Problem:** Does there exist $F \subseteq E$ of size exactly k such that $G[E \setminus F]$ is a set of disjoint paths?

These problems are naturally motivated by gene mapping, where biologists determine the ordering of genetic markers in DNA using fragment data created by breaking chromosomes with gamma radiation (a technique known as *radiation hybrid mapping*) [Cox90; RI91; Slo97]. The success of this method depends on being able to construct a linear ordering of the markers based on their co-location on a fragment. Unfortunately, human error in distinguishing markers often means the constraints implied by markers' co-occurrence on fragments are incompatible with all possible linear orderings, necessitating an algorithm to find the "best" ordering (one which violates the fewest constraints). CO-PATH SET solves the special case where each DNA fragment contains exactly two genetic markers (corresponding to an edge in the graph); any linear ordering of the markers must correspond to some set of paths, and we minimize the number of unsatisfied constraints (edges in the co-path set).

Recent algorithmic results related to CO-PATH SET include two parameterized algorithms deciding k-CO-PATH SET [Fen14; Fen15], the faster of which [Fen15] has time complexity $O^*(2.17^k)$, and a (10/7)-approximation algorithm [Che11]. However, as written, both parameterized results [Fen14; Fen15] contain a flaw in their analysis which invalidates their probability of a correct solution in the given time[1]. The best known bound prior to [Fen14] is an $O^*(2.45^k)$ algorithm [Zha14a]. In this chapter, we prove:

Theorem 1. *k-CO-PATH SET is decidable in $O^*(1.588^k)$ linear-fpt time with probability at least $2/3$.*

We note that standard amplification arguments apply, and Theorem 1 holds for any success probability less than 1. Further, if f is an increasing function with $\lim_{n\to\infty} f(n)=1$, we can solve k-CO-PATH SET with success probability at least $f(n)$ in $O(1.588^k\, n \operatorname{polylog}(n))$.

We describe our result in thorough detail in the remainder of this chapter, but will now describe our approach at a high-level to aid with the reader's intuition. We begin by kernelizing using the

[1] Step 2.11 in both versions of Algorithm R-MCP checks if a candidate co-path set F has size $\leq k_1$ (as they are sweeping over all possible sizes of candidates and want to restrict the size accordingly). If F is too large, the algorithm discards it and continues to the next iteration. However, in order for their analysis to hold, the probability that the candidate is contained in a co-path set must be $\geq (1/2.17)^{k_1}$ (or $(1/2.29)^{k_1}$ in [Fen14]) for *every* iteration. Candidates which are too large may have significantly smaller probability of containment, yet are counted in the exponent of the analysis.

technique from [Fen15], which handles easy cases such as connected components that are cycles and paths, contracts long paths, and applies slightly more technical rules to achieve a kernel. We then make use of a branching technique which yields bounded degree instances. The restrictive structure of path forests enables efficient branching. As part of branching, we delete any instances where more than k edges were removed, and then we pass the remaining instances to our application of the Cut&Count framework. Cut&Count [Cyg11a] is a general framework used for solving decision problems where there is a global constraint. It requires bounded treewidth and uses dynamic programming over a tree decomposition. The key idea is that when building potential solutions in the natural way via dynamic programming, the framework guarantees that non-path forests are seen an even number of times. Then we can count the number of times a potential solution is seen modulto two and obtain a guaranteed “no" if no solution exists and a “yes" with high probability if one does (this is via a one-sided Monte Carlo method). This naturally leads to our result in Theorem 1.

The remainder of this chapter is organized as follows: after essential definitions and notation in Section 3.2, we start in Section 3.3 by giving a new $O^*(4^{tw(G)})$ algorithm `tw-copath` for solving Co-Path Set parameterized by treewidth (tw) using the Cut&Count framework [Cyg11a]. Finally, Section 3.4 describes the linear-fpt algorithm referenced in Theorem 1, which solves k-Co-Path Set on general graphs in $O^*(1.588^k)$ by applying `tw-copath` to a set of “reduced instances” generated via kernelization and a branching procedure[2] `deg-branch`.

3.2 Preliminaries

3.2.1 Treewidth

Our `tw-copath` algorithm in Section 3.3 uses dynamic programming over a *tree decomposition*, and its running time depends on the related measure of *treewidth* [Rob86], which we denote $tw(G)$. Treewidth is a measure of how “tree-like" a graph is, with having small treewidth implying being closer to a graph. In order to comprehend the notion of treewidth, an understanding of tree decompositions is first necessary. Tree decompositions were originally associated with the graph minor theorem [Rob86].

3.2.2 Tree Decompositions

A *tree decomposition* [Rob84] of a graph $G(V,E)$ is a pairing of a tree T and a set of bags $X = \{X_1, ..., X_n\}$, where a bag is a subset of V. The nodes of T are exactly the bags in X. The following constraints must hold:

1. Every $v \in V$ must be in at least one bag
2. For every $uv \in E$, u and v must be in a common bag
3. For every bag b on a path from bag a to bag c in T, $a \cap c \in b$.

[2]The properties of our reduced instances guarantee we can find a tree decomposition of small width in poly(k) time.

Note that we assume our tree decompositions are rooted, which can be done by arbitrarily selecting a bag to be the root. The *width* of a tree decomposition is the size of the largest bag (in terms of number of vertices in the bag) minus 1. *Treewidth* of a graph G is the minimum width of a tree decomposition of G.

We utilize an enhanced version of tree decompositions, *nice tree decompositions* [Klo94]. In this paper, a *nice tree decomposition* is a rooted tree decomposition where each node of the tree is one of five specific types:

- **Leaf node**: Has no children in the tree and contains no vertices.
- **Introduce vertex node**: Has one child and contains exactly the same vertices as the child plus an additional vertex v. Such a bag introduces v.
- **Introduce edge node**: Has one child and contains exactly the same vertices as the child. They are labeled with an edge $uv \in E$, where u and v are contained in the bag. This node is said to introduce uv. We require each edge in E is introduced exactly once.
- **Forget vertex node**: Has one child and contains exactly the same vertices as the child, with the exception of one single vertex w. Such a bag forgets w.
- **Join node**: Contains the exact same vertices as it's two children, y and z, who contain the exact same vertices as each other.

Note that by the definition of *tree decomposition*, once a vertex is forgotten, it will not be found further up in the tree. Thus, a vertex is forgotten exactly once. Additionally, we enforce that the root node is of type "forget vertex" (and thus has an empty bag).

Transitioning from a tree decomposition to a nice tree decomposition does not increase treewidth, as we can carry out the transformation by simply adding intermediate nodes between the bags of the original tree decomposition. This transition can be done in polynomial time with respect to the size of the input graph [Cyg11a]. Using nice tree decompositions simplifies finding a dynamic programming algorithm which computes values over the tree decomposition, as there are a finite number of cases to consider.

While explaining our dynamic programming algorithm we also use Iverson's bracket notation. That is if p is a predicate we let $[p]$ be 1 if p is true and 0 otherwise [Cyg11a]. We view the value of $[p]$ as a multiplicative factor when it precedes an equation or expression. We also use the shorthand $f[x \to y]$ to denote updating a function f so that $f(x) = y$ and all other values are unchanged.

3.2.3 Fast Subset Convolution

We use fast subset convolution [Bjö07] to reduce the complexity of handling join nodes in the nice tree decomposition (Section 3.3). This technique maps functions of the vertices in a join bag to values in $\mathbb{Z}_p = \mathbb{Z}/p\mathbb{Z}$ (where p is chosen based on the application). The key complexity result we

rely on uses the $\mathbb{Z}_p$ product, which is defined below. We write $\mathbb{Z}_p^B$ for the set of all vectors t of length $|B|$ assigning a value $t(b) \in \mathbb{Z}_p$ to each element of $b \in B$.

Definition 2 ($\mathbb{Z}_p$ product)**.** *Let $p \geq 2$ be a fixed integer and let B be a finite set. For $t_1, t_2, t \in \mathbb{Z}_p^B$ we say that $t_1 + t_2 = t$ if $t_1(b) + t_2(b) = t(b)$ (in $\mathbb{Z}_p$) for all $b \in B$. For a ring R and functions $f, g : \mathbb{Z}_p^B \to R$, define the $\mathbb{Z}_p$ product, $*_x^p$ as*

$$(f *_x^p g)(t) = \sum_{t_1 + t_2 = t} f(t_1) g(t_2).$$

Fast subset convolution guarantees that certain $\mathbb{Z}_p$ products can be computed quickly.

Lemma 1 (Cygan et al. [Cyg11a])**.** *Let $R = \mathbb{Z}$ or $R = \mathbb{Z}_q$ for some constant q. The $\mathbb{Z}_4$ product of functions $f, g : \mathbb{Z}_4^B \to R$ can be computed in $4^{|B|}|B|^{O(1)}$ time and $4^{|B|}|B|^{O(1)}$ ring operations.*

3.3 An $O^*(4^{tw(G)})$ Algorithm via Cut&Count

We start by giving an fpt algorithm for Co-Path Set parameterized by treewidth. Our primary tool is the Cut&Count framework, which enables $c^{tw} n^{O(1)}$ one-sided Monte Carlo algorithms for connectivity-type problems with constant probability of a false negative. Cut&Count has previously been used to improve the best-known bounds for several well-studied problems, including Connected Vertex Cover, Hamiltonian Cycle, and Feedback Vertex Set [Cyg11a]. Pilipczuk showed that an $O^*(c^{tw})$ algorithm for some constant c can be designed with the Cut&Count approach for Co-Path Set because the problem can be expressed in the specialized graph logic known as ECML+C [Cyg11b]. However, since our end goal is to improve on existing algorithms for k-Co-Path Set in general graphs using a bounded treewidth kernel, we need to develop a specialized dynamic programming algorithm with a small value of c. We show:

Theorem 2. *There exists a one-sided fpt Monte Carlo algorithm* `tw-copath` *deciding k-Co-Path Set for all k in a graph G in $O^*(4^{tw(G)})$ time with failure probability $\leq 1/3$, when a tree decomposition of width tw is given as input.*

The Cut&Count technique has two main ingredients: (i) an algebraic approach to counting which uses arithmetic in $\mathbb{Z}_2$ (enabling faster algorithms) alongside a guarantee that undesirable objects are seen an even number of times (so a non-zero result implies a desired solution has been seen), and (ii) the idea of defining the problem's connectivity requirement through consistent cuts. In this context, a *consistent cut* is a partitioning (V_1, V_2) of the vertices of a graph into two sets such that no edge uv has $u \in V_1$ and $v \in V_2$ and all vertices of degree 0 are in V_1 . Since each connected component must lie completely on one side of any consistent cut, we see that a graph G has exactly $2^{cc(G) - n_I(G)}$ such cuts, where $cc(G)$ is the number of connected components and $n_I(G)$ is the number of isolates (vertices with degree 0). In order to utilize parity with the number of consistent cuts, we introduce *markers*, which create even numbers of consistent cuts for graphs that are not path forests. Our counting algorithm `tw-copath`, which computes the parity of the size of the collection

of subgraphs with consistent cuts which adhere to specific properties pertaining to CO-PATH SET, employs dynamic programming over a nice tree decomposition. We further use weights and the Isolation Lemma to bound the probability of a false negative arising from multiple valid markings of a solution. We use fast subset convolution [Bjö07] to reduce the complexity required for handling join bags in the dynamic programming. In the remainder of this section, we present the specifics for applying these techniques to solve CO-PATH SET.

In order to provide intuition, consider the following example; a graph with two connected components which are both C_4s (cycles on four vertices). For simplicity, we ignore the fact that such components would be handled by a kernelization step in our approach. Here, if $k = 2$, it is clear that a solution exists (delete one edge from each cycle), but the answer is no when $k = 1$. We will point back to this example throughout the following sections.

3.3.1 Cutting

We first provide formal definitions of markers and marked consistent cuts, which we use to ensure that path forests are counted exactly once during our dynamic programming.

Definition 3. *A triple (V_1, V_2, M) is a **marked consistent cut** of a graph G if (V_1, V_2) is a consistent cut and $M \subseteq E(G[V_1])$. We refer to the edges in M as the **markers**. A marker set is **proper** if it contains at least one edge in each connected component of G which is not an isolate.*

Note that if a marked consistent cut contains a proper marker set, all vertices are on the V_1 side of the cut. This is because by the definition of a consistent cut, all isolates are on the V_1 side, and if every connected component contains a marker then all connected components must fall entirely on the V_1 side as well. Therefore for any proper marker set there exists exactly one consistent cut, while all marker sets which are not proper will be paired with an even number of consistent cuts because unmarked components may lie in V_1 or V_2. We use proper marker sets to distinguish desired subgraphs by assigning markers in such a way that when we prune the dynamic programming table for solutions (as described later in the section), the only subgraphs we consider which may have a proper marker set are path forests. We know because the marker set is proper that the subgraph has a unique consistent cut, and thus these path forests will only be counted once in some entry of the dynamic programming table, while all other subgraphs will be counted an even number of times. Note that we are not claiming that all path forests will have proper marker sets. From our earlier example, if $k = 2$, there will be a setting where we assign one marker to each connected component and there is only consistent cut (both components in V_1). If $k = 1$, we will only assign a marker to one component and the second could exist on either side of the cut, and thus there are two consistent cuts in this setting. This is what allows us to do our counting modulo two, as odd total counts imply the existence of a solution.

We refer to the complement of a co-path set (the edges in the disjoint paths) as a *cc-solution*, and call it a *marked-cc-solution* when paired with a proper marker set of size exactly equal to its number of non-isolate connected components. In our example, the cc-solution consists of two disjoint

four-vertex paths, and is marked when the proper marker set has size two. While cc-solutions can be viewed as solutions due to their complementary nature, being marked is crucial in our counting algorithm and thus subgraphs which are marked-cc-solutions are what correspond to solutions in the dynamic programming table.

We now describe our use of the Isolation Lemma, which guarantees we are able to use parity to distinguish solutions. Let $f(X)$ denote $\sum_{x\in X} f(x)$.

Isolation Lemma 1 ([Mul87])**.** *Let $\mathscr{F} \subseteq 2^U$ be a non-empty set family over universe U. A function $\omega\colon U \to \mathbb{Z}$ is said to isolate $\mathscr{F}$ if there is a unique $S \in \mathscr{F}$ with $\omega(S) = \min_{F\in\mathscr{F}} \omega(F)$. Assign weights $\omega\colon U \to \{1,2,...,N\}$ uniformly at random, where the value of N is of the reader's choice. Then the probability that ω isolates $\mathscr{F}$ is at least $1-|U|/N$.*

Intuitively, if $\mathscr{F}$ is the set of solutions (or complements of solutions) to an instance of Co-Path Set and $|\mathscr{F}|$ is even, then `tw-copath` would return a false negative. This is because while each solution is counted an odd number of times in `tw-copath`, because there are an even number of solutions, the total count of solutions is even, making the combined count of solutions and non-solutions even and the algorithm would incorrectly determine a solution does not exist (a false negative). From our earlier example, there are sixteen valid solutions when $k = 2$ as we could remove one edge from each cycle, pairwise. Thus, while each solution would be counted once, our total count would still be even. The Isolation Lemma allows us to partition $\mathscr{F}$ based on the weight of each solution (as assigned by ω), and guarantees at least one of the partition's blocks has odd size with constant probability. We let U contain two copies of every edge $e \in E$: one representing e as a marker and one as an edge in the cc-solution. Then 2^U denotes all pairs of edge subsets (potential marked-cc-solutions), and we set $N = 3|U| = 6E$ (selected to achieve success probability in Theorem 1). Each copy of an edge is assigned a weight in $[1,N]$ uniformly at random by ω and the probability of finding an isolating ω is thus $2/3$. We denote the values assigned by ω to the set of marker copies by ω_M, and likewise to the set of edge-in-cc-solution copies by ω_E.

3.3.2 Counting

A marked-cc-solution C of a graph G corresponds to a co-path set of size k when the number of edges and markers in C match specific values which depend on k and $|E(G)|$. These values are easily deduced because we know the deletion of a co-path set solution of size k will leave $|E(G)|-k$ edges in a cc-solution. Furthermore, because a forest has $n-m$ connected components, the number of markers in C needs to be at most $|V(C)|-|E(G)|+k$. All isolates from a forest can be removed and the resulting graph is still a forest, and thus the actual number of markers necessary in C is $|V(C)|-n_I(C)-|E(G)|+k$. From our earlier example, the number of edges when $k=2$ is $|E(G)|-k = 8-2 = 6$ and the number of markers necessary is $|V(C)|-n_I(C)-|E(G)|+k = 8-0-8+2 = 2$, which we point out is the number of connected components in the aforementioned cc-solution.

We now describe a dynamic programming (DP) algorithm over a nice tree decomposition which returns modulo two the number of appropriately sized marked-cc-solutions in the root's subtree (for

a fixed k). Since no-instances have no appropriately sized marked-cc-solutions, and yes-instances have at least one, odd parity for the number of marked-cc-solutions of size corresponding to k implies a solution to the k-Co-Path Set instance must exist.

During the DP algorithm we actually count (for all values (m,e)) the number of *cc-candidates*, which are subgraphs $G' \subseteq G$ with maximum degree 2, exactly e edges, and a marked consistent cut with m markers. The following lemma justifies counting cc-candidates in place of marked-cc-solutions. Note that the weight of a marked-cc-solution or a cc-candidate is equal to the sum of its marker weights and its edge weights.

Lemma 2. *The parity of the number of marked-cc-solutions in G with e edges and weight w is the same as the parity of the number of cc-candidates $G' \subseteq G$ with e edges, $|V(G')|-e-n_I(G')$ markers, and weight w.*

Proof. Consider a subgraph $G' \subseteq G$ with maximum degree 2 and e edges. Let M' be a marking of G' such that $\omega_E(E(G'))+\omega_M(M')=w$. Assume first that G' is a collection of paths. We know that G' has $|V(G')|-e-n_I(G')$ non-isolate connected components. If M' is a proper marker set of G', then $|M'|=|V(G')|-e-n_I(G')$ and (G',M') has exactly one consistent cut. Therefore (G',M') contributes one to both the number of marked-cc-solutions and the number of cc-candidates, respectively.

If otherwise M' is not a proper marker set, then (G',M') contains an unmarked connected component and has an even number of consistent cuts, and therefore contributes an even number to the count of cc-candidates and zero to the number of marked-cc-solutions. Finally, if G' contains at least one cycle then $cc(G') > |V(G')|-e-n_I(G')$. Therefore at least one connected component does not contain a marker, and the number of consistent cuts is even, so the contribution to the count of cc-candidates is again even and the contribution to the count of marked-cc-solutions is zero. We conclude that the parity of the number of marked-cc-solutions and the parity of the number of cc-candidates is the same. □

Our dynamic programming algorithm is a bottom-up approach over a nice tree decomposition. We build cc-candidates for all values of m and e (encoding the option to add/not add edges and select/not select edges as markers), and keep track of various parameters ensuring that when pruning the DP table we only consider cc-candidates which could be valid solutions to the k-Co-Path Set instance. We use the number of edges to ensure our solution is of the correct size, and the number of markers and non-isolate vertices to determine when a subgraph is acyclic. The weight parameter allows us to distinguish between solutions and decreases the likelihood of a false negative occurring via the Isolation Lemma.

Finally, we need a parameter that encodes the degree information required to properly combine cc-candidates as we iterate up the tree. We call this parameter a *degree-function* and define it on the vertices V of a bag as $f : V \to \Sigma = \{0, 1_1, 1_2, 2\}$, where $f(v)$ corresponds to v's degree in the associated cc-candidates of the table entry — for vertices of degree 1, their value 1_j denotes which side of the partition (V_1, V_2) they are on. Vertices with degree 0 are on the V_1 side of the cut by definition and degree 2 vertices cannot gain additional incident edges, so we need not keep track of their side of the

Table 3.1 Dynamic programming table parameters and upper bounds.

Variable	Parameter	Maximum value
a	# of non-isolated vertices	n
e	# of edges	n^2
m	# of markers	n^2
w	weight of edges and markers	$4n^4$

cut. In summary, we have table entries $A_x(a,e,m,w,s)$ counting the number of cc-candidates with a non-isolated vertices, e edges, m markers, weight w, and degree-function s, where all vertices which have been introduced in the subtree rooted at x are present and only edges which have been introduced in this subtree may be present.

In the following description of the dynamic programming algorithm over a nice tree decomposition T, we let z_1, z_2 denote the children of a join node; otherwise, the unique child is denoted y.

Leaf:

$$A_x(0,0,0,0,\emptyset) = 1;\ A_x(a,e,m,w,s) = 0 \text{ for all other inputs.}$$

Introduce vertex v:

$$A_x(a,e,m,w,s[v \to 0]) = A_y(a,e,m,w,s);\ A_x(a,e,m,w,s[v \to i]) = 0,\ \forall i \neq 0.$$

Introduce edge uv:

$$\begin{aligned} A_x(a,e,m,w,s) = A_y(a,e,m,w,s) &+ \sum_{\substack{\alpha_t \in \mathit{subs}(s(t)) \\ t \in \{u,v\}}} [\![\phi_2(\alpha_u,\alpha_v)]\!] A_y(a',e-1,m,w',s') \\ &+ \sum_{\substack{\alpha_t \in \mathit{subs}(s(t)) \\ t \in \{u,v\}}} [\![\phi_1(\alpha_u,\alpha_v)]\!] \Big(A_y(a',e-1,m,w',s') + A_y(a',e-1,m-1,w'',s') \Big), \end{aligned}$$

where $\phi_j(\alpha_u,\alpha_v) = (\alpha_u = 1_j \vee s(u) = 1_j) \wedge (\alpha_v = 1_j \vee s(v) = 1_j)$, $a' = a - (|\{1_1,1_2\} \cap \{s(u),s(v)\}|)$, $w' = w - \omega_E(uv)$, $w'' = w - \omega_E(uv) - \omega_M(uv)$, $s' = s[u \to \alpha_u, v \to \alpha_v]$, and the *subs* function returns all the values the degree-function in child node y could have assigned to vertices u and v based on current degree-function s (summarized below).

$s(v)$	0	1_1	1_2	2
$\mathit{subs}(s(v))$	$\emptyset$	0	0	$\{1_1,1_2\}$

We now argue this formula's correctness. The term $A_y(a,e,m,w,s)$ handles the case when uv is excluded from the cc-solution. We handle the case when uv is added to the cc-solution by iterating over all possible *subs* values for each endpoint, only considering counts in child y's entries where u

and v have the appropriate *subs* values (preventing us from ever having a vertex with degree greater than 2). Note that we use the ϕ_j function to guarantee that if s labels u or v as an isolate, we do not use the introduced edge. We have a summation for both possible j values in order to consider uv falling on either side of the cut. The formulation of a' assures that each endpoint of degree 1 is now included in the count of non-isolates (i.e. when u and/or v had degree 0 in y). We utilize the marker weight of uv to distinguish when we choose it as a marker (only if on V_1 side of cut), and increment m accordingly. In either case, we update w appropriately (with w' if no marker, w'' if marker introduced).

Forget vertex h:

$$A_x(a,e,m,w,s) = \sum_{\alpha \in \{0,1_1,1_2,2\}} A_y(a,e,m,w,s[h \to \alpha]).$$

As a forgotten vertex can have degree 0, 1 or 2 in a cc-candidate, we must consider all possible values that s assigns to h in child bag y. Note that cc-candidates in which h is both not an isolate and not a member of a connected component that contains a marker will cancel modulo two, as h can be on either side of the cut and all parameters will be identical.

Join:

We compute A_x from A_{z_1} and A_{z_2} via fast subset convolution [Bjö07] taking care to only combine table entries whose degree-functions are *compatible*, ensuring that only joins which preserve the constraints of the degree-functions of the children nodes occur. For example, if a vertex v has been assigned to different sides of the cut in A_{z_1} and A_{z_2}, or a total degree greater than two over A_{z_1} and A_{z_2}, then the degree-functions are incompatible because v cannot be on both sides of the cut or have total degree greater than two. Similarly, the value it is assigned by the degree-function in A_x must accurately correspond to the values in A_{z_1} and A_{z_2}.

Definition 4. *At a join node x with children z_1 and z_2, the degree-functions s_1 from A_{z_1}, s_2 from A_{z_2}, and s from A_x are **compatible** if one of the following holds for every vertex v in x: (i) $s_i(v) = 0$ and $s_l(v) = s(v), i \neq l$ or (ii) $s_1(v) = s_2(v) = 1_j$ and $s(v) = 2$ for $i, j, l \in [1,2]$.*

In order to apply Lemma 1, we let B be the bag at x, and transform the values assigned by the degree function s to values in $\mathbb{Z}_4$. Let $\phi : \{0, 1_1, 1_2, 2\} \to \mathbb{Z}_4$ and $\rho : \{0, 1_1, 1_2, 2\} \to \mathbb{Z}$ be defined as in the table below, extending to vectors by component-wise application.

	0	1_1	1_2	2
ϕ	0	1	3	2
ρ	0	1	1	2

We use ϕ to apply Lemma 1, while the function ρ (which corresponds to a vertex's degree) is used in tandem to ensure the compatibility requirements are met: if $\phi(s_1) + \phi(s_2) = \phi(s)$, then necessarily $\rho(s_1) + \rho(s_2) \geq \rho(s)$. From the above table it is easy to verify that $\phi(s_1) + \phi(s_2) = \phi(s)$ and $\rho(s_1) + \rho(s_2) = \rho(s)$ together imply that s_1, s_2 and s are compatible. We sum over both functions when computing values for join nodes, to make sure that solutions from the children are combined only when there is compatibility.

Assign $t_1 = \phi(s_1)$, $t_2 = \phi(s_2)$, and $t = \phi(s)$ in accordance with Lemma 1. Let $\rho(s) = \sum_{v \in B} \rho(s(v))$; that is $\rho(s)$ is the sum of the degrees of all the vertices in the join node, as assigned by degree-function s. By defining functions f and g as follows:

$$f^{\langle d,a,e,m,w\rangle}(\phi(s)) = [\![\rho(s) = d]\!] A_{z_1}(a,e,m,w,s),$$
$$g^{\langle d,a,e,m,w\rangle}(\phi(s)) = [\![\rho(s) = d]\!] A_{z_2}(a,e,m,w,s),$$

and writing $\vec{r}_i$ for the vector $\langle d_i, a_i, e_i, m_i, w_i\rangle$ in order to consider all ways to split the parameter values of x between the two children nodes, we can now compute

$$A_x(a,e,m,w,s) = \sum_{\vec{r}_1 + \vec{r}_2 = \langle \rho(s), a', e, m, w\rangle} (f^{\vec{r}_1} *_x^4 g^{\vec{r}_2})(\phi(s))$$

where $a' = a + |s_1^{-1}\{1_1, 1_2\} \cap s_2^{-1}\{1_1, 1_2\}|$. We point out that

$$\sum_{\vec{r}_1 + \vec{r}_2 = \langle \rho(s), a', e, m, w\rangle} (f^{\vec{r}_1} *_x^4 g^{\vec{r}_2})(\phi(s)) = 1$$

only if both $\phi(s_1) + \phi(s_2) = \phi(s)$ and $\rho(s_1) + \rho(s_2) = \rho(s)$; that is, exactly when s_1, s_2 and s are compatible.

We conclude this section by describing how we search the DP table for marked-cc-solutions at the root node r. By Lemma 2, the parity of the number of marked-cc-solutions with $|E| - k$ edges and weight w is the same as the parity of the number of cc-candidates G' with $|E| - k$ edges, $|V(G')| - (|E| - k) - n_I(G')$ markers and weight w. These candidates are recorded in the table entries $A_r(a, |E| - k, a - |E| + k, w, \emptyset)$, where a is the number of non-isolates. Therefore, if there exists some a and w so that $A_r(a, |E| - k, a - |E| + k, w, \emptyset) = 1$, then we have a yes-instance of k-Co-Path Set. Note that the degree-function is $\emptyset$ in this entry because there are no vertices contained in the root node by definition.

By Lemma 1, the time complexity of `tw-copath` for a join node B is $O^*(4^{|B|})$, which is $O^*(4^{tw})$. Note that for the other four types of bags, as we only consider one instance of s per table entry, the complexity for each is $O^*(4^{tw})$. We point out that the size of the table is polynomial in n because there are a linear number of bags and a polynomial number of entries (combinations of parameters) for each bag. Since the nice tree decomposition has size linear in n, the bottom-up dynamic programming runs in total time $O^*(4^{tw})$. This complexity bound combined with the correctness of `tw-copath` discussed above proves Theorem 2.

3.4 Achieving $O^*(1.588^k)$ in General Graphs

In order to use `tw-copath` to solve k-Co-Path Set in graphs with unbounded treewidth, we combine kernelization and a branching procedure to generate a set of *reduced instances* – bounded treewidth

subgraphs of the input graph G. Specifically, we begin by constructing a kernel of size at most $6k$ as described in [Fen15]. Our reduced instances are bounded degree subgraphs of the kernel given by a branching technique. We prove that (1) at least one reduced instance is an equivalent instance; (2) we can bound the number of reduced instances; and (3) each reduced instance has bounded treewidth. Finally, we analyze the overall computational complexity of this process.

3.4.1 Kernelization and Branching

We start by describing our branching procedure `deg-branch` (Algorithm 1), which uses a degree-bounding technique similar to that of Zhang et al. [Zha14a]. Our implementation takes an instance (G,k) of Co-Path Set and two non-negative integers ℓ and D, and returns a set of reduced instances $\{(G_i, k-\ell)\}$ so that (1) each G_i is a subgraph of G with exactly $|E|-\ell$ edges and maximum degree at most D; and (2) at least one $(G_i, k-\ell)$ is an equivalent instance to (G,k). The size of the output (and hence the running time) of `deg-branch` depends on both input parameters ℓ and D. We will select D to achieve the desired complexity in `copath` in Section 3.4.3. We also make use of a budget parameter b, which keeps track of how many more edges can be removed per the constraints of ℓ (b is initially set to ℓ).

Our branching procedure leverages the observation that if a co-path set S exists, then every vertex has at most two incident edges not in S. Specifically, for every vertex of degree greater than D, we branch on pairs of incident edges which could remain after removing a valid co-path set (calling each pair a *candidate*), creating a search tree of subgraphs.

Algorithm 1: Generating reduced instances

```
1  Algorithm deg-branch(G, k, ℓ, D, b)
2      Let v be a vertex of maximum degree in G
3      if deg(v) ≥ D+1 and b ≥ D−1 then
4          Arbitrarily select vertices u_1, ... u_{D+1} from N(v)
5          R = ∅, E_v = {{v, u_i} | i ∈ [1, D+1]}
6          for e_1, e_2 ∈ E_v, e_1 ≠ e_2 do
7              E'_v = E_v \ {e_1, e_2}
8              R = R ∪ deg-branch(G \ E'_v, k, ℓ, D, b−(D−1))
9          return R
10     else if b = 0 and deg(v) ≤ D then return {(G, k−ℓ)}
11     else return ∅                                          // Discard G
```

Algorithm 1 returns a set of reduced instances which have had exactly ℓ edges removed. The size of the set is at most the number of leaves in the search tree of the branching process (inequality can result from the algorithm discarding branches in which the number of edits necessary to branch on a vertex exceeds the number of allowed deletions remaining). We now give an upper bound on the

Table 3.2 Numerically obtained constants c_d, $3 \leq d \leq 17$, used in Lemma 4; originally given in Table 6.1 of [Gas10].

d	3	4	5	6	7	8	9	10
c_d	0.1667	0.3334	0.4334	0.5112	0.5699	0.6163	0.6538	0.6847
d	11	12	13	14	15	16	17	
c_d	0.7105	0.7325	0.7514	0.7678	0.7822	0.7949	0.8062	

size of this set.

Lemma 3. *Let T be a search tree formed by* `deg-branch`(G, ℓ, D, k, b). *The number of leaves of T is at most* $\binom{D+1}{2}^{\ell/(D-1)}$.

Proof of Lemma 3. The number of children of each interior node of T is $\binom{D+1}{2}$, resulting in at most $\binom{D+1}{2}^{depth(T)}$ leaves. The depth of T is limited by the second condition of the `if` on line 3 of Algorithm 1. For each recursive call, b is decremented by $(D-1)$, until $b \leq D-1$. As b is initially set to ℓ, this implies $depth(T) \leq \ell/(D-1)$, proving the claim. □

Finally, we argue that at least one member of the set of reduced instances returned by `deg-branch` is equivalent to the original. Consider a solution F to k-Co-Path Set in the original instance (G, k). Every vertex has at most two incident edges in $G[E \setminus F]$, and since all candidates are considered at every high-degree vertex, at least one branch correctly keeps all of these edges.

3.4.2 Treewidth of Reduced Instances

Our algorithm `deg-branch` produces reduced instances with bounded degree; in order to bound their treewidth, we make use of the following result, which originated from Lemma 1 in [Fom09] and was extended in [Gas10].

Lemma 4. *For $\epsilon > 0$, there exists $n_\epsilon \in \mathbb{Z}^+$ s.t. for every graph G with $n > n_\epsilon$ vertices,*

$$tw \leq \left(\sum_{i=3}^{17} c_i n_i \right) + n_{\geq 18} + \epsilon n,$$

where n_i is the number of vertices of degree i in G for $i \in \{3, \ldots, 17\}$, $n_{\geq 18}$ is the number of vertices of degree at least 18, and c_i is given in Table 3.2. Moreover, a tree decomposition of the corresponding width can be constructed in polynomial time in n.

Since the structure of k-Co-Path Set naturally provides some constraints on the degree sequence of yes-instances, we are able to apply Lemma 4 to our reduced instances to effectively bound treewidth. We first find an upper bound on the number of degree-3 vertices in any yes-instance of k-Co-Path Set.

Lemma 5. *Let n_i be the number of vertices of degree i in a graph G for any $i \in \mathbb{Z}^+$, and Δ be the maximum degree of G. If (G,k) is a yes-instance of k-Co-Path Set, then $n_3 \leq 2k - (\sum_{i=4}^{\Delta}(i-2)n_i)$.*

Proof. Since (G,k) is a yes-instance, removing some set of $\leq k$ edges results in a graph of maximum degree 2. For a vertex of degree $j \geq 3$, at least $j-2$ incident edges must be removed. Thus, $n_3 + 2n_4 + 3n_5 + \ldots + (\Delta-2)n_\Delta \leq 2k$ (each removed edge counts twice – once for each endpoint). □

Lemma 6. *Let (G,k) be an instance of k-Co-Path Set such that G has n vertices and max degree at most $\Delta \in \{3,\ldots,17\}$. If (G,k) is a yes-instance, then the treewidth of G is upper bounded by $k/3+\epsilon n+c$, for any $\epsilon > 0$ and constant $c = n_\epsilon$ as defined in Lemma 4. A tree decomposition of the corresponding width can be constructed in polynomial time in n.*

Proof. Let n_ϵ be defined as in Lemma 4. Let G' be the graph formed by adding $N = n_\epsilon$ isolates to G. By Lemma 4, because G' has maximum degree at most Δ, $tw(G') \leq (1/6)n_3 + (1/3)n_4 + \ldots + c_\Delta n_\Delta + \epsilon(N+n)$. We can substitute the bound for n_3 from Lemma 5, which yields:

$$\begin{aligned} tw(G') &\leq \frac{2k - (\sum_{i=4}^{\Delta}(i-2)n_i)}{6} + \frac{n_4}{3} + \ldots + c_\Delta n_\Delta + \epsilon(N+n) \\ &\leq \frac{k}{3} + \epsilon(n+N). \end{aligned}$$

Note that the inequality holds because we can pair the negative terms of $(\sum_{i=4}^{\Delta}(i-2)n_i)/6$ with the corresponding terms of $n_4/3 + \ldots + c_\Delta n_\Delta$ and the value of $c_j n_j - (j-2)(n_j)/6$ is non-positive for all $j \in [4,17]$. Since $N = n_\epsilon$ is a constant, we have $tw(G') \leq k/3 + \epsilon n + c$. Since $G \subseteq G'$ and treewidth is monotone under subgraph inclusion, this proves the claim. □

We point out that when applying Lemma 6 to reduced instances, computing the desired tree decomposition is polynomial in k (since they are subgraphs of a $6k$-kernel).

3.4.3 The Algorithm `copath`

This section describes how we combine the above techniques to prove Theorem 1. As shown in Algorithm 2, we start by applying `6k-kernel` [Fen15] to find G', a kernel of size at most $6k$; this process deletes $k - k'$ edges. We then guess the number of edges $k_1 \in [0, k']$ to remove during branching, and use `deg-branch` to create a set of reduced instances Q_{k_1}, each of which have $k' - k_1$ edges. Note that `deg-branch` considers *all* possible reduced instances, and thus if a (cc-)solution exists, it is contained in at least one reduced instance. To ensure the complexity of finding the reduced instances does not dominate the running time, we set the degree bound D of the reduced instances to be 10 (any choice of $10 \leq D \leq 17$ is valid). By considering all possible values of k_1, we are assured that if (G,k) is a yes-instance, some Q_{k_1} contains a yes-instance. Each reduced instance is then passed to `tw-copath`, which correctly decides the problem with probability 2/3.

Algorithm 2: Deciding k-Co-Path Set

```
1 Algorithm copath (G,k)
2     (G', k') = 6k-kernel(G, k)
3     for k_1 ← 0 to k' do
4         Q_{k_1} = deg-branch(G', k', k_1, 10, k_1)
5         foreach (G_i, k_2) ∈ Q_{k_1} do
6             if tw-copath(G_i, k_2) then return true
7     return false
```

Proof of Theorem 1. We now analyze the running time of `copath`, as given in Algorithm 2. By Lemma 3, the size of each Q_{k_1} is $O(1.561^{k_1})$. For each reduced instance (G_i, k_2) in Q_{k_1}, we have $tw(G_i) \leq k_2/3 + \epsilon(6k) + c$ by Lemma 6.

Applying Theorem 2, `tw-copath` runs in time $O^*(4^{k_2/3+\epsilon 6k})$ for each reduced instance (G_i, k_2) in Q_{k_1} (with success probability at least 2/3). Each iteration of the outer `for` loop can then be completed in time

$$O^*(1.561^{k_1} 4^{k_2/3+\epsilon 6k}) = O^*(4^{k/3+\epsilon 6k}) = O^*(1.588^k),$$

where we use that $k_1 + k_2 = k' \leq k$, and choose $\epsilon < 10^{-5}$. Since this loop runs at most $k+1$ times, this is also a bound on the overall computational complexity of `copath`. Additionally `copath` is linear-fpt, as the kernelization of [Fen15] is $O(n)$, and the kernel has size $O(k)$, avoiding any additional poly(n) complexity from the `tw-copath` subroutine. Note that by Lemma 4 the tree decomposition can be found in polynomial time in the size of the reduced instance. Since reduced instances are subsets of $6k$-kernels, the linearity is unaffected because the graph has size polynomial in k. □

3.5 Conclusion

We give an $O^*(4^{tw})$ fpt algorithm for Co-Path Set. By coupling this with kernelization and branching, we derive an $O^*(1.588^k)$ linear-fpt algorithm for deciding k-Co-Path, significantly improving the previous best-known result of $O^*(2.17^k)$. We believe that the idea of combining a branching algorithm which guarantees equivalent instances with bounds on the degree sequence from the problem's constraints can be applied to other problems in order to obtain a bound on the treewidth (allowing treewidth-parameterized approaches to be extended to general graphs).

One natural question is whether similar techniques extend to the generalization of Co-Path Set to k-uniform hypergraphs (as treated in Zhang et al. [Zha14a]). It is also open whether the combined parameterization asking for a co-path set of size k resulting in ℓ disjoint paths is solvable in sub-exponential fpt time.

CHAPTER

4

MINING MAXIMAL INDUCED BICLIQUES USING ODD CYCLE TRANSVERSALS

Executive Summary

Many common graph data mining tasks take the form of identifying dense subgraphs (e.g. clustering, clique-finding, etc). In biological applications, the natural model for these dense substructures is often a complete bipartite graph (biclique), and the problem requires enumerating all maximal bicliques (instead of just identifying the largest or densest). The best known algorithm in general graphs is due to Dias et al., and runs in time $O(M|V|^4)$, where M is the number of *maximal induced bicliques* (MIBs) in the graph. When the graph being searched is itself bipartite, Zhang et al. give a faster algorithm where the time per MIB depends on the number of edges in the graph. In this work, we present a pair of new algorithms for enumerating MIBs in general graphs, whose run time depends on how "close to bipartite" the input is. Specifically, the runtime is parameterized by the size k of an odd cycle transversal (OCT), a vertex set whose deletion results in a bipartite graph. Our first algorithm runs in time $O(M|V||E|k^2 3^{k/3})$, which is an improvement on Dias et al. whenever $k \leq 3\log_3|V|$. The second runs in time $O(M|V||E|k)$, which is no worse than Dias et al. for all values of k. We implement both algorithms alongside a variant of Dias et al.'s in open-source C++ code [Hor19], and experimentally verify that the OCT-based approaches are faster in practice on graphs with a wide variety of sizes, densities, and OCT decompositions. Further, we identify different regimes where each of the new algorithms is dominant.

The implementation of these algorithms was done in collaboration with Kyle Kloster and Trey Woodlief. I designed the new enumeration algorithms and constructed the corresponding proofs of correctness and complexity independently. I communicated the algorithms to Kyle and Trey who led the implementation; I provided code review and verification of correctness throughout. This chapter was advised by Blair D. Sullivan and the results were published in the proceedings of the 2019 SIAM Conference on Discrete Mathematics (SDM) [Klo19] and the 2019 Special Event on Analysis of Experimental Algorithms (SEA^2) [Sul19].

4.1 Introduction

Bicliques (complete bipartite graphs) naturally arise in many data mining applications, including detecting cyber communities [Kum99], data compression [Aga94], epidemiology [Mus07], artificial intelligence [Wil82], and gene co-expression analysis [KU08; Kay11]. In many settings, the bicliques of interest are *maximal* (not contained in any larger biclique) and/or *induced* (each side of the bipartition is independent in the host graph), and there is a large body of literature [Ale04; Dia05; Epp94; Li07; Mak04; Mus07; San03; Zha14b] giving algorithms for enumerating all such subgraphs. Many of these approaches make strong structural assumptions on the host graph; the case when the host graph is bipartite has been particularly well-studied, and the `iMBEA` algorithm of Zhang et al. has been empirically established to be state-of-the-art [Zha14b]. In general graphs, the only known non-trivial algorithm for enumerating maximal induced bicliques (MIBs) is that of Dias et al. [Dia05] which does so in lexicographic order. We note that, as described, this method may fail to enumerate all MIBs (see Section 4.7).

We consider the problem of efficiently enumerating all MIBs in general graphs. In particular, we provide two novel approaches. First we give an algorithm `OCT-MIB` that extends ideas from `iMBEA` to work on non-bipartite graphs by using an *odd cycle transversal* (OCT set, see Section 2.4): a set of nodes O such that $G[V \setminus O]$ is bipartite. We prove that our algorithm has runtime $O(Mnm \cdot n_O^2 \cdot 3^{n_O/3})$ where $n_O = |O|$, M is the number of MIBs in the graph, and n and m denote the number of vertices and edges in the graph respectively. This is asymptotically faster than the approach of Dias et al. whenever the OCT set has size $n_O \leq 3\log_3(n)$. Since all graphs have OCT sets (although they can be size $O(n)$, as in cliques), our algorithm can be run in the general case; its correctness does not require minimality or optimality of O.

We also present `OCT-MIB-II` which again leverages odd cycle transversals to enumerate MIBs in time $O(Mnmn_O)$. In contrast to `OCT-MIB`, the worst-case runtime of `OCT-MIB-II` is not dependent on the number of MISs in O, making it better than `OCT-MIB` when $n_O \in \omega(1)$. Further, we give a more general algorithm for MIB-enumeration, `Enum-MIB`, which has runtime $O(Mnm)$. `OCT-MIB-II` and `Enum-MIB` stem from the same framework which we provide. `Enum-MIB` is essentially a modified version of the algorithm of Dias et al. [Dia05], which achieves a faster runtime by dropping the lexicographic output requirement. This algorithm is not optimized for near-bipartite graphs. We point out that while the worst-case theoretical running-times of `OCT-MIB` and `OCT-MIB-II` are

worse than Enum-MIB, we observe that in practice these approaches are much faster on near-bipartite graphs.

We additionally provide several additional algorithms which may be of independent interest. The first is LexMIB, a modification of the algorithm of Dias et al. [Dia05], which addresses the flaw in the original method and enumerates all MIBs in lexicographic order in time $O(Mn^4)$. The second is for a variant of biclique enumeration where we are only interested in bicliques with one part inside a specified independent set of the host graph. Specifically, given a graph $G = (V, E)$ with V partitioned into $X \cup Y$ with X an independent set, we wish to enumerate all *maximal crossing bicliques* (MCBs) $A \times B$ with $A \subseteq X$, $B \subseteq Y$. We give an algorithm MCB which enumerates all maximal crossing bicliques in time $O(|X||Y|m)$ per MCB.

Further, we implement OCT-MIB, OCT-MIB-II, LexMIB, and Enum-MIB in open source C++ code, and evaluate their performance on a suite of synthetic graphs with known OCT decompositions. Our experiments show that OCT-MIB-II and OCT-MIB greatly outperform their competitors in many settings, with OCT-MIB-II faster in a majority of near-bipartite cases.

We begin with preliminaries and a brief discussion of related work, then describe each of our new algorithms and provide proofs of their correctness and runtimes in Sections 4.3 and 4.4. We highlight several implementation details in Section 4.5, before presenting our experimental evaluation in Section 4.6. A description of LexMIB and the associated formal results are in Section 4.7.

4.2 Preliminaries

4.2.1 Related work

The complexity of finding bicliques is well-studied, beginning with the results of Garey and Johnson [Gar79] which establish that in bipartite graphs, finding the largest balanced biclique (where the two sides are of roughly equal order) is NP-hard but the largest biclique (number of vertices) can be found in polynomial time. Finding the biclique with the largest number of edges was shown to be NP-complete in general graphs [Yan78], but the case of bipartite graphs remained open for many years. Several variants (including the weighted version) were proven NP-complete in [Daw01], and in 2000, Peeters finally resolved the problem, proving the edge maximization variant is NP-complete in bipartite graphs [Pee03]. Particularly relevant to the mining setting, Kuznetsov showed that enumerating maximal bicliques in a bipartite graph is #P-complete [Kuz01], the NP-completeness analogue for counting problems [Val79].

For the problem of enumerating maximal *induced* bicliques, the best known algorithm in general graphs is due to Dias et al. [Dia05]; in the non-induced setting, other approaches include a consensus algorithm [Ale04], an efficient algorithm for small arboricity [Epp94], and a general framework for enumerating maximal cliques and bicliques [Gél09]. We note that as described, the method in [Dia05] may fail to enumerate all MIBs; we describe a graph eliciting this behaviour, along with a modified algorithm (LexMIB) with proof of correctness in Section 4.7. We note that our correction increases the runtime of the approach from $O(n^3)$ to $O(n^4)$ per MIB.

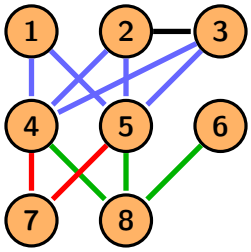

Figure 4.1 An example of a biclique which is not induced (blue edges), an induced biclique which is not maximal (red edges), and a maximal induced biclique (green edges). The blue edges do not form an induced biclique because vertices 2 and 3 share an edge. The red edges do not form a maximal biclique because vertex 8 is independent from vertex 7 and shares edges with vertices 4 and 5.

There has also been significant work on enumerating MIBs in bipartite graphs. We note that since all bicliques in a bipartite graph are necessarily induced (as all bicliques must have one side contained in one part and the other side contained in the second part), non-induced solvers for general graphs (such as [Ale04]) can be applied, and have been quite competitive. The best known approach, however, is an algorithm due to Zhang et al. [Zha14b] that directly exploits the bipartite structure[1]. Other approaches in the bipartite setting include frequent closed itemset mining [Li07] and transformations to the maximal clique problem [Mak04]; faster algorithms are known when a lower bound on the size of bicliques to be enumerated is assumed [Mus07; San03].

4.2.2 Notation and Terminology

Let $G = (V, E)$ be a graph. We denote $n = |V|$ and $m = |E|$, and use $N(v)$ to represent the neighborhood of a node $v \in V$. An independent set T in G is a *maximal independent set* (MIS) if T is not contained in any other independent set of G. We use $\mathscr{I}(S)$ to denote all nodes which are independent from a set S and $\mathscr{C}(S)$ to denote all nodes which are completely connected to a set S.

A biclique $A \times B$ in a graph $G = (V, E)$ consists of disjoint sets $A, B \subset V$ such that every vertex of A is connected to every vertex of B. We say a biclique $A \times B$ is *induced* if both A and B are independent sets in G. A biclique is *maximal* in G if no biclique in G properly contains it. See Figure 4.1 for examples of bicliques, induced bicliques, and maximal induced bicliques. Given a fixed independent set $X \subseteq V$, we define a *crossing* biclique with respect to X to be an induced biclique $A \times B$ such that $A \subseteq X$ and $B \subseteq V \setminus X$.

If G is bipartite, we write $G[L, R]$, where the vertices are partitioned as $V = L \cup R$ and we refer to the two partitions as the left and right "sides" of the graph. For a biclique $A \times B$ in $G[L, R]$, by convention we list the "left" set first, i.e. $A \subseteq L$ and $B \subseteq R$.

If G has OCT set O, we denote the corresponding OCT decomposition of G by $G[L, R, O]$, where the induced subgraph $G[L, R]$ is bipartite, and called the *bipartite part*. We write n_L, n_R, and n_O for

[1] Due to a typo in their runtime (see Section 4.8), the worst-case complexity of this algorithm is not an improvement on the $O(n^2)$ time per MIB of several other approaches. However, in practice, the experimental results in [Zha14b] support this being faster than [Ale04].

$|L|, |R|$, and $|O|$, respectively. We let $n_B = n_L + n_R$. Given an arbitrary vertex set $T \subseteq V$ we abbreviate $T^O = T \cap O$ and $T^{L,R} = T \setminus O$.

We present algorithms for enumerating maximal induced bicliques in two settings. The first setting, MAXIMAL INDUCED BICLIQUE ENUMERATION, is our primary focus and is what both `OCT-MIB` and `OCT-MIB-II` solve.

MAXIMAL INDUCED BICLIQUE ENUMERATION

Input: A graph $G = (V, E)$.
Output: All maximal induced bicliques in G.

The second setting, MAXIMAL CROSSING BICLIQUE ENUMERATION arises as a subproblem in our approach to MAXIMAL INDUCED BICLIQUE ENUMERATION via `OCT-MIB`.

MAXIMAL CROSSING BICLIQUE ENUMERATION

Input: A graph $G = (V, E)$; with V partitioned into $X \cup Y$ s.t. X is independent in G.
Output: All maximal bicliques $A \times B$ in G where $A \subseteq X$ and $B \subseteq Y$.

We now describe our algorithms for enumerating all MIBs in general graphs, both of which are designed for "near-bipartite" graphs. We first describe and prove the correctness of `OCT-MIB` before doing the same for `OCT-MIB-II`.

4.3 OCT-MIB

`OCT-MIB` enumerates all MIBs in a graph G by using an OCT decomposition $G[L, R, O]$ to drive a divide and conquer approach. Removing an OCT set from G enables use of efficient methods for enumerating MIBs in the bipartite setting, such as Zhang et al.'s `iMBEA` algorithm. Then a given MIB, $A \times B$, in the bipartite graph $G[L, R]$ can be checked for maximality in G by attempting to add vertices from O to $A \times B$.

Each MIB not found in $G[L, R]$ necessarily contains at least one vertex $v \in O$, allowing us to enumerate them by iterating over each vertex $v \in O$ and identifying all MIBs that contain v. This process requires careful bookkeeping that we organize using the observation that in all MIBs containing a given vertex v, one side of the biclique must be an independent set completely connected to v. Hence, we proceed with constructing "seed" bicliques from independent sets in $N(v)$. We then "grow" these into maximal bicliques by adding vertices from an MIS in O containing v.

In 4.3.1 we provide more detailed algorithm outlines for both `OCT-MIB` and `MCB`, highlighting key phases. We also provide high-level pseudo-code for `OCT-MIB` and describe *blueprints*, a shared data structure. In Section 4.3 we provide a complete description of `OCT-MIB` and do the same for `MCB` in Section 4.3.3.

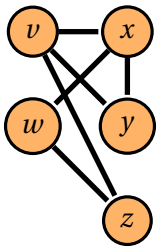

Figure 4.2 If v is the vertex we initialize with, our initial bicliques will be $\{v\} \times \{x, z\}$ and $\{v\} \times \{y, z\}$, noting that x and y cannot be on the same side of an induced biclique because they share an edge. When we expand these initial bicliques with w, the expanded bicliques are $\{v, w\} \times \{x, z\}$ and $\{v, w\} \times \{z\}$, with the latter being subsumed by the former.

4.3.1 Algorithm outline & data structure

In both `MCB` and `OCT-MIB` we take an independent set S and build bicliques $A \times B$ such that $S_I := S \cap (A \times B) \subseteq A$. We let $\bar{S_I} := A \setminus S_I$. During the ***initialization phase*** we find bicliques of the form $A \times B$ where A contains exactly one node from S. During this phase, we enumerate MISs in a subgraph using `MIS` [Tsu77]. As an example consider a vertex v and its neighbors x, y, z. If x and y share an edge while z is independent from both, then our initial bicliques will be $\{v\} \times \{x, z\}$ and $\{v\} \times \{y, z\}$ (see Figure 4.2). In `OCT-MIB`, we also utilize `MCB` as part of initialization.

We then "grow" the bicliques found during initialization in the ***expansion phase*** by repeatedly adding a node $w \in S \setminus A$ to S_I and removing nodes from B and $\bar{S_I}$ to ensure $A \times B$ is still an induced biclique. We refer to this process as *expanding with* w. Continuing our example from above, if vertex w was neighbors with x and z but not y, the "expanded bicliques" would be $\{v, w\} \times \{x, z\}$ and $\{v, w\} \times \{z\}$, the latter which would be subsumed by the former (see Figure 4.2). In `MCB` we let $S = X$, and in `OCT-MIB` we let each MIS in O be S once. While `MCB` only contains ***initialization*** and ***expansion phases***, `OCT-MIB` also contains the ***bipartite phase*** alluded to earlier, where the MIBs which are completely contained in $L \cup R$ are found. (Line 3 of Algorithm 3; this can be completed using `iMBEA`).

We employ an ordering ϕ on the vertices of S to limit the number of redundant expansions we make. We say $A \times B$ is *near-maximal* if $A \times B$ is maximal with respect to $V \setminus (S \setminus S_I)$ and there is some set of nodes $S' \subseteq S \setminus A$ such that $(A \cup S') \times B$ is a maximal biclique. We allow $S' = \emptyset$ so maximal bicliques are also near-maximal. We refer to near-maximal bicliques where all of the missing nodes occur later in ϕ as *future-maximal*, and we only expand on future-maximal bicliques (discarding all others). We group bicliques which contain the same set of S-nodes in *bags*. During the ***expansion phase*** we iterate over the bags, expanding their set of bicliques with all $x \in S$ such that $\phi(x) > \phi(y)$ for all $y \in S$ occuring in some biclique in the bag. For each S-node we expand with, we create a new bag to hold the new future-maximal bicliques. The general approach of `OCT-MIB` is outlined in Algorithm 3. The pseudo-code for `MCB` is similar but does not include a ***bipartite phase*** (lines 3 to 7) and only uses `MIS` in initialization. We now describe a key data structure used in both `OCT-MIB` and `MCB`.

Algorithm 3: `OCT-MIB` pseudo-code. Key phases: Bipartite (Lines 3–6), Initialization (Lines 11–19), and Expansion (Lines 20–32). Note that we require $V = L \cup R \cup O$, L, R, and O are pairwise disjoint, and L and R are both independent sets. `OCT-MIB` returns a list of the maximal induced bicliques in G.

```
1  Algorithm OCT-MIB(G = (V, E), L, R, O)
2      M ← ∅
3      B_1 ← maximal bicliques in G[L, R]
4      for b ∈ B_1 do
5          if b is maximal in G then
6              add b to M
7      I_O = MIS(O)
8      for S ∈ I_O do
9          T ← ∅ Fix φ an order of S
10         for v ∈ S do
11             B_2 ← unique initial bicliques via MCB & MIS
12             for b ∈ B_2 do
13                 if not-future-max(b, G) then
14                     remove b from B_2
15                 if maximal(b, G) then
16                     add b to M (and keep b in B_2)
17             if B_2 is not empty then
18                 add bag B_2 to T
19         while T is not empty do
20             Pick B ∈ T and let T ← T \ B
21             for v ∈ S: φ(v) > φ(w) ∀ w ∈ S ∩ B do
22                 B_3 ← ∅
23                 for biclique b ∈ B do
24                     b* ← expand b with v
25                     if not-future-max(b*, G) then
26                         continue
27                     B_3 ← B_3 ∪ {b*}
28                     if maximal(b*, G) then
29                         add b* to M (and keep b* in B_3)
30                 if B_3 is not empty then
31                     add B_3 to T
32     return M
```

DEFINITION. A **blueprint** is an octuple of sets $(S_I,\ IF_I,\ CC_I,\ S_W,\ S_P,\ IF_O,\ CC_O,\ O_{IF})$ and a specified node *next*. The sets $S_I, S_W, S_P \subseteq S$ satisfy

- $S_I \subseteq S$ contains the S-nodes in the biclique
- $S_W := \{x \in S \mid \phi(x) > \phi(w) \forall\, w \in S_I\}$
- $S_P := S \setminus (S_I \cup S_W)$.

The sets $IF_I, IF_O, CC_I, CC_O \subseteq V(G) \setminus S$ with

- $IF_I \subseteq \mathscr{I}(S_I)$, $IF_O := \mathscr{I}(S_I) \setminus IF_I$, $CC_I \subseteq \mathscr{C}(S_I)$, $CC_O := \mathscr{C}(S_I) \setminus CC_I$,

and the set $O_{IF} := (\mathscr{I}(S_I) \cap O) \setminus S$. Finally,

$$next := \operatorname*{argmin}_{s \in \{S_W \cap \mathscr{I}(IF_I) \cap \mathscr{C}(CC_I)\}} \phi(s).$$

By design, $(S_I \cup IF_I) \times CC_I$ is an induced biclique and is *represented* by the blueprint. We refer to blueprints and the bicliques they represent interchangeably in the description and discussion of the algorithms.

To guide the reader, we now describe the algorithmic roles of other sets in the blueprint. Within S, S_W is the set of nodes that are still candidates to be expanded with, S_P is the set of nodes that have been considered to be in a biclique with the current S_I elsewhere. We use IF_O and CC_O to check near-maximality, and O_{IF} to check global maximality in `OCT-MIB`. The vertex *next* is used to prevent expansions which produce non-future-maximal bicliques.

4.3.2 Algorithm Details

We begin by explaining checks for various properties of a blueprint, including `not-future-max`(b, G) and `maximal`(b, G). If the node we are expanding with is ordered later than *next*, then we are able to detect that the expansion will not yield a new blueprint. We say that a biclique is *invalid* if CC_I is empty. A blueprint b is *not future-maximal* if at least one of the following conditions is met; (i) $S_P \cap (\mathscr{C}(CC_I) \cap \mathscr{I}(IF_I)) \neq \emptyset$; (ii) $CC_O \cap (\mathscr{I}(CC_I) \cap \mathscr{C}(IF_I)) \neq \emptyset$; or (iii) $IF_O \cap (\mathscr{I}(IF_I) \cap \mathscr{C}(CC_I)) \neq \emptyset$. Note conditions (ii) and (iii) imply b is not near-maximal. Finally a blueprint b is *maximal* $(S_W \cup O_{IF}) \cap (\mathscr{C}(CC_I) \cap \mathscr{I}(IF_I)) = \emptyset$. We note that each of these checks can be done in $O(m)$ time.

Bipartite phase. Zhang et al. developed an algorithm to find all maximal induced bicliques in a bipartite graph in $O(\mathscr{B}mn)$ time where $\mathscr{B}$ is the number of maximal bicliques [Zha14b]. We run this algorithm on $G[L, R]$, and for each biclique found we check if an OCT node can be added to either side, which can be done in $O(m)$ time. If an OCT node cannot be added then we have found a MIB.

Initialization phase. Recall that `OCT-MIB` iterates over MISs in O; let $S = s_1, \ldots, s_{|S|}$ be the current MIS. For each s_i we create a new bag r_i which will hold blueprints. Let $S_I = \{s_i\}$, $S_W = \{s_{i+1}, \ldots, s_{|S|}\}$, $S_P = \{s_1, \ldots, s_{i-1}\}$ and $O_{IF} = O \setminus (N(s_i) \cup S)$. Now we find candidate blueprints in three rounds, adding the future-maximal ones to r_i.

In the first round, for each MIS $I \subseteq N(s_i)$, we check if any node from $\overline{N}_L(s_i) := L \setminus N(s_i)$ or $\overline{N}_R(s_i) := R \setminus N(s_i)$ is in $\mathscr{C}(I)$. If no node from either set is completely connected to I, we create a candidate blueprint where $CC_I = I$, $CC_O = N(s_i) \setminus I$, $IF_I = \emptyset$, $IF_O = (L \cup R) \setminus N(v_i)$, and S_I, S_W, S_P, and O_{IF} are defined as above. Otherwise we continue on to processing the next MIS in $N(s_i)$ (without creating a candidate blueprint for I).

In the second round, we run MCB on the subgraph induced on $\overline{N}_L(s_i) \cup (N(s_i) \setminus L)$ with $X = \overline{N}_L(s_i)$. Then for each returned MCB $A \times B$, we find the set of nodes $\overline{N}_R(s_i)^+ := \overline{N}_R(s_i) \cap (\mathscr{I}(A) \cap \mathscr{C}(B))$. We create a candidate blueprint with $CC_I = B$, $CC_O = N(s_i) \setminus B$, $IF_I = A \cup \overline{N}_R(s_i)^+$, $IF_O = (L \cup R) \setminus (N(v_i) \cup A \cup \overline{N}_R(s_i)^+)$, and S_I, S_W, S_P, and O_{IF} defined as above.

In the third round, we run MCB on the subgraph induced on $\overline{N}_R(s_i) \cup (N(s_i) \setminus R)$ with $\overline{N}_R(s_i)$ as the designated independent set. For each returned MCB $A \times B$, we check if $\overline{N}_L(s_i) \cap (\mathscr{I}(A) \cap \mathscr{C}(B)) = \emptyset$. If so, we create a candidate blueprint where $CC_I = B$, $CC_O = N(s_i) \setminus B$, $IF_I = A$, $IF_O = (L \cup R) \setminus (N(v_i) \cup A)$, and S_I, S_W, S_P, and O_{IF} are defined as above. Otherwise, we continue processing the next MCB (without creating a candidate blueprint for $A \times B$).

We check each candidate blueprint for future-maximality, adding it to r_i if true, and discarding it otherwise. If the blueprint is maximal we add $(S_I \cup IF_I) \times CC_I$ to the set of maximal bicliques and let $next = \infty$. If the blueprint is not maximal because an OCT node can be added to it we also let $next = \infty$; if it is not maximal because of an S-node later in the ordering we let *next* be the first such node from S_W. The blueprint remains in r_i in either case. As long as r_i is non-empty we add it to T, the set of bags to be processed in the ***expansion phase***.

Expansion phase. We now process bags from T until it is empty. We refer to removing a bag b from T and expanding on all of the blueprints in b with a vertex v as *branching on b with v*. In OCT-MIB S_I, S_W, S_P, and O_{IF} are the same in all blueprints in a given bag and we branch on a bag with all of the nodes in S_W in the order which matches the order of S; $w_1, \ldots, w_{|S_W|}$.

Let w_i be the node we are currently branching with and c_i be a bag we create to hold new blueprints formed by expanding with w_i. Let $P = (S_I,\ IF_I,\ CC_I,\ S_W,\ S_P,\ IF_O,\ CC_O,\ O_{IF},\ next)$ be the blueprint currently being expanded with. If we can detect that the expansion will not yield a new blueprint we terminate this expansion. Otherwise we create a new blueprint $P' = (S'_I,\ IF'_I,\ CC'_I,\ S'_W,\ S'_P,\ IF'_O,\ CC'_O,\ O'_{IF},\ next')$ as follows.

Let $S'_I = S_I \cup \{w_i\}$, $IF'_I = IF_I \setminus N(w_i)$, $CC'_I = CC_I \cap N(w_i)$, $S'_W = \{w_{i+1}, \ldots, w_{|S_W|}$, $S'_P = S_P \cup \{w_1, \ldots, w_{i-1}\}$, $IF'_O = IF_O \setminus N(w_i)$, $CC'_O = CC_O \cap N(w_i)$, and $O'_{IF} = O_{IF} \setminus N(w_i)$. If $(S'_I \cup IF'_I) \times CC'_I$ is invalid or it is not future-maximal, we terminate the expansion.

We must consider the case where the tuples (CC'_I, IF'_I) from different blueprints are identical after expansion. To handle this we maintain a hashtable for the current w_i being branched with, and hash each tuple (CC'_I, IF'_I) and terminate if we find a conflict. This hashtable can be discarded once we finish branching with the current node, w_i.

If the expansion has not been terminated then P' is future-maximal so we add P' to c_i. If the blueprint is maximal we add $(S'_I \cup IF'_I) \times CC'_I$ to the set of maximal bicliques and let $next' = \infty$. If the blueprint is not maximal solely because of nodes from O'_{IF} we also let $next' = \infty$, but if it is

not maximal because of a node from S'_W we let *next* be the first such node from S'_W. The blueprint remains in c_i in either case. Once again we continue the ***expansion phase*** until all bags have been branched upon.

4.3.3 MCB Algorithm

In this section we present the details of our algorithm MCB for finding all maximal crossing bicliques.

MCB Description

Given an instance (G, X, Y), MCB enumerates all maximal crossing bicliques. MCB makes use of the same checks as OCT-MIB as described in Section 4.3.

Initialization phase. Recall that $S = X$ in MCB. To begin this phase, we fix an order of S, $v_1, \ldots, v_{|X|}$. For each v_i we create a new bag r_i which will hold blueprints. We let $S_I = \{v_i\}$, $S_W = \{v_{i+1}, \ldots, v_{|X|}\}$, and $S_P = \{v_1, \ldots, v_{i-1}\}$. We then find all maximal independent sets in $G[N(v_i)]$ in $O(|Y|m)$ time per *MIS* [Tsu77]. For each MIS I we create a blueprint where $CC_I = I$, $CC_O = N(v_i) \setminus I$, and S_I, S_W, and S_P are defined as above. In MCB IF_I, IF_O, and O_{IF} are not used in any blueprints.

For each blueprint, if it is not future-maximal, we discard it; otherwise, we add it to r_i. If the blueprint is maximal we add $S_I \times CC_I$ to the set of maximal crossing bicliques and let $next = \infty$. If it is not maximal we let *next* be the first node from S_W which is completely connected to CC_I. The blueprint remains in r_i in either case. As long as r_i is non-empty we add it to T, the set of bags to be processed in the ***expansion phase***.

Expansion phase. Once we have initialized with each v_i, we process bags from T until it is empty. We refer to branching in the same manner as in Section 4.3. Note that S_I, S_W, and S_P are the same in all of the blueprints at bag b. We will branch on b with all nodes in S_W, which we order $x_1, \ldots, x_{|S_W|}$ to be consistent with the order of S. Let x_i be the node we are currently branching on b with and c_i be a bag we create to hold new blueprints formed by expanding with x_i. When branching with x_i, we iterate over the blueprints in b and expand on them one at a time. Whether an expansion is terminated or completed, we continue with expanding the next blueprint in b.

Let $P = (S_I,\ CC_I,\ S_W,\ S_P,\ CC_O,\ next)$ be the blueprint currently being expanded on. If we determine the expansion will not yield a future-maximal blueprint, we terminate. Otherwise we will create a new blueprint P' with values $(S'_I,\ CC'_I,\ S'_W,\ S'_P,\ CC'_O,\ next')$ as follows. We let $S'_I = S_I \cup \{x_i\}$, $CC'_I = CC_I \cap N(x_i)$, $S'_W = \{x_{i+1}, \ldots, x_{|S_W|}\}$, $S'_P = S_P \cup \{x_1, \ldots, x_{i-1}\}$, and $CC'_O = CC_O \cap N(x_i)$. If $S'_I \times CC'_I$ is invalid or it is not future-maximal we terminate this expansion.

We must consider the case where two CC'_I's from different blueprints are identical after expansion. To handle this we maintain a hashtable for the current x_i being branched with, and hash each CC'_I, terminating if we find a conflict. This hashtable can be discarded once we finish branching with the current node.

The blueprint P' is future-maximal if it has not been discarded, so we add it to c_i. If the blueprint is maximal we add $S'_I \times CC'_I$ to the set of maximal crossing bicliques and let $next = \infty$. If it is not maximal we let *next* be the first node from S'_W which is completely connected to CC'_I. P' remains in

c_i in either case. Once we have expanded on all blueprints from b we add c_i to T if it is non-empty, continuing to branch on any bag in T that has not yet been branched on.

4.3.4 Correctness & Complexity: OCT-MIB

We now establish the correctness and asymptotic complexity of OCT-MIB.

Theorem 3. OCT-MIB *finds all maximal induced bicliques.*

Proof. Suppose that there is a MIB $A \times B$ that OCT-MIB does not find. If there are no nodes from O in $A \cup B$ then the biclique would be found in the bipartite phase, thus we may assume there is a node from O in the biclique.

Without loss of generality we may assume that A contains OCT-nodes and that if A contains nodes from L or R then B contains nodes from the other. We let $A^O = A \cap O$ and $A^{L,R} = A \setminus A^O$. Note that A^O must be contained in some MIS S in O. Let a_1^O be the first node in A^O with respect to ϕ; we show that when we initialize with a_1^O, $B \subseteq CC_I$ and $A^{L,R} \subseteq IF_I$ for some blueprint in $r_{a_1^O}$. $A^{L,R}$ is either empty or contains nodes from one of $\{L, R\}$, and we show that a candidate blueprint as described above is created in both cases.

By definition, B must be contained within some MIS I_B in $N(a_1^O)$. If $A^{L,R}$ is empty and a candidate blueprint with $B \subseteq CC_I$ is not created in the first round, then there must be nodes in $\overline{N}_L(a_1^O)$ or $\overline{N}_R(a_1^O)$ which are in $\mathscr{C}(I_B)$. If a node $\bar{l}$ from $\overline{N}_L(a_1^O)$ is in $\mathscr{C}(I_B)$ then $\bar{l} \times I_B$ is a crossing biclique in the instance of MCB called in the second round. Thus a maximal crossing biclique which contains I_B is returned and a candidate blueprint with $B \subseteq CC_I$ is created. Otherwise a node $\bar{r}$ from $\overline{N}_R(a_1^O)$ is in $\mathscr{C}(I_B)$ and $\bar{r} \times I_B$ is a crossing biclique in the instance of MCB called in the third round, leading to the creation of a candidate blueprint with $B \subseteq CC_I$.

If $A^{L,R} \neq \emptyset$, assume without loss of generality that $A^{L,R} \subseteq L$. Then there is a crossing biclique $A^{L,R} \times B$ in the instance of MCB called in the second round, and a candidate blueprint with $B \subseteq CC_I$ and $A^{L,R} \subseteq IF_I$ is created.

If a candidate blueprint with $B \subseteq CC_I$ and $A^{L,R} \subseteq IF_I$ is pruned away because of an S_P node, then $A \times B$ is not a maximal biclique. So there exists a blueprint P^* in $r_{a_1^O}$ where $B \subseteq CC_I$ and $A^{L,R} \subseteq IF_I$. Expanding P^* so that $S_I = A$ would yield the biclique $A \times B$ as $(S_I \cup IF_I) \times CC_I$. Thus as long as we expand for each node in A we will find the biclique $A \times B$.

Assume that an expansion with a node in A is not made and let a_k^O be the first such node. The blueprint would not have been discarded because of CC_I' being empty or a node from S_P' or O_{IF}' being in $\mathscr{C}(CC_I') \cap \mathscr{I}(IF_I)$, as this would imply $A \times B$ is not a MIB. If a node from from CC_O' or IF_O' made P^* not near-maximal when expanding with the node prior to a_k^O in A, a_{k-1}^O, consider a blueprint Q^+ formed by adding nodes from IF_O' to IF_I' and CC_O' to CC_I' such that $(S_I' \cup IF_I') \times CC_I'$ forms a future-maximal biclique. This blueprint must exist at the bag created by expanding with a_{k-1}^O and it contains B in CC_I and $A^{L,R}$ in IF_I. We now let that blueprint be P^*.

Thus a_k^O must be greater than $next$ for blueprint P^* at the bag with $S_I = a_1^O, \ldots, a_{k-1}^O$. Because of how we constructed $next$, it is not in S_I. Therefore $next$ is not in A but because X is an inde-

pendent set, it is independent from A. Furthermore $next$ is completely connected to B because it is completely connected to CC_I and $B \subseteq CC_I$. Thus $A \times B$ would not be a maximal biclique and we obtain a contradiction. We can apply this argument inductively to show the complete correctness of the algorithm.

□

Theorem 4. *Given a graph G with OCT decomposition $G[L,R,O]$,* `OCT-MIB` *runs in $O(Mmnn_O^2 I_O)$ time, where M is the number of MIBs and $I_O \leq 3^{n_O/3}$ is the number of maximal independent sets in O.*

Proof. We note that G has no isolates, therefore n_L and n_R are $O(m)$. We first compute the complexity of finding a maximal biclique when iterating over a single MIS S in OCT. In the initialization phase, finding the MISs in round one takes $O(mn)$ time per MIS. Finding the MCBs in rounds two and three takes $O(mnn_O)$ per MCB. This is because the arboricity, the minimum number of forests into which its edges can be partitioned, of Y in each call to `MCB` is $O(n_O)$. Therefore the time it takes to initialize a blueprint is $O(mn_O)$ not $O(mn)$ [Chi85]. Because of how we have utilized $next$, each node in S is expanded with $O(1)$ times per MIB. Each expansion can be done in $O(m)$ time, and thus the total time spent expanding is $O(mn_O)$ per maximal biclique. Since we only expand on blueprints which are future-maximal, every expansion is accounted for.

There may be an additional $O(n_O)$ initializations of a blueprint. Thus the total complexity of finding a single maximal biclique when iterating over S is $O(mnn_O^2)$. A biclique can be found once per MIS in OCT, and thus the total complexity of `OCT-MIB` is $O(Mmnn_O^2 I_O)$, where M is the number of maximal bicliques and I_O is the number of MIS's in OCT.

□

4.3.5 Correctness & Complexity: `MCB`

We now establish the correctness and asymptotic complexity of `MCB`.

Theorem 5. `MCB` *finds all maximal crossing bicliques.*

Proof. Suppose there is a maximal crossing biclique $A \times B$, $A \subseteq S = X$, $B \subseteq Y$ that our algorithm does not find. Consider the ordering of A which is consistent with the ordering of S that we fixed upon initialization and let a_1 be the first node. Upon initialization B must be contained in an MIS in a_1's neighborhood. Let P^* be a blueprint which has this property. Applying the expansions which lead to $S_I = A$ to blueprint P^* yields $A \times B$ as $S_I \times CC_I$. Thus we must show that each of these expansions are made for some P^*.

Assume that an expansion with a node in A is not made and let a_k be the first such node. The blueprint would not have been discarded because of CC'_I being empty or a node from S'_P being completely connected to CC'_I, as this would imply $A \times B$ is not a maximal crossing biclique. If a node from CC'_O was completely independent from CC'_I when expanding with the node prior to a_k in A, a_{k-1}, consider a blueprint Q^+ formed by adding a maximal independent set of the CC'_O nodes

which can be added to CC'_I in Q. This blueprint must exist at the bag created by expanding with a_{k-1} and it contains B in CC_I. We now let Q^+ be P^*.

Thus a_k must be greater than $next(x)$, for blueprint P^* at the bag with $S_I = a_1, \ldots, a_{k-1}$. Because of how we constructed $next$, we know that x is not in S_I. Therefore x is not in A but because X is an independent set, it is independent from A. Furthermore x is completely connected to B because it is completely connected to CC_I and $B \subseteq CC_I$. Thus $A \times B$ would not be a maximal biclique and we obtain a contradiction. We can apply this argument inductively to show the complete correctness of the algorithm. □

Theorem 6. `MCB` *runs in* $O(M|X||Y|m)$ *on the instance* (G, X, Y)*, where* M *is the number of maximal crossing bicliques.*

Proof. First note that if a node in X has no neighbors in Y it will not be in any biclique so we can delete it, and vice versa. Therefore $|X|$ and $|Y|$ are $O(m)$.

We view the complexity of our algorithm through the lens of the amount of time it takes to find each maximal crossing biclique. First note that the time it takes to initialize a single blueprint and check it for maximality is $O(|Y|m)$, while the time it takes to expand a blueprint is $O(m)$. Now consider a maximal blueprint P which is found via expanding from a bag H. The bag H is formed by branching on a series of other bags, H's *ancestors*; $A_1, \ldots, A_t$. One could trace the expansions which led to P to find the corresponding blueprint in each ancestor bag. Let A_F be the closest ancestor bag of H where the corresponding blueprint is maximal if such a bag exists and let $A_F = A_1$ otherwise. Because of how we have utilized $next$, for each node v in X there is at most one total expansion of a corresponding blueprint from a bag in $\{A_F, \ldots, A_t\}$. Thus the total time spent expanding is $O(|X|m)$ per blueprint. Furthermore because we only expand future maximal blueprints every expansion is accounted for.

We note that there may be an additional $O(|X|)$ initializations of a blueprint, which impacts the complexity. Thus the total complexity of finding a single maximal crossing biclique is $O(|X||Y|m)$. □

4.4 `OCT-MIB-II`

In this section we provide a description of our second novel algorithm for solving MAXIMAL INDUCED BICLIQUE ENUMERATION, `OCT-MIB-II`. Like `OCT-MIB`, it is optimized for near-bipartite graphs. We additionally give a MIB-enumeration algorithm for general graphs, `Enum-MIB`. Both of these algorithms follow the same general framework, which we begin by describing.

4.4.1 Algorithm Framework

The framework uses two subroutines, `MakeIndMaximal` and `AddTo`. `MakeIndMaximal` takes in (C, S), where C is an induced biclique and $S \subseteq V$, and either returns a MIB C^+ where $C \subseteq C^+$, $C^+ \subseteq C \cup S$, $C \neq \emptyset$, or returns $\emptyset$. If it returns $\emptyset$ and $C \neq \emptyset$ then there is another MIB D which

contains C and $v \in (V \setminus S) \setminus C$. `AddTo` takes in (C, v) where $C = C_1 \times C_2$ is an induced biclique and $v \in V \setminus (C_1 \cup C_2)$, and returns the induced biclique where v is added to C_1, $N(v)$ is removed from C_1, and $\overline{N}(v)$ is removed from C_2 if $C_2 \cap N(v) \neq \emptyset$; otherwise, $\emptyset$ is returned. Both `MakeIndMaximal` and `AddTo` operate in $O(m)$ time. We include the algorithmic details and proofs of the complexity and correctness for these routines in Section 4.4.4.

The MIB-enumeration framework (shown in Algorithm 4) begins by finding a seed set of MIBs $\mathscr{C}_S$. At a high level, it operates by attempting to add vertices from the designated set I_S to previously found MIBs to make them maximal. We utilize a dictionary $\mathscr{D}$ to track which MIBs have already been found and a queue $\mathscr{Q}$ to store bicliques which have not yet been explored. We now prove two technical lemmas used to show the correctness of this framework.

Algorithm 4: MIB-enumeration algorithm framework

```
1  Framework MIB-Enumeration(G = (V, E), I_S)
2      C_S = FindSeedSet(G)
3      Add each C ∈ C_S to D and Q
4      while Q is not empty do
5          X × Y ← pop(Q)
6          for j ∈ I_S \ (X ∪ Y) do
7              C_1 = AddTo(X × Y, j)
8              C_1' = MakeIndMaximal(C_1, I_S)
9              if C_1' is not in D then
10                 Add C_1' to D and Q
11             C_2 = AddTo(Y × X, j)
12             C_2' = MakeIndMaximal(C_2, I_S)
13             if C_2' is not in D then
14                 Add C_2' to D and Q
15     return D
```

Lemma 7. *Let $X \times Y$ be a MIB in graph G which contains a non-empty subset of $R \times S$, another MIB in G. Running* `AddTo` *with parameters $X \times Y$ and $v \in R \setminus (X \cup Y)$ returns a biclique which contains $R \cap X$, $S \cap Y$, and v if $Y \cap N(v) \neq \emptyset$.*

Proof. By construction, v must be independent from R and completely connected to S. Thus, none of $R \cap X$ will be removed from X and all of $S \cap Y$ will remain in Y, as required. Therefore, as long as $Y \cap N(v) \neq \emptyset$, the desired biclique is returned. □

Lemma 8. *In Algorithm 4, if there exists a MIB $A' \times B'$ in $\mathscr{D}$ such that $A \setminus I_S \subseteq A'$, $B \setminus I_S \subseteq B'$ and $(A \cup B) \cap (A' \cup B') \neq \emptyset$, for each MIB $A \times B$ in G, then all MIBs in G are included in $\mathscr{D}$.*

Proof. Assume not. Let $A \times B$ be a MIB in G which is not in $\mathscr{D}$ with $|(A \cup B) \setminus I_S|$ maximum. Let $A' \times B'$ be

the MIB in $\mathscr{D}$ such that $A \setminus I_S \subseteq A'$, $B \setminus I_S \subseteq B'$ and $(A \cup B) \cap (A' \cup B') \neq \emptyset$ and let $v \in ((A \cup B) \setminus (A' \cup B')) \subseteq I_S$. Without loss of generality assume $B \cap B' \neq \emptyset$ and $v \in A$.

Consider the iteration of Algorithm 4 when $X \times Y = A' \times B'$ and $j = v$ (defined in lines 5 and 6). By Lemma 7, one of the calls to `AddTo` returns an induced biclique C which contains $A \setminus I_S$, $B \setminus I_S$, and v. Both sides of C are non-empty (since $B \cap B' \neq \emptyset$ and $v \in A$). If $C = A \times B$ we obtain a contradiction, as `MakeIndMaximal` (C, I_S) would return C, resulting in its addition to $\mathscr{D}$. Otherwise, either `MakeIndMaximal` returns $\emptyset$ or a biclique $C' = A' \times B'$ which is added to $\mathscr{D}$. Since both sides of C are nonempty, if `MakeIndMaximal` returns $\emptyset$, there exists a MIB in G containing C and $x \in (V \setminus I_S) \setminus C$. Let $A' \times B'$ be such a MIB; since it has more vertices in $V \setminus I_S$ than C, it must be in $\mathscr{D}$, and we set $C' = A' \times B'$. In either case, $C \subseteq (A' \cup B')$, and $|(A \cup B) \setminus (A' \cup B')| < |(A \cup B) \setminus (X \cup Y)|$. We can repeat this argument for the new $A' \times B'$, noting that $(A \cup B) \cap (A' \cup B')$ will include vertices on both sides. Thus, the argument still holds without any assumption on the non-empty side of the intersection and $|(A \cup B) \setminus (A' \cup B')|$ will strictly decrease; when it reaches 0, $A' \times B' = A \times B$, a contradiction. □

Note that as `MakeIndMaximal` only returns MIBs, this framework will only include MIBs in $\mathscr{D}$. Together with Lemma 8, this yields the following corollary.

Corollary 1. *If for every MIB $A \times B \in G$ there is a MIB $A' \times B' \in \mathscr{C}_S$ such that $A \setminus I_S \subseteq A'$, $B \setminus I_S \subseteq B'$ and $(A \cup B) \cap (A' \cup B') \neq \emptyset$, then upon completion of Algorithm 4, $\mathscr{D}$ will contain exactly the MIBs in G.*

Recall that `AddTo` and `MakeIndMaximal` each run in $O(m)$ time. Combining this with the fact that each MIB in G is popped at most once from $\mathscr{Q}$ we have:

Corollary 2. *The time complexity of this framework is $O(Mmn + INIT)$, where $INIT$ is the time needed by* `FindSeedSet` *to compute $\mathscr{C}_S$.*

4.4.2 `Enum-MIB`

We now present `Enum-MIB`, which follows the MIB-enumeration framework. To form $\mathscr{C}_S$, for each vertex $v \in V$ we run `MakeIndMaximal` $(\{v\} \times \{x\}, V)$ where $x \in N(v)$ and add it to $\mathscr{C}_S$. We also let $I_S = V$. To show the correctness of this approach, we note that $V \setminus V = \emptyset$ and any MIB contains the empty set. Thus all that remains to show is that for each MIB there is a MIB in $\mathscr{C}_S$ with which it has a non-empty intersection. As every $v \in V$ is in some MIB in $\mathscr{C}_S$, this condition is met. Thus, via Corollary 1, `Enum-MIB` will find all MIBs. There may be $O(n)$ duplicates in $\mathscr{C}_S$ which can be removed in $O(n)$ time per duplicate. As `MakeIndMaximal` runs in $O(m)$ time, by Corollary 2, the time complexity of `Enum-MIB` is $O(Mmn)$. We note that `Enum-MIB` is essentially a simplified version of the `LexMIB` algorithm from Section 4.7 which does not guarantee lexicographic order on output.

4.4.3 `OCT-MIB-II`

Next we describe `OCT-MIB-II`, an algorithm for enumerating all MIBs in a graph with a given OCT decomposition $G[L, R, O]$. `OCT-MIB-II` also makes use of the MIB-enumeration framework described in Section 4.4.1. In the calls to `MakeIndMaximal` we let $I_S = O$. To form $\mathscr{C}_S$, we begin by

running iMBEA [Zha14b] to find the set $\mathscr{C}_B$ of MIBs in $G[L \cup R]$. For each $C_B \in \mathscr{C}_B$ we run MakeIndMaximal on (C_B, O). This creates a set X_B of MIBs in G.

Then for each node $o \in O$, we find the set of MISs in $N(o)$. This can be done in $O(mn)$ time per MIS using the algorithm of Tsukiyama et al. [Tsu77]. For each MIS I_o found, run MakeIndMaximal on the induced biclique $\{o\} \times I_o$. Let the multiset of all MIBs produced by this process be denoted X_Q. Note that a MIB may be in X_Q up to $O(n_O)$ times (once per $o \in O$, stemming from an MIS in $N(o)$), but we can remove duplicates from X_Q in $O(n)$ per MIB, forming X'_M. We then let $\mathscr{C}_S = X_B \cup X'_M$. Thus, FindSeedSet runs in $O(mnn_O)$ per unique MIB found, and by Corollary 2, the total time complexity of OCT-MIB-II is $O(Mmnn_O)$.

To show the correctness of OCT-MIB-II, we must show that for every MIB B_i in G, we include a corresponding MIB $f(B_i)$ in $\mathscr{C}_S$ which includes all of B_i's non-OCT nodes and a node in B_i. If an entire MIB C is contained in O, then any MIB containing $\{o\} \times I_o$ for $o \in C$ suffices. If a MIB has non-OCT nodes on both sides, then there must be a MIB in X_B which contains these non-OCT nodes because there is a MIB in $G[L \cup R]$ containing them. If a MIB has all of its non-OCT nodes on one side, then there is an OCT node o which is neighbors with all of the non-OCT nodes, which thus must be contained in an MIS in $N(o)$. Thus, by Corollary 1, we find all of the MIBs in G.

4.4.4 Algorithm Subroutines

We now provide algorithmic details and proofs of the complexity and correctness of MakeIndMaximal and AddTo.

4.4.4.1 MakeIndMaximal

Recall that MakeIndMaximal takes in (C, S), where C is an induced biclique and $S \subseteq V$, and either returns a MIB C^+ where $C \subseteq C^+$, $C^+ \subseteq C \cup S$, $C \neq \emptyset$, or returns $\emptyset$. If it returns $\emptyset$ and $C \neq \emptyset$ then there is another MIB D which contains C and $v \in (V \setminus S) \setminus C$. We give pseudo-code for MakeIndMaximal in Algorithm 5.

Lemma 9. MakeIndMaximal *returns a MIB C^+ where $C \subseteq C^+$ and $C^+ \subseteq C \cup S$, or returns $\emptyset$.*

Proof. Referring to the pseudo-code in Algorithm 5, it is clear that $C \subseteq C^+$, as no vertices are ever removed from the input biclique C. Furthermore, the only vertices added to C^+ are from S, so $C^+ \subseteq C \cup S$ and C^+ is the only biclique returned by MakeIndMaximal. Note that neither side of C is empty and the only vertices added are independent from the side of the biclique which they are added to and completely connected to the other side, so if we do not return $\emptyset$ the object returned is an induced biclique. If no node from outside of S can be added to C^+, then we will not return $\emptyset$ and thus C^+ is maximal.

□

Lemma 10. *If* MakeIndMaximal *returns $\emptyset$ and $C \neq \emptyset$ then there is another MIB D in G which contains C and $v \in (V \setminus S) \setminus C$.*

Algorithm 5: `MakeIndMaximal`

```
1  Algorithm MakeIndMaximal(G = (V, E), C = C_1 × C_2, S)
2      Let C_S = S \ (C_1 ∪ C_2)
3      if C == ∅ then
4          return ∅
5      for v ∈ C_S do
6          if |N(v) ∩ C_1| == |C_1| & |N(v) ∩ C_2| == 0 then
7              C_2 = C_2 ∪ {v} C_S \ {v}
8      for v ∈ C_S do
9          if |N(v) ∩ C_2| == |C_2| & |N(v) ∩ C_1| == 0 then
10             C_1 = C_1 ∪ {v}
11     V_S = V \ (S ∪ C_1 ∪ C_2)
12     for v ∈ V_S do
13         if |N(v) ∩ C_1| == |C_1| & |N(v) ∩ C_2| == 0 then
14             return ∅
15     for v ∈ V_S do
16         if |N(v) ∩ C_2| == |C_2| & |N(v) ∩ C_1| == 0 then
17             return ∅
18     return C^+ = C_1 × C_2
```

Proof. Note that $C \subseteq C^* = C_1 \times C_2$ at line 12. As `MakeIndMaximal` returns $\emptyset$ there must be a vertex $v \in V_S = V \setminus (S \cup C^*)$ which can be added to C^*. Let D be a MIB containing C^* and v, thus D suffices to prove the lemma.

□

Lemma 11. `MakeIndMaximal` *runs in* $O(m)$ *time.*

Proof. Note that because G is connected, $n \in O(m)$. Setting C_S and V_S can be done in $O(n)$ time. In each for loop, we can scan all of the edges incident to each v in the iterated-over set and keep count of how many nodes from C_i have been seen (checking for inclusion can be done in $O(1)$ time with an $O(n)$ initialization step). Thus, each edge is scanned at most once per for loop.

□

4.4.5 `AddTo`

Recall that `AddTo` takes in (C, v) where $C = C_1 \times C_2$ is an induced biclique and $v \in V \setminus (C_1 \cup C_2)$, and returns the induced biclique where v is added to C_1, $N(v)$ is removed from C_1, and $\overline{N}(v)$ is removed from C_2 if $C_2 \setminus \overline{N}(v) \neq \emptyset$ and $\emptyset$ otherwise. We give pseudo-code for `AddTo` in Algorithm 6.

Lemma 12. `AddTo` *returns the induced biclique where* v *is added to* C_1, $N(v)$ *is removed from* C_1, *and* $\overline{N}(v)$ *is removed from* C_2 *if* $C_2 \setminus \overline{N}(v) \neq \emptyset$, *and* $\emptyset$ *otherwise.*

Algorithm 6: `AddTo`

```
Algorithm AddTo(G = (V, E), C = C_1 × C_2, v ∈ V \ (C_1 ∪ C_2))
    C_1' = (C_1 ∪ {v}) \ N(v)
    C_2' = C_2 ∩ N(v)
    if C_2' == ∅ then
        return ∅
    return C_1' × C_2'
```

Proof. Referring to the pseudo-code in Algorithm 6, it is clear that v is added to C_1 and $N(v)$ is removed from C_1. Additionally v's non-neighbors are effectively removed from C_2 by intersecting it with $N(v)$. If $C_2' = \emptyset$ then $C_2 \setminus \overline{N}(v) = \emptyset$ and $\emptyset$ is returned. Otherwise $C_1' \neq \emptyset$ since it includes v and thus $C_1' \times C_2'$ is a biclique. $C_1' \times C_2'$ must be an induced biclique as $C_2' \subseteq C_2$, $C_1' \setminus \{v\} \subseteq C_1$, and $C_1 \times C_2$ is an induced biclique and $(N(v) \cap C_1') = \emptyset$ by definition.

□

Lemma 13. `AddTo` *runs in* $O(m)$ *time.*

Proof. Note that because G is connected, $n \in O(m)$. `AddTo` can be completed by scanning all of v's $O(m)$ incident edges in tandem with an $O(n)$ preprocessing step to allow for constant-time look-ups when checking for inclusion in a set.

□

4.5 Implementation Details

In this section we describe several relevant implementation details and design decisions in our algorithm implementations [Hor19].

We always (re-label and) store vertices as $\{0, 1, \ldots n\}$ and maintain internal dictionaries as needed to recover original labels – e.g. when taking subgraphs. This allows us to leverage native data types and structures; vertices are stored as `size_t`.

For efficiency in subroutines, we utilize two representations of G. One representation is as adjacency lists, stored as sorted vectors (to improve union and intersection relative to dictionaries or unsorted vectors). This representation is essential in the performance of checking for (future) maximality in `OCT-MIB` or `MakeIndMaximal` and `AddTo` in `OCT-MIB-II`. We also store the graph as a dictionary of dictionaries which is more amenable to taking subgraphs (as when finding MISs in both `OCT-MIB` and `OCT-MIB-II`). Deleting a node requires time $O(N(v))$ as compared to $O(N(v)\Delta(G))$, where $\Delta(G)$ is the maximum degree, in the adjacency list representation.

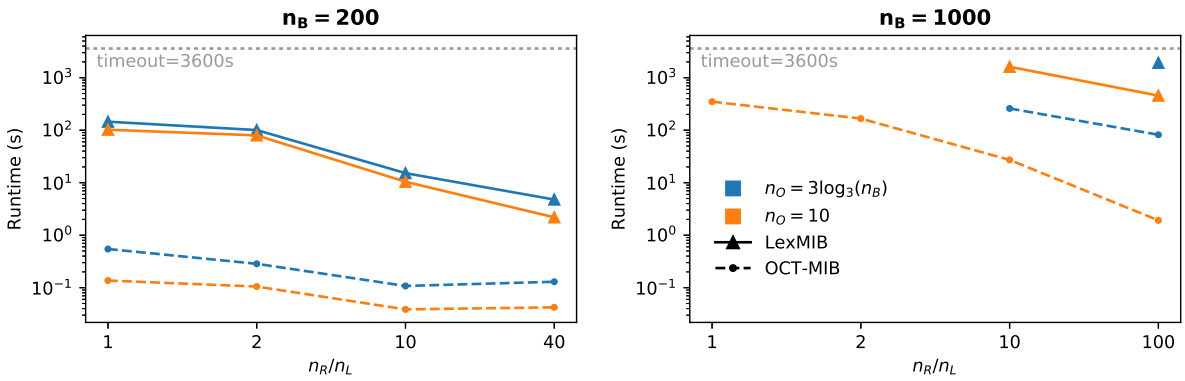

Figure 4.3 Runtimes of `OCT-MIB` and `LexMIB` under varied bipartite balance conditions. For $n_B = 200$ (*Left*, and $n_B = 1000$ (*Right*), each curve represents the runtime in seconds of an algorithm on graphs with a given OCT size and varied balance. When n_B was 1000, `OCT-MIB` timed out (3600s) on 90% of instances with balance 1:1 and 1:2. `LexMIB` timed out on 100% of instances at these settings, as well as 70% of those with balance 1:10.

4.6 Empirical Evaluation

In this section we evaluate the performance of `OCT-MIB`, `OCT-MIB-II`, and `Enum-MIB` on a suite of synthetic graphs with a variety of sizes, densities, degree distributions, and OCT decomposition structures, to see how various aspects of the graph impact the runtime. We also implemented `LexMIB`, our modified version of the approach described in [Dia05] (see Section 4.7), but do not include it in our evaluation as it was consistenly orders of magnitude slower than the other approaches (see Figure 4.3 as an example, further examples are included in [Klo19]). All code is open source under a BSD 3-clause license and publicly available as part of MI-Bicliques at [Hor19].

4.6.1 Data and experimental setup

Data. For convenience, throughout this section, we assume $n_L \geq n_R$ and let $n_B = n_L + n_R$. Our synthetic data was generated using a modified version of the random graph generator of Zhang et al. [Zha14b] that augments random bipartite graphs to have OCT sets of known size. The generator allows a user to specify the sizes of L, R, and O (n_L, n_R, and n_O), the expected edge densities between L and R, O and $L \cup R$, and within O, and the coefficient of variation (cv; the standard deviation divided by the mean) of the expected number of neighbors in L over R and in $L \cup R$ over O. The generator is seeded for replicability. We use the naïve OCT decomposition $[L, R, O]$ returned by the generator for our algorithm evaluation, but the techniques mentioned in Section 4.2 could also be used to find alternative OCT sets. Unless otherwise specified, the following default parameters are used: expected edge density $\bar{d} = 5\%$, $cv = 0.5$, $n_B = 1000$ and $n_L/n_R = 1/10$; additionally, the edge density between O and $L \cup R$ is the same as that between L and R.

To add the edges between L and R, the edge density and cv values are used to assign vertex degrees to R, and then neighbors are selected from L uniformly at random; this was implemented

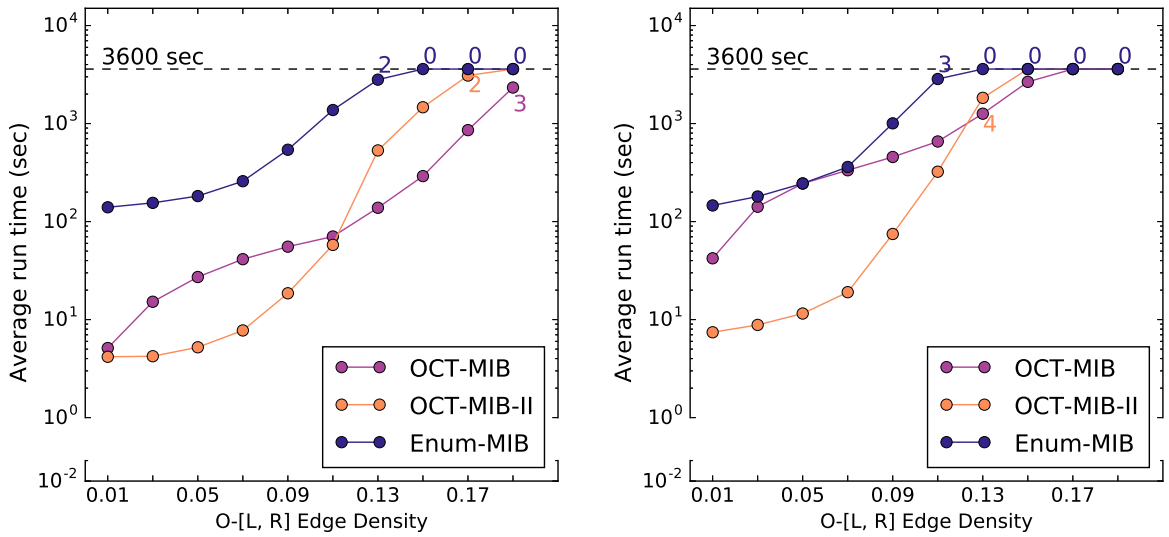

Figure 4.4 Runtimes of the MIB-enumerating algorithms on graphs where $n_B = 1000$, $n_L/n_R = 10$, and the expected edge density between O and $L \cup R$ was varied with all other densities were 0.05. $n_O = 10$ in the left panel and $n_O = 3\log_3 n_B$ in the right panel.

in the generator of [Zha14b]. Edges are added between O and $L \cup R$ via the same process, only with the corresponding edge density and *cv* values. Finally, we add edges within O with an Erdős-Rényi process based on expected density (no *cv* value is used here).

In most experiments we limit n_O to be $O(3\log_3 n_B)$, and use a timeout of one hour (3600s). Unless otherwise noted we run each parameter setting with five seeds and plot the average over these instances, using the time-out value as the runtime for instances that don't finish. If all instances of a parameter setting did not finish, we annotate the corresponding point with the number of points which *did* finish.

We began by running our algorithms on an initial corpus of graphs with roughly 1000 vertices (see 4.6.2). We then scaled up both n_B and n_O to create new sets of experiments, discussed in 4.6.3. We also ran our algorithms on computational biology graphs from [Wer14], which have been shown to be near-bipartite; these results are in 4.6.4.

Hardware. All experiments were run on identical hardware; each server had four Intel Xeon E5-2623 v3 CPUs (3.00GHz) and 64GB DDR4 memory. The servers ran Fedora 27 with Linux kernel 4.16.7-200.fc27.x86_64. The C/C++ codes were compiled using gcc/g++ 7.3.1 with optimization flag -O3.

4.6.2 Initial Benchmarking

We begin by evaluating our algorithms on a dataset designed to independently test the effect of each parameter (the expected densities in various regions of the graph, the *cv* values, n_O, n_B, and n_L/n_R) on the algorithms' runtime. We observe that `OCT-MIB-II` is generally the best algorithm,

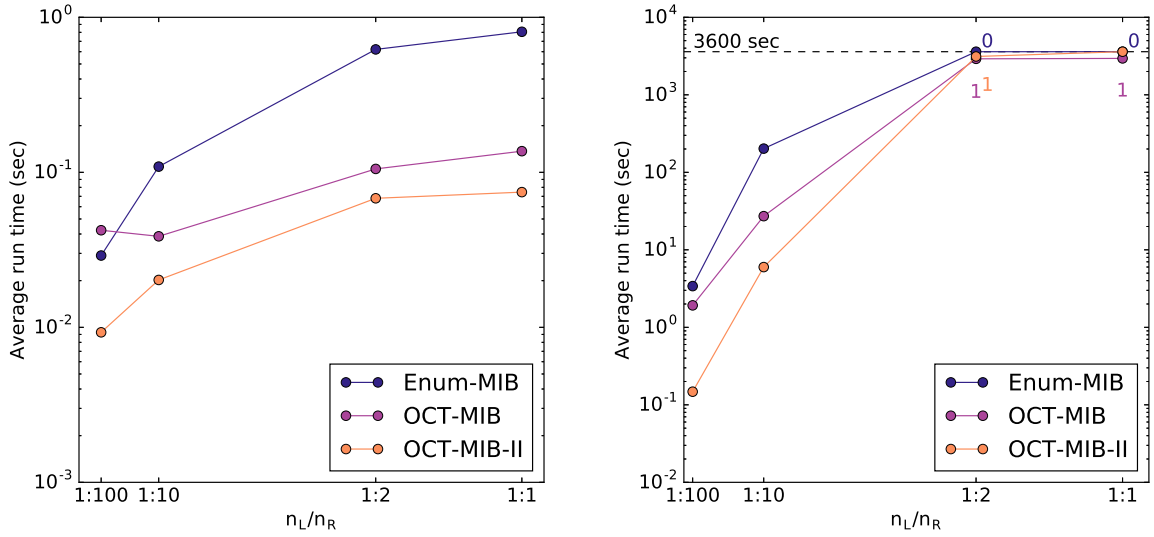

Figure 4.5 Runtimes of the MIB-enumerating algorithms on graphs where $n_O = 10$ and the ratio n_L/n_R was varied. The left panel has $n_B = 200$ and the right panel has $n_B = 1000$.

and include comprehensive plots of all experiments below.

We observe that in general, `OCT-MIB-II` outperforms `OCT-MIB` and `Enum-MIB`. This is the case when the varying parameter is the density within O (Figure 4.10), the *cv* between L and R (Figures 4.7 and 4.8), the size of the OCT set n_O (Figure 4.9), and the ratio between L and R (Figures 4.5 and 4.6), amongst other settings. In these "near-bipartite" synthetic graphs, `Enum-MIB` unsurprisingly is slowest on most instances. When $n_B = 1000$ and $n_O = 3\log_3(n_B)$, `Enum-MIB` outperforms `OCT-MIB` when the density within O increases above 0.05 (Figure 4.10). This is likely due to the adverse effect of the number of MISs in the OCT set on `OCT-MIB`. The most interesting observation occurs when varying the edge density between O and $L \cup R$ (Figure 4.4). In the $n_O = 10$ case (left panel), `OCT-MIB-II` is the fastest algorithm until the density exceeds 0.11, when `OCT-MIB` becomes faster. We believe this is likely due to `OCT-MIB` efficiently pruning away attempted expansions which are guaranteed to fail, while the number of MISs in O does not increase. This behavior is also seen in the case where $n_O = 3\log_3 n_B$ (right panel), though the magnitude of the difference is not as extreme.

4.6.3 Larger Graphs

Given the fast runtimes achieved in Section 4.6.2 we created a new corpus of larger synthetic graphs where we scaled up n_B to 10,000 and varied n_O in two settings, and increased the timeout to 7200 seconds. When the expected density was 0.03 and $n_L/n_R = 100$, `OCT-MIB-II` outperformed `OCT-MIB` for all values of n_O by at least an order of magnitude and finished on all instances, whereas `OCT-MIB` timed out on all instances with $n_O \geq 13$ (left panel of Figure 4.14). However, when the expected density was 0.01 and $n_L/n_R = 9$, `OCT-MIB` was faster (right panel of Figure 4.14). We speculated that

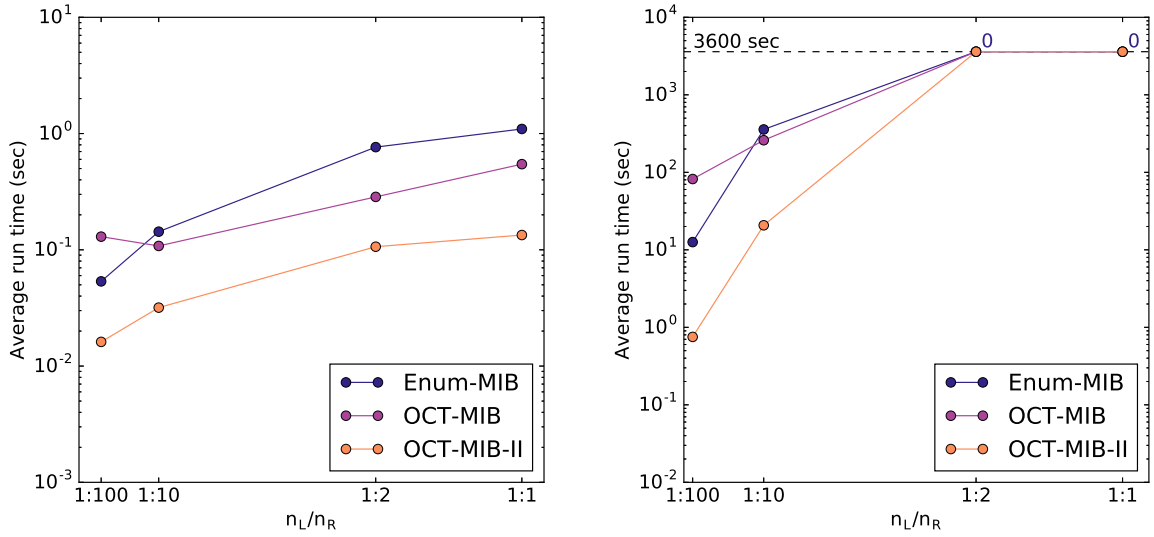

Figure 4.6 Runtimes of the MIB-enumerating algorithms on graphs where $n_O = 19 \approx 3\log_3(n_B)$ and the ratio n_L/n_R was varied. The left panel has $n_B = 200$ and the right panel has $n_B = 1000$.

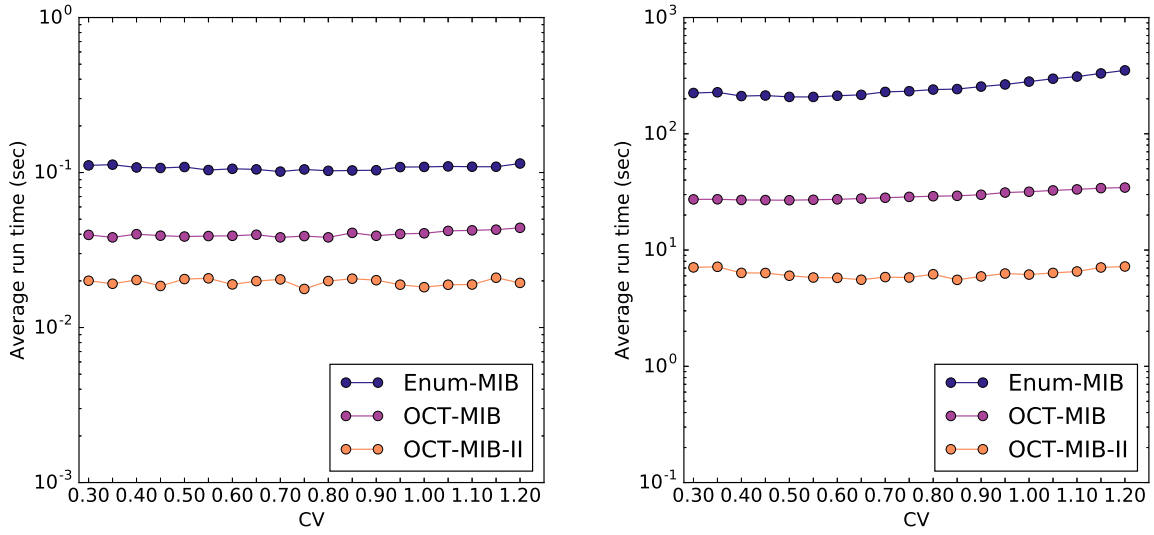

Figure 4.7 Runtimes of the MIB-enumerating algorithms on graphs where $n_O = 10$ and the coefficient of variation between L and R was varied. The left panel has $n_B = 200$ and the right panel has $n_B = 1000$.

this was due to the sparsity of O, allowing for a speed-up due to the efficient pruning of `OCT-MIB` similar to what was seen in Section 4.6.2. To test this theory, we increased expected edge density within O to 0.05 while leaving the other parameters the same (right panel of Figure 4.14), and observed that once $n_O \geq 9$, `OCT-MIB-II` outperforms `OCT-MIB`, confirming our hypothesis.

We additionally created graphs with $n_O > 3\log_3 n_B$, which was not done in our initial set of

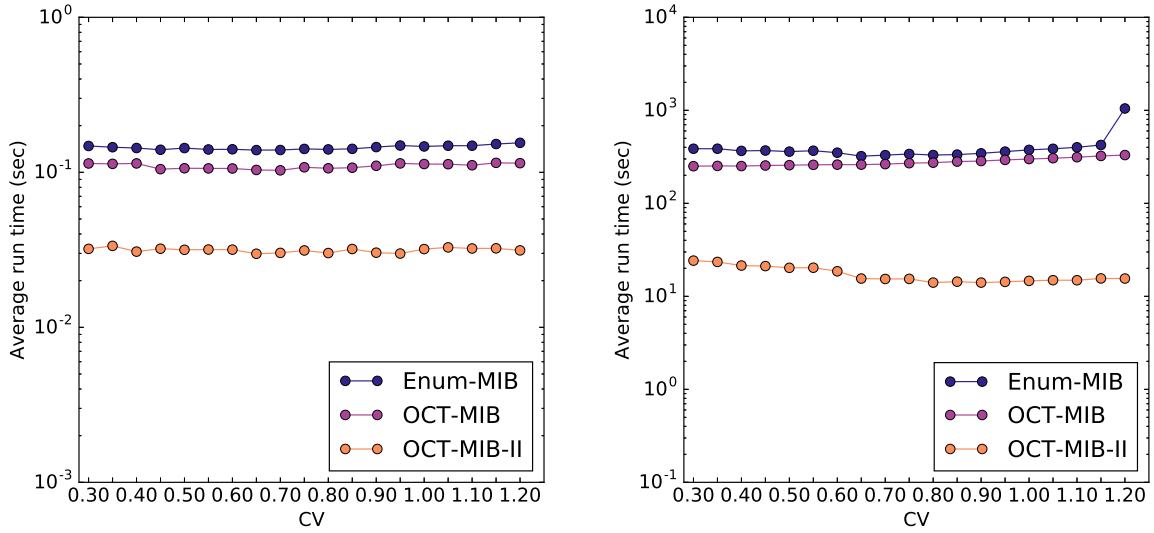

Figure 4.8 Runtimes of the MIB-enumerating algorithms on graphs where $n_O = 19 \approx 3\log_3(n_B)$and the coefficient of variation between L and R was varied. The left panel has $n_B = 200$ and the right panel has $n_B = 1000$.

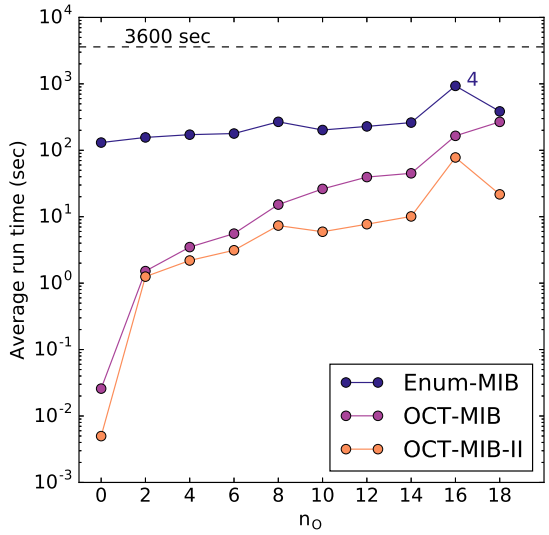

Figure 4.9 Runtimes of the MIB-enumerating algorithms on graphs where $n_B = 1000$ and n_O was varied.

experiments (Figure 4.15). These graphs had n_B values up to 4000 and for each value of n_B, we used three values of n_O; $10, 3\log_3 n_B$, and $\sqrt{n_B}$. The results were most interesting for the MIB-enumerating algorithms (Figure 4.15 top). `OCT-MIB` performed the worst of the three algorithms when $n_O = \sqrt{n_B}$, but outperformed `Enum-MIB` in the other settings. This verifies the analysis from [Klo19] on the range in which `OCT-MIB` is most effective. In general, `OCT-MIB-II` once again was the fastest algorithm

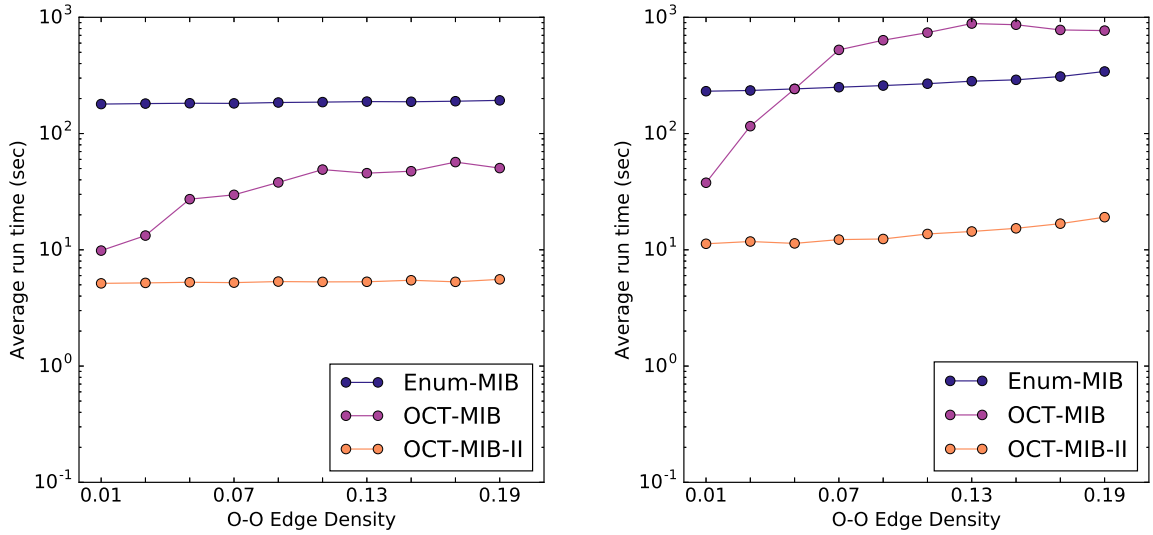

Figure 4.10 Runtimes of the MIB-enumerating algorithms on graphs where $n_B = 1000$ and the expected edge density within O was varied. The left panel has $n_O = 10$ and the right panel has $n_O == 19 \approx 3\log_3(n_B)$.

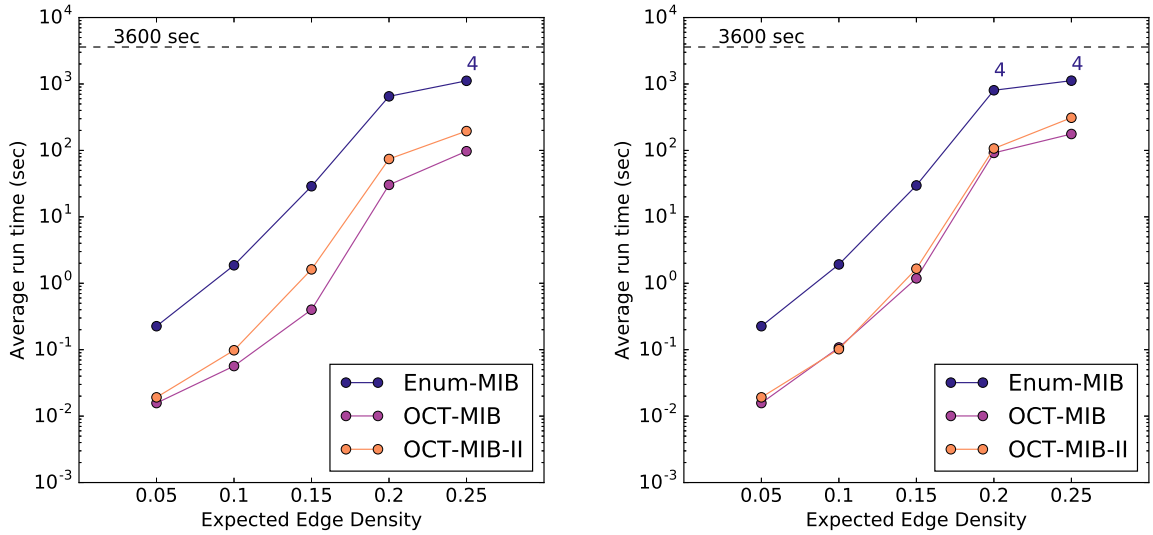

Figure 4.11 Runtimes of the MIB-enumerating algorithms on graphs where $n_B = 150$, $n_L = n_R$ and $n_O = 5$. The expected edge density between L and R and between $\{L,R\}$ and O was varied. In the left panel the expected edge density within O was fixed to 0.05 and in the right panel it varied with the rest of the graph.

and did best when n_O was smaller. The impact of n_O on `OCT-MIB-II` and `Enum-MIB` appeared comparable.

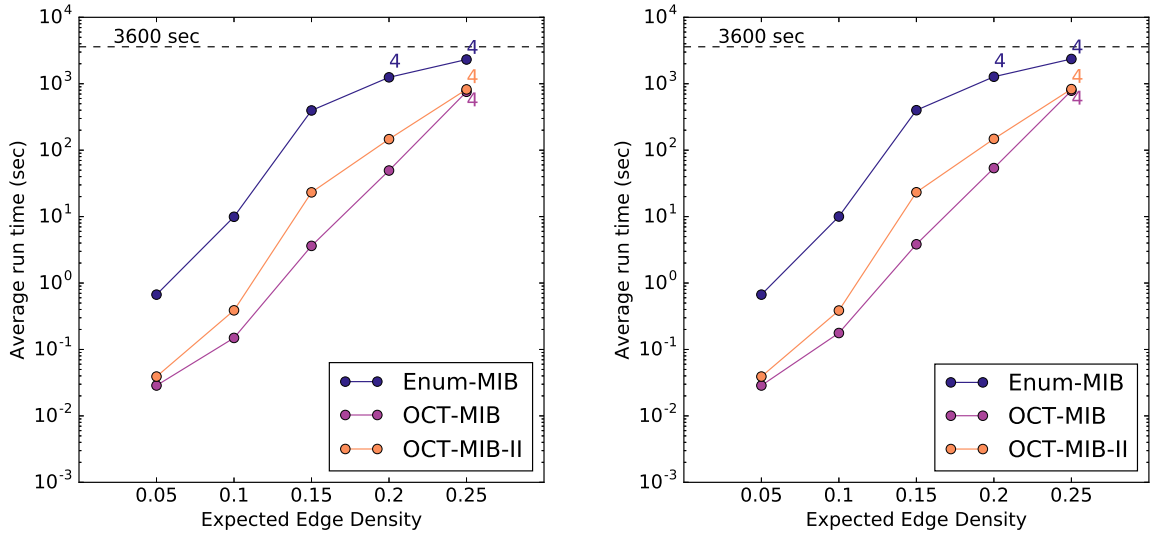

Figure 4.12 Runtimes of the MIB-enumerating algorithms on graphs where $n_B = 200$, $n_L = n_R$ and $n_O = 5$. The expected edge density between L and R and between $\{L,R\}$ and O was varied. In the left panel the expected edge density within O was fixed to 0.05 and in the right panel it varied with the rest of the graph.

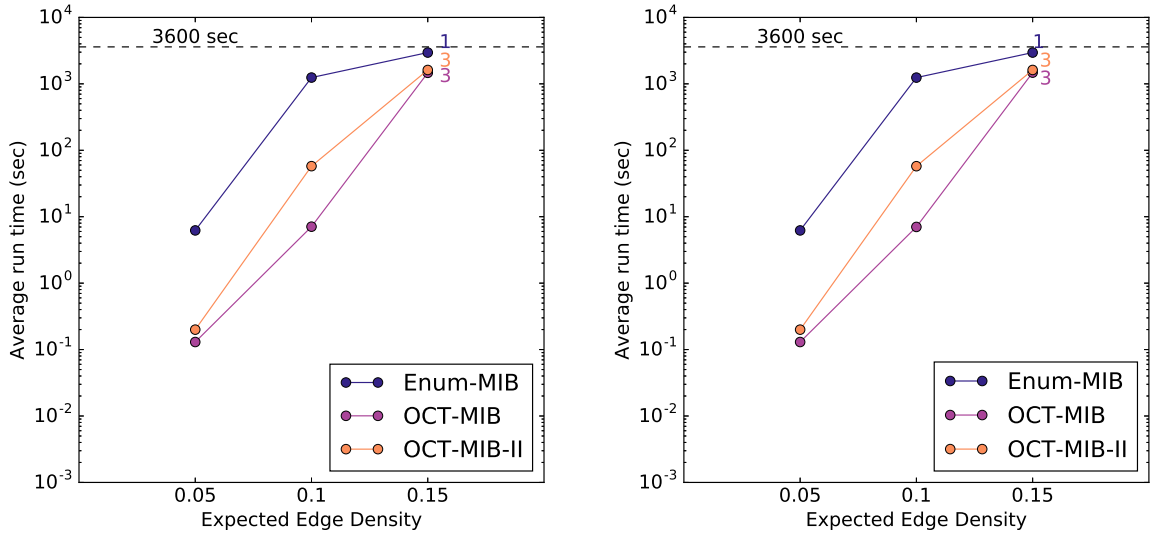

Figure 4.13 Runtimes of the MIB-enumerating algorithms on graphs where $n_B = 300$, $n_L = n_R$ and $n_O = 5$. The expected edge density between L and R and between $\{L,R\}$ and O was varied. In the left panel the expected edge density within O was fixed to 0.05 and in the right panel it varied with the rest of the graph.

4.6.4 Computational Biology Data

Finally, we tested performance on real-world data using the graphs from [Wer14], which come from computational biology. These graphs have previously been exhibited to have small OCT sets [Hüf05],

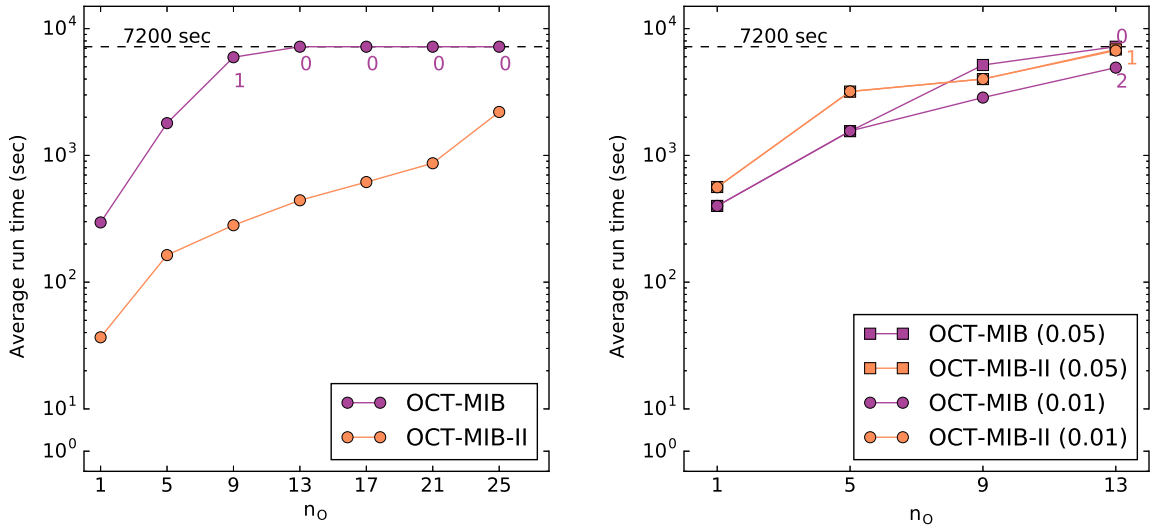

Figure 4.14 Runtimes of the OCT-based MIB-enumeration algorithms on graphs where $n_B = 10000$ and n_O varies. In the left panel, $n_L = 9901, n_R = 99$ ($n_L/n_R \approx 100$) and the expected edge density is 0.03. In the right panel, $n_L = 9091, n_R = 909$ ($n_L/n_R \approx 10$) and the expected edge density (excluding within O) is 0.01; the marker-type denotes the expected edge density within O (see legend). For these larger instances we used 3 seeds and a 7200s timeout.

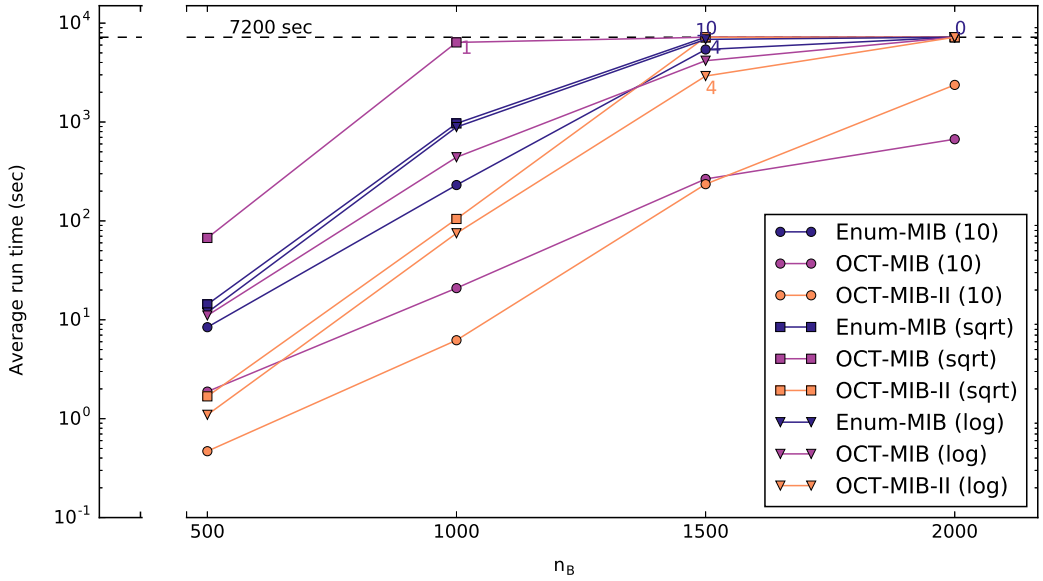

Figure 4.15 Runtimes of the MIB-enumerating algorithms on graphs where $n_L/n_R = 9$ and all expected edge densities are 0.05. n_B is varied (x-axis) and the marker-type denotes the value of $n_O \in \{10, \sqrt{n_B}, 3\log_3(n_B)\}$ (see legend). The time-out value is set to 7200s.

and we used the implementation from [Goo18] of Hüffner's iterative compression algorithm [Hüf05] to find the OCT decompositions. Computing the OCT decomposition for each graph ran in less than

ten seconds, and often in less than one second. As can be seen in Tables 4.1 and 4.2, `OCT-MIB-II` performs the best of the MIB-enumerating algorithms.

4.7 Discussion of Dias et al.

Here we identify graphs containing induced maximal bicliques which would not be discovered by the algorithm of Dias et al. as originally stated in [Dia05], which we refer to as `Dias`. We then describe a modified approach, which we prove is guaranteed to output all induced maximal bicliques in lexicographic order[2] in time $O(Mn^4)$, where M is the number of MIBs in the graph.

For the reader's convenience we have transcribed the pseudo-code of `Dias` from [Dia05] in Algorithm 7. Line 13 relies on a subroutine described in the original paper, which finds the lexicographically least biclique containing an given set of nodes in $O(n^2)$ time. For consistency with [Dia05], we let N_j be the neighbors of node j and $\bar{N}_j$ the non-neighbors throughout this subsection.

Algorithm 7: `Dias` pseudo-code. Note that there is an order on V. `Dias` was intended to output the maximal induced bicliques of G in lexicographic order.

```
1  Algorithm Dias(G = (V, E))
2      Find the least biclique B* of G
3      Q ← ∅
4      insert B* in the queue Q
5      while Q ≠ ∅ do
6          find the least biclique B = X ∪ Y of Q
7          remove B from Q and output it
8          for each vertex j ∈ V \ B do
9              X_j ← X ∩ {1,...,j}; Y_j ← Y ∩ {1,...,j}
10             if X_j ∩ N_j ≠ ∅ or Y_j ∩ N̄_j ≠ ∅ then
11                 X'_j ← (X_j \ N_j) ∪ {j}; Y'_j ← Y_j \ N̄_j
12                 if there exists no l ∈ {1,...,j−1} \ B_j such that X'_j ∪ Y'_j ∪ {l} extends to a
                     biclique of G then
13                     find the least biclique B' of G containing X'_j ∪ Y'_j, if any
14                     if B' ≠ ∅ and B' ∈ Q then
15                         Include B' in Q
16             Swap contents of X_j and Y_j, repeat lines 10 to 15 (once per for loop)
```

We point out that Q needs to be able to recall all MIBs which have been stored in it, which can easily be accomplished by augmenting the data structure without impacting the complexity. Furthermore, the pseudo-code in Algorithm 7 was corrected in line 12 to exclude j from the range

[2] We also modified `Dias` to output all MIBs in non-lexicographic order, with runtime $O(nmM)$ (see Section 4.4.2).

Table 4.1 The runtimes (rounded to nearest thousandth-of-a-second) of the biclique-enumeration algorithms on the Afro-American subset of the Wernicke-Hüffner computational biology data [Wer14].

G	n_B	m	n_O	$\lvert M_I \rvert$	OCT-MIB-II	OCT-MIB	Enum-MIB
aa-10	69	191	6	178	0.008	0.023	0.057
aa-11	102	307	11	424	0.055	0.115	0.259
aa-13	129	383	12	523	0.083	0.239	0.47
aa-14	125	525	19	1460	0.366	0.902	1.254
aa-15	66	179	7	206	0.01	0.019	0.053
aa-16	13	15	0	15	0	0	0
aa-17	151	633	25	2252	1.023	2.132	3.457
aa-18	87	381	14	660	0.1	0.173	0.389
aa-19	191	645	19	1262	0.449	1.569	2.385
aa-20	224	766	19	1607	0.705	2.431	3.809
aa-21	28	90	9	116	0.006	0.013	0.008
aa-22	167	641	16	1520	0.423	1.387	2.629
aa-23	139	508	18	1766	0.435	0.788	1.651
aa-24	258	1108	21	3890	2.108	9.14	14.167
aa-25	14	15	1	10	0	0.001	0.001
aa-26	92	284	13	583	0.084	0.186	0.309
aa-27	118	331	11	458	0.054	0.27	0.343
aa-28	167	854	27	2606	1.464	2.201	4.162
aa-29	276	1058	21	3122	1.909	8.418	10.707
aa-30	39	71	4	56	0.002	0.007	0.006
aa-31	30	51	2	37	0.002	0.002	0.002
aa-32	143	750	30	4167	2.286	7.694	5.29
aa-33	193	493	4	578	0.046	0.204	0.993
aa-34	133	451	13	705	0.132	0.316	0.756
aa-35	82	269	10	459	0.037	0.108	0.178
aa-36	111	316	7	248	0.015	0.076	0.155
aa-37	72	170	5	135	0.005	0.018	0.054
aa-38	171	862	26	4270	2.428	5.223	7.586
aa-39	144	692	23	2153	0.872	1.574	3.034
aa-40	136	620	22	2727	1.022	2.086	2.973
aa-41	296	1620	40	11705	16.519	82.439	50.205
aa-42	236	1110	30	6967	5.646	45.56	21.244
aa-43	63	308	18	905	0.137	0.294	0.311
aa-44	59	163	10	211	0.014	0.024	0.051
aa-45	80	386	20	1768	0.336	0.775	0.859
aa-46	161	529	13	719	0.157	0.438	0.922
aa-47	62	229	14	572	0.057	0.082	0.138
aa-48	89	343	17	896	0.144	0.338	0.497
aa-49	26	62	5	50	0.004	0.002	0.003
aa-50	113	468	18	1272	0.322	0.778	1.098
aa-51	78	274	11	429	0.035	0.082	0.174
aa-52	65	231	14	690	0.073	0.135	0.2
aa-53	88	232	12	340	0.036	0.186	0.162
aa-54	89	233	12	286	0.027	0.063	0.113

Table 4.2 The runtimes (rounded to nearest thousandth-of-a-second) of the biclique-enumeration algorithms on the Japanese subset of the Wernicke-Hüffner computational biology data [Wer14].

G	n_B	m	n_O	$\lvert M_I\rvert$	OCT-MIB-II	OCT-MIB	Enum-MIB
j-10	55	117	3	52	0.002	0.009	0.01
j-11	51	212	5	63	0.003	0.014	0.011
j-13	78	210	6	224	0.015	0.028	0.074
j-14	60	107	4	44	0.004	0.007	0.003
j-15	44	55	1	13	0.001	0	0.004
j-16	9	10	0	10	0	0	0
j-17	79	322	10	317	0.025	0.051	0.127
j-18	71	296	9	154	0.011	0.038	0.053
j-19	84	172	3	105	0.002	0.01	0.019
j-20	241	640	1	274	0.013	0.065	0.484
j-21	33	102	9	107	0.006	0.012	0.008
j-22	75	391	9	221	0.02	0.051	0.08
j-23	76	369	19	682	0.095	0.404	0.217
j-24	142	387	4	150	0.013	0.027	0.089
j-25	14	14	0	14	0	0	0
j-26	63	156	6	156	0.007	0.019	0.035
j-28	90	567	13	492	0.073	0.13	0.244

of l values.

In the proof of correctness in [Dia05] they show that for any MIB B' there exists a node $j \in B'$ and another MIB B which does not contain j, but does contain both $B'_{j-1} = B' \cap \{1,\ldots,j-1\}$ and some $l \in \{1,\ldots,j-1\} \setminus B'_{j-1}$. They examine the maximal such j for each B', and consider the iteration where B and j are defined as such (lines 7 and 8). Without loss of generality, assume that j should be added to X_j (as defined in line 9). If Y_j only contains non-neighbors of j and X_j contains non-neighbors of j which are not in B', then no biclique will be found in line 15. Thus the algorithm as written will not produce all MIBs, as shown in Figure 4.16.

4.7.1 Modified `Dias`

We now describe `LexMIB` and show that it finds all MIBs in lexicographic order. The first issue in `Dias` arises when in line 9, Y_j is empty and X_j does not contain any neighbors of j. In this case, we fail to satisfy the **if**-condition in line 10. The problem is resolved by appending an additional **or**-condition in line 10: "**or** $|Y_j| = 0$".

The other problematic case is when Y'_j is empty and X_j contains non-neighbors of j which are not in the lexicographically next MIB (e.g. node 0 for $\{1,3\} \times \{5\}$ in Figure 4.16). In this case, $B' = \emptyset$ in line 14, so we add a new **else-if** clause to the conditional: "**else if** $B' = \emptyset$ and $Y'_j = \emptyset$ **then for all** $v \in N_j$ find the least biclique containing $(N(v) \cap X'_j, \{v\})$".

We now argue that `LexMIB` will find any biclique B' missed by `Dias` in the above manner. Let $S = B' \cap X'_j$. Let v_1 be the least node in B' which shares an edge with j. Consider the iteration of the above process where $v = v_1$. Clearly all of S is contained in $N(v_1) \cap X'_j$. Furthermore due to

the maximality of j, the original argument from [Dia05] can be used to show that finding the least biclique containing $(N(v_1) \cap X'_j, \{v\})$ returns B'. In Figure 4.16 where biclique C is missed, when $j = 3$ and $B = \{0,1,4\} \times \{2\}$, $v_1 = 5$ and $N(v_1) \cap X'_j = \{1,3\}$, which returns this previously missing biclique $C = \{1,3\} \times \{5\}$.

This augmentation increases the delay time of the algorithm to be $O(n^4)$ since finding the least biclique may be called $O(n)$ times within the **for** loop at line 8, which itself has $O(n)$ iterations. Note that this addition does not alter their argument for the bicliques being output in lexicographic order.

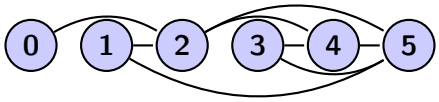

Figure 4.16 An example of a graph where `Dias` would not find all MIBs. The MIB $C = \{1,3\} \times \{5\}$ would not be found. In the notation of the algorithm $j = 3$ and $B = \{0,1,4\} \times \{2\}$ should yield C. When $X_j = \{0,1\}$, no biclique is found as $\{0,1,3\}$ have no common neighbors. When $X_j = \{2\}$ the biclique $\{2,3\} \times \{4\}$ is found. Even if we removed the if-condition on line 12, when $j = 5$ we would add the bicliques $\{1,4\} \times \{5\}$ and $\{0,5\} \times \{2\}$ to Q. When $B = \{0,5\} \times \{2\}$ (line 7), no new bicliques are added to Q and then the next least biclique output in line 6 would not be C.

4.8 Extremal Case for Zhang et al.

The $O(Bm)$ runtime[3] stated for the algorithm `iMBEA` in [Zha14b] contains a typo. The corrected runtime is $O(Bnm)$. In their analysis they bound the number of nodes in their seach tree and derive their complexity by bounding the time spent on each one by $O(m)$. They show that the number of intermediate search tree nodes is at most $\sum_{i=0}^{d-1}(n-1)^i$, which is $O(B)$, and that the number of leaves is at most $(n-1)^d$ which is said to be $O(B)$. However, by the geometric series, the number of leaves can only be bounded by $O(Bn)$, giving an $O(Bn)$ bound on the number of search tree nodes. We now provide an example of a graph where the number of leaves is $\Omega(Bn)$, implying that the number of leaves cannot be $O(B)$ in the general case.

Graphs which are a perfect matching plus an apex to one node from each edge in the matching provide an example of a graph family where this extremal behavior manifests. Note that these graphs are bipartite with n_a nodes in the smaller partition and $n_b = n_a + 1$ in the larger. See Figure 4.17 for the instance where $n_a = 4$ and $n_b = 5$. There are $B = (n_a + 1)$ MIBs in such a graph. Assume the nodes in the smaller partition are labeled $a_1, \ldots, a_{n_a}$. For $1 < i < n_a$, `iMBEA` attempts to expand the biclique containing a_i and its two neighbors with all nodes in $\{a_{i+1}, \ldots, a_{n_a}\}$, each of which creates a leaf in the search tree. Thus there are $\sum_{i=2}^{n_a-1} n_a - i = \Omega({n_a}^2)$ leaves created and the number of leaves is $\Omega(B n_a) = \Omega(Bn)$.

[3] where B is the number of MIBs found

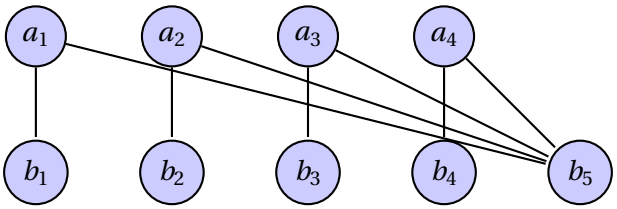

Figure 4.17 An example of a graph in which `iMBEA` would run in time $\Omega(Bmn)$.

4.9 Conclusions

We present a suite of new algorithms for enumerating maximal induced bicliques in general graphs, two of which are parameterized by the size of an odd cycle transversal. It is particularly noteworthy that the parameterized algorithms empirically outperform the general approaches even when their asymptotic worst-case complexities are worse – and when the parameters are far from the constant regime often necessary for FPT approaches to be efficient. This also highlights a weakness of standard complexity analysis, as many aspects of an algorithm get "swept under the rug". Additionally, we describe a flaw in the algorithm of Dias et al. [Dia05], and give a corrected variant `LexMIB`.

In [Sul19] we also gave a novel maximal biclique (MB) enumeration technique optimized for near-bipartite graphs. Recall these structures are not necessarily induced. This new technique outperformed general methods on all graphs mentioned in section 4.6. Our method combined aspects of our `OCT-MIB-II` technique mentioned above with the `MICA` algorithm of Alexe et al. [Ale04]. It was interesting to find that even though MIB and MB-enumeration are closely related problems, the MB-enumerating algorithms are often an order of magnitude faster than their MIB-enumerating counterparts. The reason for this can likely be attributed to two causes: the number of MBs is significantly less than the number of MIBs in sparse graphs, and that the stricter structure of MIBs requires more work to ensure. For $S \subseteq V$, there is exactly one MB of the form $S \times T \subseteq V$ in G, but there can be many MIBs with this structure.

We implement and benchmark all of the algorithms on a corpus of synthetic and real-world computational biology graphs, and establish that parameterized approaches are often at least an order of magnitude faster than the general approaches. This remains true even when $n_O \in O(\sqrt{n})$. It would be interesting to experimentally evaluate as n_O increases, at what point the standard methods outperform those optimized for near-bipartite graphs. Finally, we note as in [Sul19], the current implementations of the algorithms could be improved by replacing the MIS-enumeration algorithm with that of [Tsu77], and the MIB-enumeration on bipartite graphs with the implementation used in [Zha14b].

CHAPTER

5

APPROXIMATING VERTEX COVER USING STRUCTURAL ROUNDING

Executive Summary

In this chapter, we provide the first practical evaluation of the structural rounding framework for approximation algorithms. Structural rounding works by first editing to a well-structured class, efficiently solving the edited instance, and “lifting" the partial solution to recover an approximation on the input. We focus on the well-studied VERTEX COVER problem, and edit to the class of bipartite graphs (where VERTEX COVER has an exact polynomial time algorithm). In addition to the naïve lifting strategy for VERTEX COVER described by Demaine et al. in the original structural rounding paper, we introduce a suite of new lifting strategies and measure their effectiveness on a large corpus of synthetic graphs. We find that in this setting, structural rounding significantly outperforms standard 2-approximations for VERTEX COVER. Further, simpler lifting strategies are extremely competitive with the more sophisticated approaches. The implementations are available as an open-source Python package, and all experiments are replicable.

The implementation of these algorithms was done collaboration with Brian Lavallee. This chapter was advised by Blair D. Sullivan and its contents were accepted to the 2020 SIAM Symposium on Algorithm Engineering and Experiments (ALENEX20).

5.1 Introduction

Approximation algorithms are a crucial tool for navigating the tradeoff between solution quality and runtime. Many approximation algorithms are only able to offer suitable solution quality guarantees

by exploiting the structure of limited classes of graphs, undercutting the scope of their applicability. *Structural rounding* is a recent framework [Dem19] for extending approximation algorithms for restricted classes to graphs "near" those classes. The approach consists of three phases: editing the input graph so that it has a given structural graph property (e.g. bounded treewidth), efficiently solving the edited-to instance, then "lifting" the partial solution to a valid outcome on the original graph. In this work, we provide the first implementation and experimental evaluation of structural rounding, using it to solve VERTEX COVER on near-bipartite graphs.

We chose this problem-graph class pair because of the famous result of König, that VERTEX COVER is polynomial time solvable on bipartite graphs [Kön31], using e.g. the Hopcroft-Karp algorithm [Hop73]. The theoretical guarantee of structural rounding in this setting is considerably worse than that of the standard 2-approximations (see Section 5.2.1). However, we show experimentally that structural rounding is in fact competitive with (and often much better than) the traditional approaches – despite these weaker theoretical guarantees.

While many algorithms already exist for the first two phases of structural rounding, the notion of "lifting" solutions has been relatively unexplored. The original structural rounding paper [Dem19] mainly employs naïve lifting algorithms. For example, in VERTEX COVER the entire edit set is added to the partial solution. We introduce a suite of new lifting algorithms for VERTEX COVER and experimentally evaluate their quality, comparing them to one another (as well as naïve and greedy approaches). We provide guarantees on the quality of these new lifting techniques relative to the optimal lift for a given partial solution.

Our experiments show that structural rounding outperforms traditional approximation techniques on a wide variety of graphs, and that naïve and greedy lifting algorithms are competitive with more sophisticated methods.

We begin by discussing preliminaries and previous work, including the structural rounding framework and existing approaches for approximating VERTEX COVER in Section 5.2. We then describe the new lifting methods in Section 5.3. In Section 5.4, we outline our experimental design, and then highlight interesting details of our implementation in Section 5.5. We discuss key takeaways and our results in Section 5.6.

5.2 Preliminaries

Much of this chapter pertains to VERTEX COVER, which asks for a set $S \subseteq V$ of minimum size such that every edge in G has an endpoint in S.

5.2.1 Structural Rounding

The structural rounding framework is conceptually simple: edit the input graph to a target class, apply an existing algorithm for that class (approximation or exact), and lift the partial solution to be a valid solution on the original graph. Under relatively broad assumptions, structural rounding extends algorithmic results for a structural class to graphs which are near that class. The framework supports

arbitrary graph edit operations and both minimization and maximization problems, provided they jointly satisfy two properties: a combinatorial property called "stability" and an algorithmic property called "structural lifting" [Dem19]. Roughly, these properties bound the amount each edit operation can change the optimal solution, but are parameterized to enable the derivation of tighter bounds when the problem has additional structure.

Formally, structural rounding makes use of the following definitions; we restrict our statements to the minimization setting (since VERTEX COVER is a minimization problem).

Definition 5. *A graph G' is γ-editable from a graph G under edit operation ψ if there is a sequence of $k \leq \gamma$ edits $\psi_1, \psi_2, \ldots, \psi_k$ of type ψ such that $G' = \psi_k(\psi_{k-1}(\cdots \psi_2(\psi_1(G))\cdots))$. A graph G is γ-close to a graph class $\mathscr{C}$ under ψ if some $G' \in \mathscr{C}$ is γ-editable from G under ψ.*

We follow [Dem19] and use Cost_Π to represent the cost function for solutions to a problem Π and $\text{OPT}_\Pi(G)$ for the value of optimal solution to the problem Π on graph G.

Definition 6. *A graph minimization problem Π is stable under an edit operation ψ with constant c' if $\text{OPT}_\Pi(G') \leq \text{OPT}_\Pi(G) + c'\gamma$ for any graph G' that is γ-editable from G under ψ.*

Definition 7. *A minimization problem Π can be structurally lifted with respect to an edit operation ψ with constant c if, given any graph G' that is γ-editable from G under ψ, along with a corresponding edit sequence $\psi_1, \psi_2, \ldots, \psi_k$ with $k \leq \gamma$, a solution S' for G' can be converted in polynomial time to a solution S for G such that $\text{Cost}_\Pi(S) \leq \text{Cost}_\Pi(S') + c \cdot k$.*

We can now state the main result of structural rounding (Theorem 4.1 in [Dem19]):

Theorem 7. *Let Π be a minimization problem that is stable under the edit operation ψ with constant c' and that can be structurally lifted with respect to ψ with constant c. If Π has a polynomial-time $\rho(\lambda)$-approximation algorithm in the graph class $\mathscr{C}_\lambda$, and $(\mathscr{C}_\lambda, \psi)$-EDIT has a polynomial-time (α, β)-approximation algorithm, then there is a polynomial-time $((1 + c'\alpha\delta) \cdot \rho(\beta\lambda) + c\alpha\delta)$-approximation algorithm for Π on any graph that is $(\delta \cdot \text{OPT}_\Pi(G))$-close to the class $\mathscr{C}_\lambda$.*

We note that an (α, β)-approximation is one which uses at most α times the optimal number of edits needed to reach a graph in class $\mathscr{C}_\lambda$ and returns a graph which is in class $\mathscr{C}_{\beta\lambda}$ (i.e. if $\mathscr{C}_\lambda$ is the class of graphs with treewidth at most λ, the returned graph has treewidth at most $\beta\lambda$). Demaine et al. demonstrated precisely how the ingredients of the framework fit together to obtain new approximations [Dem19], and we refer the interested reader to this work for more details.

We consider VERTEX COVER under the edit operation of vertex deletion. As noted in [Dem19], the stability constant is $c' = 0$ (deleting vertices cannot increase the size of a minimum size cover), and the lifting constant is $c = 1$ for VERTEX COVER, (since the union of the deleted vertices and any solution S' on G' is a vertex cover of G). Thus, by Theorem 7, VERTEX COVER has a $(1 + \sqrt{\log(n)}\delta)$-approximation for any graph G with an edit set of size $(\delta \, \text{OPT}_{\text{VC}}(G))$.

5.2.2 VERTEX COVER 2-Approximations

VERTEX COVER admits two straight-forward 2-approximations. The best known, referred to here as `standard`, was discovered by both Gavril and Yannakakis [Pap82], and repeatedly selects an arbitrary edge (u, v) in G, places both endpoints in the cover, and sets $G = G[V \setminus \{u, v\}]$. It is clear that whenever an edge is removed from G it is incident to a node which has been added to the cover, thus every edge is covered. Since every edge that we selected shares no end points with the other edges selected, any optimal solution must contain at least one of the two nodes from each selected edge.

The second 2-approximation, which we refer to as `DFS`, follows from the work of Savage [Sav82] and constructs a cover C by taking all of the non-leaf vertices from a depth-first search tree of the graph. Clearly this is a vertex cover as every edge touches some internal vertex. Further, the DFS tree has a matching with at least $|C|/2$ edges. Because these edges are independent, at least one endpoint must be a part of any valid cover, and thus C is a 2-approximation.

We compare the effectiveness of structural rounding with both of these 2-approximations. There are also more complicated methods which yield slightly better approximation factors; namely a $2-\Theta(1/\sqrt{\log|V|})$-approximation [Kar05] and a $2/(1+\delta)$-approximation in δ-dense graphs [Kar96]. We do not compare against these algorithms due to their higher complexity.

5.2.3 Odd Cycle Transversals

Recall from Chapter 2 that an *odd cycle transversal* is a set of vertices whose removal leaves a bipartite graph. Thus, graphs with "small" odd cycle transversals, or *OCT sets*, are referred to as "near-bipartite," and this notion naturally arises in many applications [Gül04; Pan04; Sch17]. We use O to generally denote an OCT set, and given an OCT set for a graph $G = (V, E)$, we let $B = V \setminus O$. The subgraph induced on B is bipartite with parts L and R; and we let $n_L = |L|$, $n_R = |R|$, and $n_O = |O|$ for an OCT decomposition (O, L, R). While finding a smallest OCT set is NP-hard, efficient parameterized [Iwa14; Lok09] and approximation [Aga94] algorithms exist. We point out that the approximation factor in [Aga94] is $O(\sqrt{\log(n)})$, and thus using this as our editing algorithm in structural rounding leads to an approximation factor which is $\Omega(1)$. For the sake of testing the efficacy of structural rounding, the OCT sets do not need to be optimal. We make use of a simple algorithm to find OCT sets: find two disjoint maximal independent sets and let the remaining vertices be our OCT set. See Section 5.5.2 for more details on how we carry out this process.

5.3 Lifting Algorithms

In this section, we describe different lifting strategies for using structural rounding on VERTEX COVER. To formalize notation, we let S' denote the solution on the edited-to instance G', and let D be the edit set. We let $I' = G' \setminus S'$, the vertices not added to our cover on G' (see Figure 5.1). Each lifting strategy finds a set S_L such that $S_L \cup S'$ is a solution on G. Since we work with vertex deletions in this

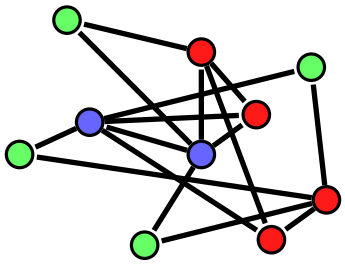

Figure 5.1 An example of structural rounding when vertex deleting to bipartite graphs while solving VERTEX COVER. The red vertices are the edit set D, the blue vertices the partial solution S', and the green vertices, I', are the ones not added to the partial cover. Note that S' is an optimal vertex cover on $G[S' \cup I']$, which is bipartite.

paper, $D \subseteq V(G)$ and $V(G') \cup D = V(G)$. For a given S', we use O_L to denote the optimal lift (set of minimum size such that $S' \cup O_L$ is a vertex cover of G), which is used to measure the quality of the various lifting strategies when appropriate.

5.3.1 Naïve Lifting

The simplest lifting strategy, `naïve` (from [Dem19]), is to let $S_L = D$. Clearly all edges in $G[I \cup D]$ have at least one endpoint in D, thus this is a valid strategy. It also aligns with Theorem 7 and Definition 7 giving lifting constant $c = 1$. However, there are no guarantees about the quality of this technique relative to O_L. Depending on the structure of D, $|O_L|$ may be $\Theta(|D|)$ or $\Theta(1)$, meaning that `naïve` could be near-optimal, or very far from it.

5.3.2 Greedy Lifting

A simple improvement upon `naïve` is to avoid inclusion of unnecessary vertices which already have all of their neighbors in the cover, which we refer to as the `greedy` technique. We achieve this by iterating over D in an arbitrary order, maintaining a list of vertices outside the cover U, where initially $U = I$ and $L = \emptyset$. For each vertex v, if it has a neighbor in U we add it to L, otherwise we add it to U. Thus, in order for a vertex to be added to L, it must cover an edge where the other endpoint is not a part of the solution. Note that this strategy takes $O(m)$ time. While the returned solution is guaranteed to be no larger than the solution returned by `naïve`, it still suffers from not having any guarantee on solution quality relative to O_L.

5.3.3 2-Approximation Lifting

While the two aforementioned strategies are good candidates for their simplicity and efficiency, we would prefer an approach with a theoretical guarantee. One natural candidate for lifting is using one of the VERTEX COVER 2-approximations from Section 5.2 on $G[I \cup D]$, referred to as `apx`. Such a lift is guaranteed to be at most twice the size of O_L. However, we note this may return solutions which

are larger than D (and in practice, one should take the smaller value, resulting in a valid structural rounding pipeline with $c = 1$ again).

5.3.4 `oct-first` Lifting

A natural extension to running a 2-approximation on $G[I \cup D]$ is to instead run it on $G[D]$, obtaining a partial lift solution, L_2. Then, if we let the nonselected vertices from D be denoted as D', $G[D' \cup I]$ is a bipartite graph, and we can exactly solve in polynomial time [Hop73] and find L_1. Note that $L_2 \cup L_1$ form a valid lifting solution. We are able to compute an approximation factor on the quality of this lift by comparing the sizes of L_1 and L_2.

Note that because each edge branched on in `standard` (or in the matching if using `DFS`) is independent, at least one vertex per edge must be included as part of any vertex cover. Thus, any lifting solution must have size at least $|L_2|/2$. Further, all of these edges are independent from the edges in $G[D' \cup I]$. When covering the edges in $G[D' \cup I]$, no fewer than $|L_1|$ vertices can be used. Thus, any valid lifting solution must contain at least $|L_2|/2 + |L_1|$ vertices and $|O_L| \geq |L_2|/2 + |L_1|$.

If we let p be the proportion of vertices added during Hopcroft-Karp ($p = |L_1|/(|L_1| + |L_2|)$), we can rewrite our lower bound on $|O_L|$ as $(|L_1| + |L_2|)((1-p)/2 + p) = (|L_1| + |L_2|)(1+p)/2$. Thus, $2|O_L|/(1+p) \geq |L_1| + |L_2|$, and `oct-first` lifting yields a $2/(1+p)$-approximation. Note that the approximation factor will range from 1 (exact; when all vertices come from Hopcroft-Karp) to 2 (when no vertices come from Hopcroft-Karp).

5.3.5 `bip-first` Lifting

A logical alternative to `oct-first` lifting is to flip the order of the steps, and first solve on the bipartite subgraph induced on the edges between I and D before running a 2-approximation on the remaining edges. We refer to this method as `bip-first` lifting. Once again, we let the set of vertices which are given by the bipartite (exact) phase be denoted L_1 and the vertices given by the approximation portion of the algorithm be denoted L_2.

Unfortunately, we are unable to obtain an approximation factor as we did with `oct-first` lifting. While it is clear that any lift must contain at least L_1, we cannot make any stronger guarantee. As a simple example, if the edges between I and D form an even-length cycle and every other vertex on the cycle has a pendant, Hopcroft-Karp may return the cycle vertices without pendants. Then, `bip-first` lifting would select all of the vertices, which is three times more than was necessary.

If we define p as we did before ($p = |L_1|/(|L_1| + |L_2|)$), we can still obtain an approximation factor better than 2 when p is large, namely $1/p$. Thus, when $p > 0.5$, we get better than a 2-approximation. We speculate that this method will still prove useful in practice as there are likely to be many edges between I and D when the edit set is found via any reasonable technique.

5.3.6 Recursive Lifting

One final type of lift comes from applying structural rounding as the lifting method. That is, after finding the partial solution in G', run structural rounding again. The precise manner in which this occurs can be done in several different ways. The first, `recursive`, is to completely re-run structural rounding on $G[I \cup D]$ by finding a new edit set on this subgraph and repeat. The second, `recursive-oct` , is to run structural rounding on $G[D]$, after which the only uncovered edges will form a bipartite graph between D and I which can be solved exactly. Alternatively, `recursive-bip` first exactly solves on the edges between D and I and then runs structural rounding on any uncovered edges in $G[D]$. For each of these methods, we use `naïve` in the structural rounding component of the approach, but any of the aforementioned lifting techniques could be used. Further, while we only do one recursive step, one could recursively use structural rounding as the lift until exhaustion. It is worth noting that the approximation factor of these approaches depends on how well you are able to edit to the class of interest when rounding.

5.4 Experimental Setup

In this section, we describe our datasets, along with how we measure approximation quality, and the hardware used for all experiments. In order to experimentally evaluate the effectiveness of structural rounding, we executed each of the aforementioned VERTEX COVER approximation algorithms on a large corpus of synthetic “near-bipartite" graphs with varied structure (controlled by generator parameters, see Section 5.4.1), as well as a collection of 130 real-world networks from diverse domains.

5.4.1 Synthetic Data

Recall that we use n_O, n_L, and n_R to denote the sizes of O, L, and R in an OCT decomposition where O is the OCT set and L and R are the bipartite parts. For convenience, throughout this section, we assume $n_L \geq n_R$ and let $n_B = n_L + n_R$. Our synthetic data was generated using a modified version of the random graph generator of Zhang et al. [Zha14b] that augments random bipartite graphs to have OCT sets of known size. The generator allows a user to specify the sizes of L, R, and O, the expected edge densities between L and R, O and $L \cup R$, and within O, and the coefficient of variation ($c\,v$; the standard deviation divided by the mean) of the expected number of neighbors in L over R and in $L \cup R$ over O. The generator is seeded for replicability.

Initially, we created graphs with each combination of $n_L/n_R \in \{1, 2, 10, 100\}$, expected edge density within $O \in \{0.001, 0.01, 0.05\}$, between O and $L \cup R \in \{0.01, 0.05\}$, and between L and $R \in \{0.001, 0.005, 0.01\}$, $n_O/n \in \{0.01, 0.05, 0.1, 0.25, 0.4\}$, and $c\,v$ of the expected number of neighbors in $L \cup R$ over $O \in \{0.5, 1.5\}$. We set n_L, n_R, and n_O based on the other parameters so that the expected number of edges was 4 million.

We also generated graphs with 40 million and 200 million expected edges after paring down the parameter space to eliminate variants with little to no impact on solution quality. We provide

Table 5.1 Summary statistics of real-world corpus.

	n	m	avg. degree	$\lvert O\rvert/n$
min	9.000	8.000	1.77	0.000
max	556686.000	987091.000	326.851	0.984
median	1104.500	2984.000	4.867	0.256
mean	18859.164	64881.578	14.092	0.343

more details on the parameter down-selection in Section 5.6.1. On the larger graphs, we only ran the 2-approximations, `naïve` and `greedy` lifting, and the most effective lifts from the 4 million edge data, `oct-first` and `recursive-oct` . We also created a collection of synthetic graphs with 100,000 edges over the entire parameter space for the purpose of testing different implementations.

We further distinguish two scenarios in the 4M-edge corpus. While each graph has an OCT decomposition which corresponds to the specified parameters given by the generator, it does not mean that the OCT heuristic employed will find this partition. Thus, we ran our algorithms on the 4 million edge graphs with two approaches; (i) with the OCT decomposition *prescribed* to be the one given by the generator (4M-pre) and (ii) with an OCT decomposition which is *procured* by our heuristic (4M-pro). We observed that differences in outcomes were minor (see Figure 5.4) and use only approach (ii) on the 40M and 200M corpuses because of it being a more realistic scenario in practice. The prescribed approach does have the benefit of allowing us to determine the impact that the different generator parameters had on the solution quality of each strategy, which we observe in Section 5.6.1.

5.4.2 Real World Data

The real world corpus is a set of 130 graphs from various domains including social networks, physical infrastructure, biology, diseases, and compilation dependencies taken from [Net]. These networks vary widely in size, average degree, and relative OCT size (see Table 5.1). We note that despite the variation in average degree, most of the graphs in the real world corpus are substantially sparser than any of the synthetic graphs used.

5.4.3 Measuring Quality

In order to measure the quality of an approximate solution, we compute the ratio of its size to half of the “worst-case" 2-approximation (the maximum cover size found by any 2-approximation run); we refer to this as the *approximation ratio*. For example, if some 2-approximation returns a cover of size 100, we know an optimal solution has at least 50 vertices, and a cover of size 75 is at-worst a 1.5-approximation.

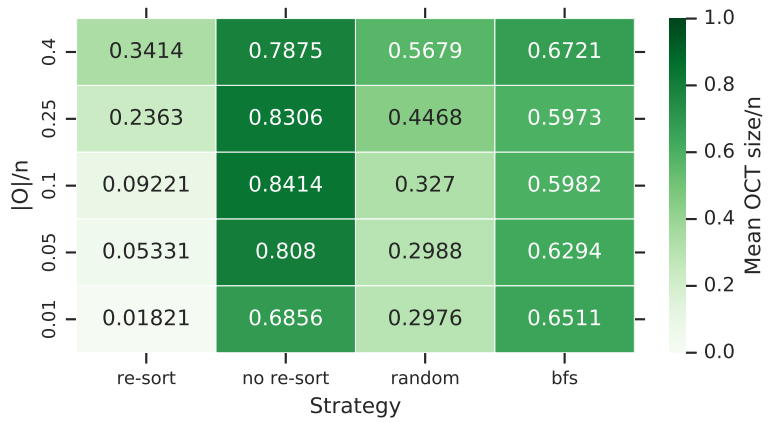

Figure 5.2 Mean relative OCT sizes for OCT decomposition algorithms on the 100K corpus, partitioned by prescribed OCT ratio. Approaches include a BFS-based technique (bfs), and the construction of two maximal independent sets with three vertex selection rules: arbitrarily (random), by minimum initial degree (no re-sort), and by minimum degree with re-sorting (re-sort).

5.4.4 Hardware

All experiments were run on identical hardware; each server had four Intel Xeon E5-2623 v3 CPUs (3.00GHz) and 64GB DDR4 memory. The servers ran Fedora 27 with Linux kernel 4.16.7-200.fc27.x86_-64. The code is entirely written in Python 3 and run using version 3.6.5. Our code is open source under a BSD 3-clause license and is publicly available [Lav19].

5.5 Implementation

In this section, we provide details on algorithm implementation along with data supporting our experimental design decisions.

Table 5.2 Mean solution sizes for and standard deviations of 2-approximation variants on the 100K corpus. We employ three edge selection techniques for `standard`: incident to a high-degree vertex (`std-high`), incident to a low-degree vertex (`std-low`), and arbitrarily (`std-rand`). We note that the large standard deviations are due to the relatively high variety of graphs in the corpus.

	Method			
	`std-high`	`std-low`	`std-rand`	`DFS`
avg. size	3340.617	3647.758	3531.626	3397.374
std. dev.	2337.914	2525.158	2404.848	2301.961

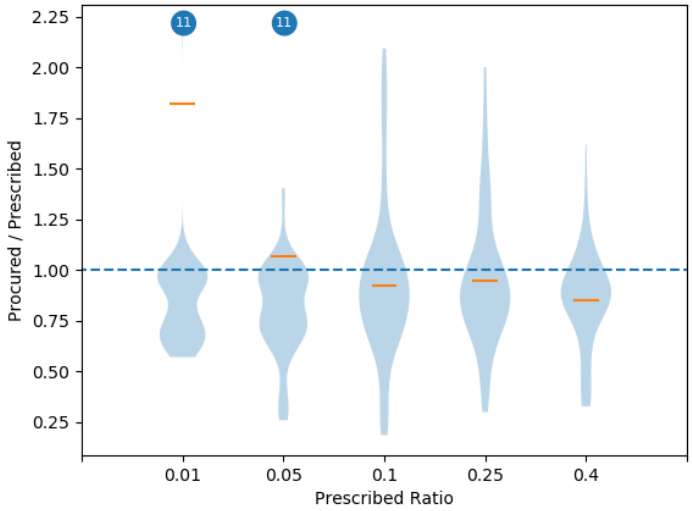

Figure 5.3 Distribution of procured OCT sizes relative to prescribed OCT sizes found on the 100K corpus. The mean of each distribution is given by the orange bars on the plot. 11 outliers were collected and shown at 2.25 in the first two distributions to preserve the scale. Note that these values are still considered in the mean. A ratio of 1 or smaller indicates that the heuristic OCT set approach found a better OCT set than the one prescribed by the generator.

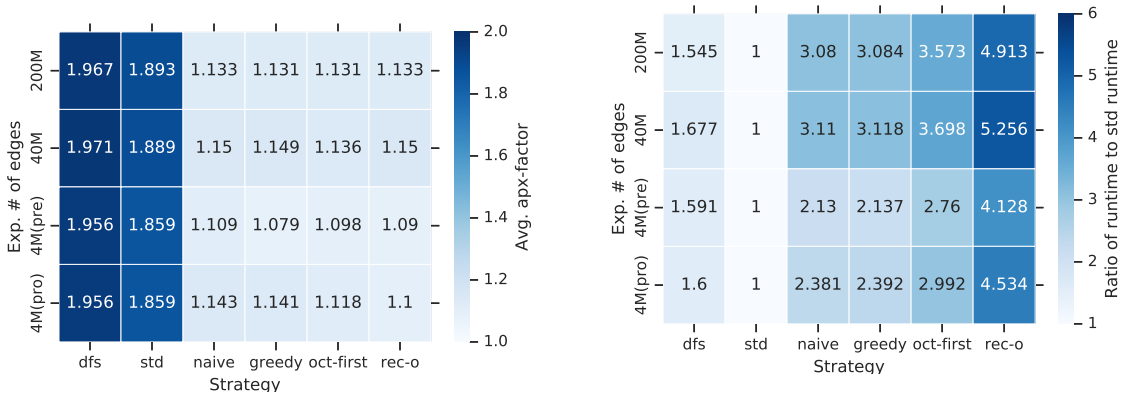

Figure 5.4 Average approximation ratios (left) and relative runtime (right) on the full synthetic corpus, partitioned by expected number of edges. For 4M edge graphs, we use both procured (“pro") and prescribed (“pre") OCT decompositions. Runtimes are given relative to that of `standard`, which is fastest on average (mean runtimes: 1.511s on 4M-pro, 1.502s on 4M-pre, 20.551s on 40M, and 136.779s on 200M). For structural rounding approaches, the time it takes to find the OCT decomposition *is* accounted for in the runtime (note that for 4M-pre, this contributes zero since the decomposition is given as input).

5.5.1 2-Approximation Variants

We tested four different 2-approximations for VERTEX COVER including `DFS` of [Sav82] and three variants of `standard` over all of the synthetic graphs with 100K edges in expectation. Specifically, the three `standard` variants differed on how edges were selected. The edge selection techniques tested were random selection, a random edge incident on a maximum degree vertex, and a random edge incident on a minimum degree vertex which we will refer to as `std-rand`, `std-high`, and `std-low` respectively. As seen in Table 5.2, the best performing 2-approximation for VERTEX COVER was generally to use `std-high`. This approach almost always outperformed `std-rand` on the synthetic graphs. Conversely, `std-low` almost always performs worse, providing us with the highest possible lower bounds on the optimal solution size. Using `DFS` did occasionally produce smaller solutions than `standard`. However, in over 50% of parameter settings it performed worse, and likewise on average, it produced larger solutions. Since `DFS` is also randomized, we chose to use `standard` to maximize performance and not require additional trials for each graph.

5.5.2 Finding OCT Sets

We compared two heuristic approaches to finding small OCT sets on the 100K corpus. We found that greedily constructing two maximal independent sets generally outperformed using a BFS-search as seen in Figure 5.2. We note that the performance of the maximal independent set heuristic crucially depends on not only choosing minimum degree vertices first, but also on updating the degree of each vertex and re-sorting as vertices are added to the independent set. When the vertices are not re-sorted or are chosen randomly, we frequently found OCT sets that far exceeded what we expected based on the generator's parameter settings. When testing the BFS-coloring heuristic, we notice that despite offering a speed advantage over the re-sorting algorithm, the OCT sets produced are almost always substantially larger. Furthermore, BFS-search loses its speed advantage when random vertices are chosen while still producing larger OCT sets. Since heuristically finding an OCT set is rarely the bottleneck in a structural rounding approximation, we opted to use the maximal independent set strategy with re-sorting.

Even when we use the maximal independent set heuristic with re-sorting, we have no guarantees that the algorithm will produce the same OCT set as the one prescribed by the generator. In Figure 5.3 though, we show that in the majority of graphs in the 100K corpus, the heuristic OCT algorithm finds OCT sets of comparable size to those prescribed by the generator.

5.5.3 Converting Maximum Matchings

Unlike the popular `NetworkX` package in Python, our implementation for converting a maximal matching found in a bipartite graph to a vertex cover is done iteratively and not recursively, allowing it to be somewhat faster. We avoid using recursion when possible in all functions in our codebase, particularly in implementations of depth-first search.

5.5.4 Variance Between Runs

We ran each algorithmic component of our framework 50 times on the 100K synthetic corpus to test the amount of variance in the runtimes. In all cases, the maximum ratio of variance to mean averaged over the corpus was at most 0.0004, giving us confidence that a single trial of each graph-algorithm combination would result in a representative runtime. We include all of the variance results in Table 5.3.

Table 5.3 We ran our framework 50 times on each graph in our corpus with 100K expected edges, and for each component of the framework we computed the variance of its runtimes divided by its mean runtime for each graph. We report the average of this value over all graphs for each component of the framework.

module	avg. σ^2/μ
DFS 2-apx	2.50e-05
Std. 2-apx	2.24e-05
OCT-finding	1.38e-04
bipartite-solve	7.86e-05
`naïve` lift	4.53e-07
apx lift	1.18e-05
`greedy` lift	7.55e-07
`oct-first` lift	1.82e-04
`bip-first` lift	2.44e-04
`recursive` lift	3.86e-04
`rec-oct` lift	4.89e-04
`rec-bip` lift	2.83e-04

5.6 Experimental Evaluation

We begin by describing the three key findings from our empirical evaluation of the structural rounding framework, then discuss in greater detail the impact of structural variation in the input (as prescribed by generator parameters, Section 5.6.1), results in the procured decomposition setting (Section 5.6.2), and finally results on the real-world corpus (Section 5.6.3).

1. *Structural rounding consistently outperforms traditional approximation techniques.* Both `standard` and DFS perform considerably worse than structural rounding-based methods when averaged over our entire corpus of graphs in both the prescribed and procured settings (left panel of Figure 5.4). Because structural rounding is a more involved technique, it tends to take at least twice as long to run compared to the 2-approximations (right panel of Figure 5.4), but absolute runtimes remain very low and usable in practical settings. The bottleneck for this application of structural

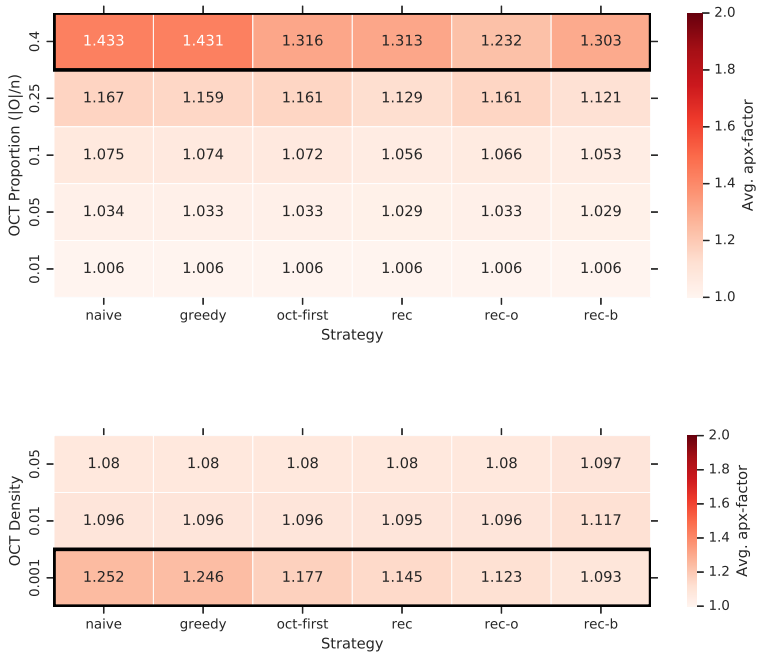

Figure 5.5 Two settings when more sophisticated lifting strategies outperform `naïve` and `greedy` on the 4M-pro data. We partition graphs based on two generator parameters: relative OCT set size (top), and expected edge density within O (bottom).

rounding is solving vertex cover exactly in the edited graph which can take up to 30 seconds in the 200M corpus. Both `naïve` and `greedy` lifting finish almost instantaneously even in the 200M corpus.

2. `Naïve` *and* `greedy` *are often comparable with more sophisticated lifting approaches.* The quality of the simpler lifts for VERTEX COVER are roughly the same (and sometimes better) than that of the `oct-first`, `bip-first`, and the recursive lifts (left panel of Figure 5.4). Given the additional time (right panel of Figure 5.4) and memory the more advanced lifts require, `greedy` combines the efficiency of traditional techniques and the accuracy of more sophisticated lifting.

3. *There are settings where the more sophisticated lifting approaches outperform* `naïve` *and* `greedy`. In the procured setting, the smarter lifting techniques outperform `naïve` and `greedy` by the most when the ratio n_L/n_R is largest, the $c\,v$ between O and $\{L,R\}$ is largest, the expected edge density in O is smallest, and the proportion of the graph in O is largest (see Figure 5.5 for examples of the latter two). This behavior is seen on a lesser scale on the larger graphs. In the prescribed setting, the difference is less pronounced and `recursive-bip` tends to distinguish itself the most from `naïve` and `greedy`, namely when the ratio n_L/n_R or the $c\,v$ between O and $\{L,R\}$ is largest, or the proportion of the graph in O is smallest.

Throughout our results we color our figures based on the dataset they represent as follows; blue is an aggregate view of the large data ($\geq$ 4M expected edges), green is the supplementary data (100K

expected edges), red is the procured setting of the 4M expected edge data (4M-pro), purple is the prescribed setting of the 4M expected edge data (4M-pre), orange is the 40M expected edge data, and grey is the 200M expected edge data.

5.6.1 Impact of Generator Parameters

We now discuss how each of the generator parameters impact the various lifting methods. We utilize the prescribed data (4M-pre) to measure these effects, as this allows us to isolate the variables.

$|L|/|R|$. When this ratio is 2, 10, or 100, `recursive-bip` lifting is the optimal strategy. However, when it is 1, it does significantly worse, and `greedy` is the best option. It is likely that this is because I, the set of vertices not selected when solving on the edited-to graph $G[L \cup R]$, is smaller than in the other cases. Then when `recursive-bip` solves on the edges between X and I it picks up most of I, which is unnecessary because most of these edges will be covered by the solution on X. When compared to the worst-case 2-approximation, all of the structural rounding methods do best when the ratio is 2 and 10. Due to the lack of distinction between these two settings, we only use ratios of 1, 10, and 100 when generating larger graphs. `DFS` monotonically worsens as this ratio increases, and is always worse than `standard`. Figure 5.6 contains the data relevant to the impact of $|L|/|R|$.

cv between O and $\{L,R\}$. `recursive-bip` is the best method in the higher setting, while `greedy` is best in the lower setting. In general, the approximation factors of the structural rounding methods are lower when the cv is lower. Interestingly, there is no difference between the settings when using 2-approximation lifting. Both 2-approximations also do better when the cv is smaller. Figure 5.7 contains the data relevant to the impact of the cv between O and $\{L,R\}$.

$|O|/n$. For the four smallest settings, `recursive-bip` is among the best approximations. However, for the largest OCT ratio (0.4), `greedy` is the optimal approach, and `naïve`, `oct-first`, and `recursive-oct` are all more effective than `recursive-bip` . This shift is likely due to the edit set X being larger relative to I, the set of vertices not selected when solving on the edited-to graph $G[L \cup R]$. Similar to our observations when varying $|L|/|R|$, it is likely that `recursive-bip` unnecessarily adds a majority of I, inflating its solution size. Figure 5.8 contains the data relevant to the impact of $|O|/n$.

Expected edge density in O. `recursive` is the best strategy in the highest density setting, while `greedy` does best in the two lower settings. In fact, `recursive` is by far the worst structural rounding approach for the lowest OCT density setting. This is likely due to the extreme sparsity in the edit set in this setting, and thus `recursive` finds a large OCT set when re-running structural rounding on $G[I \cup X]$, which in turn gets completely added to the solution due our using `naïve` for this step. One particularly interesting observation is that the structural rounding approaches tend to be split on whether they are best at higher or lower density settings, and five of them are worst on the moderate setting. This behavior was only observed for `recursive-bip` in the procured setting, and due to the minor difference between 0.01 and 0.05, we only used the two extreme settings when generating larger graphs. Both 2-approximations do best when the density is higher. Figure 5.9 contains the

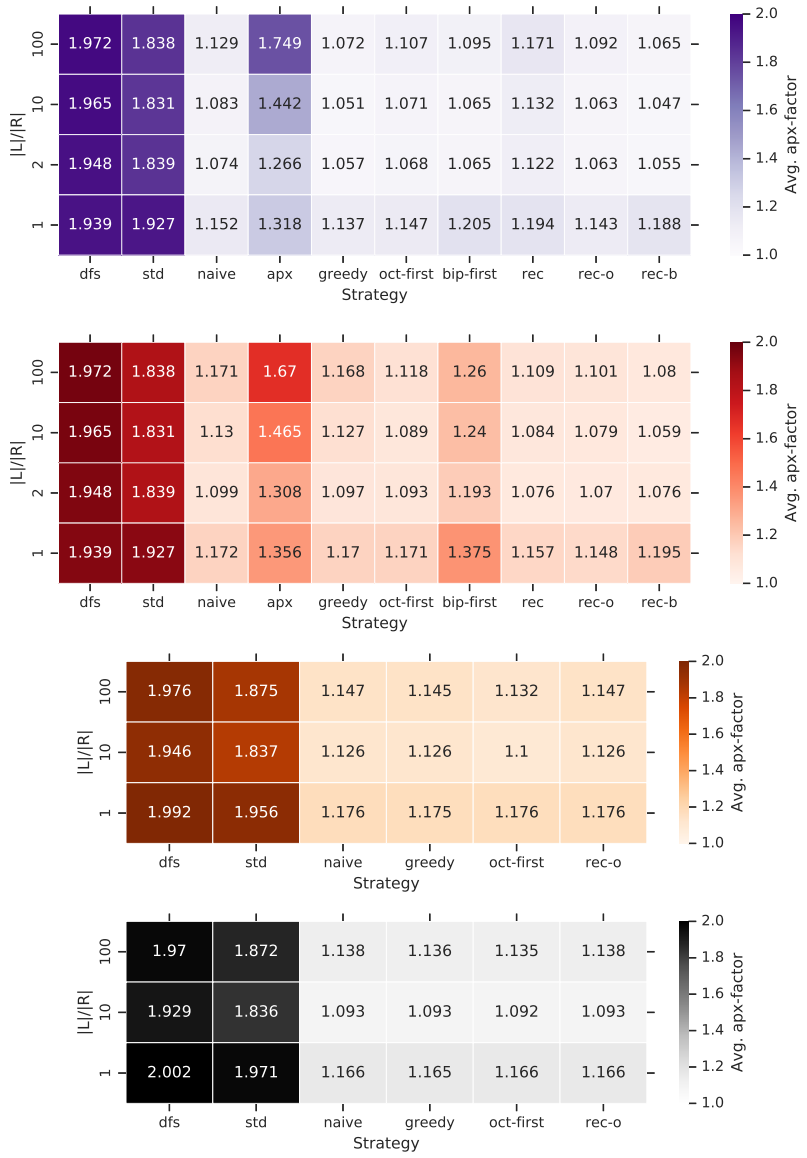

Figure 5.6 The average approximation ratios over all synthetic graphs with 4-million expected edges in the prescribed setting, with 4-million expected edges in the procured, with 40-million expected edges, and with 200-million expected edges separated by $|L|/|R|$.

data relevant to the impact of the expected OCT density.

Expected edge density between O and $\{L,R\}$. In both settings, `greedy` is the best lifting strategy. All of the structural rounding approaches have better approximation factors when this expected density is smaller, as does DFS. However, `standard` does better in the larger density setting. The magnitude of these differences tends to be small, so we did not vary this value on the larger graphs. Figure 5.10 contains the data relevant to the impact of the expected edge density between O and $\{L,R\}$.

Expected edge density between L and R. Once again, `greedy` is the optimal strategy in all settings. There is minimal impact of the quality of structural rounding when compared to the

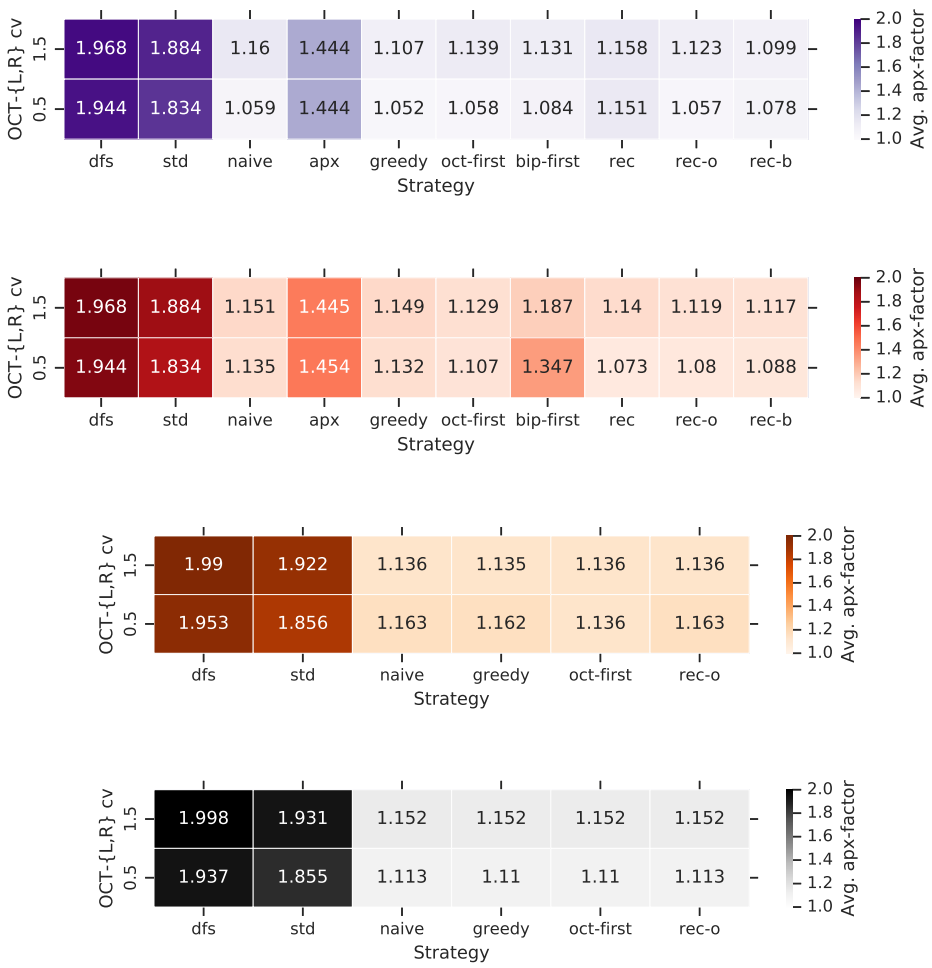

Figure 5.7 The average approximation ratios over all synthetic graphs with 4-million expected edges in the prescribed setting, with 4-million expected edges in the procured, with 40-million expected edges, and with 200-million expected edges separated by the expected $c\,v$ between O and $\{L,R\}$.

worst-case 2-approximation, and thus we did not vary this parameter on the larger graphs. The 2-approximations also did not vary much based on this density. Figure 5.11 contains the data relevant to the impact of the expected density between L and R.

5.6.2 Procured Results

While our findings on the procured data are generally consistent with the prescribed data, we observe the following anomalies. When $n_L/n_R = 2$, `recursive-oct` does slightly better than `recursive-bip` on the 4M edge graphs, while on the larger graphs `oct-first` is the optimal approach for the larger ratio settings. When the expected number of edges is 40M or 200M, DFS does best when $n_L/n_R = 10$. `recursive` is the optimal strategy when the $c\,v$ between O and $\{L,R\}$ is 0.5 for the graphs with 4M edges, while `oct-first` is preferred in this setting on the larger graphs. When $n_O/n = 0.4$, the best approach on the 4M edge graphs is via `recursive-oct` , while `oct-first` and

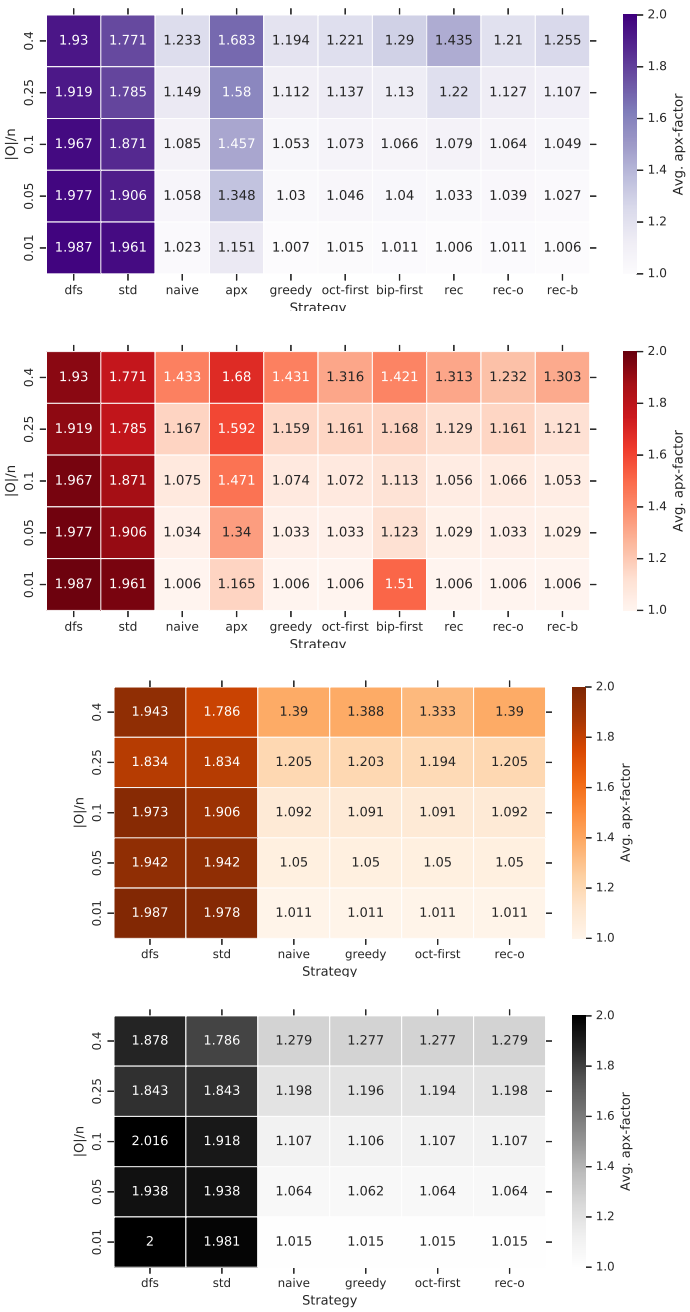

Figure 5.8 The average approximation ratios over all synthetic graphs with 4-million expected edges in the prescribed setting, with 4-million expected edges in the procured, with 40-million expected edges, and with 200-million expected edges separated by the proportion of the graph which is in O.

`greedy` perform best on the 40M and 200M edge graphs under all settings. `recursive` is not the worst-performing method on the 4M graphs when the expected density with OCT is 0.0001, which is likely due to the observed OCT set not having this extreme sparsity. There is little difference over the OCT density among the structural rounding approximations on the 200M graphs, while `oct-first` is best on the 40M graphs when the OCT density is low. `recursive-oct` is the best strategy on the 4M edge graphs when expected density between O and $\{L, R\}$ is higher, while `recursive` is optimal

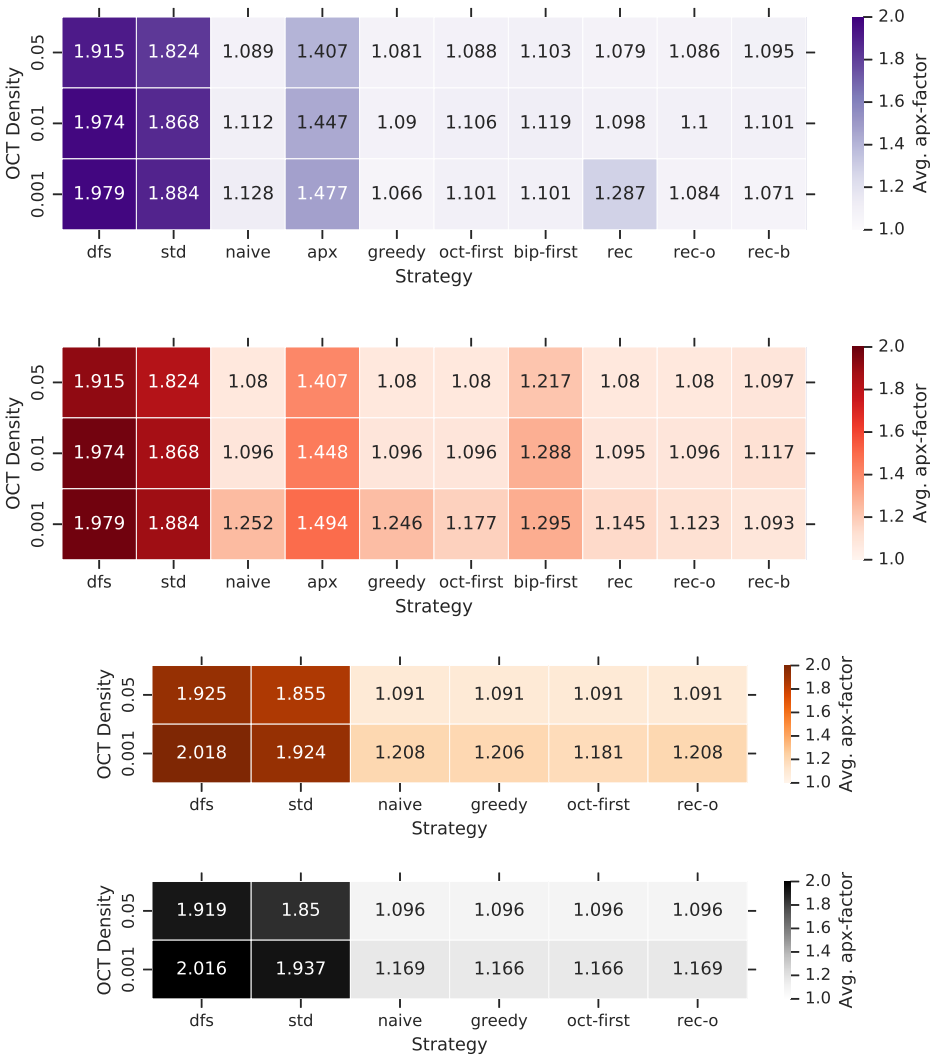

Figure 5.9 The average approximation ratios over all synthetic graphs with 4-million expected edges in the prescribed setting, with 4-million expected edges in the procured, with 40-million expected edges, and with 200-million expected edges separated by the expected edge density within O.

for the lower setting. Regarding the expected edge density between L and R, `recursive-oct` is the best lifting technique under all settings.

We observe similar behavior at all three scales of the procured data when we distinguish the graphs based on specific generator parameter values. This is highlighted in the data in Figure 5.12 where we split the data based on the generator value of n_O/n. Generally the graphs of all three sizes behave comparably when the observed size of O is close to that of the generator. When the procured proportion is greater than the prescribed value, we see more distinction among the sizes, as generally the found OCT sets are larger in the 40M and 200M settings (in all but the $n_O/n = 0.4$ setting), resulting in slightly worse solution quality in the larger graphs.

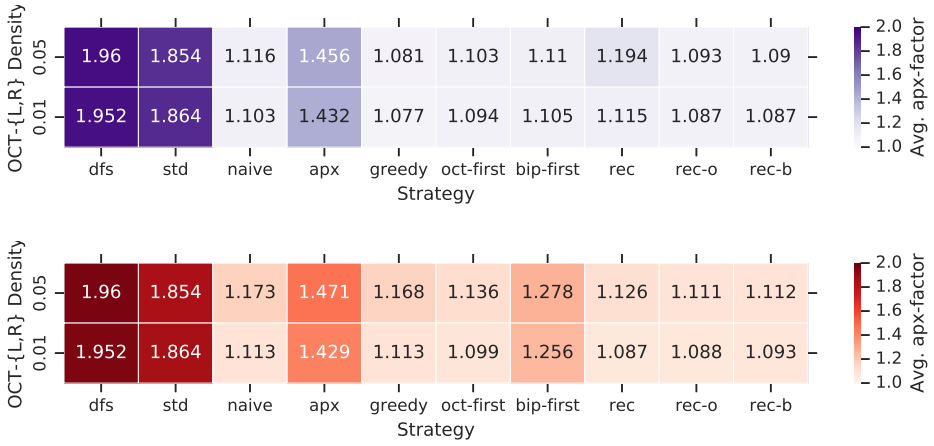

Figure 5.10 The average approximation ratios over all synthetic graphs with 4-million expected edges in the prescribed setting and in the procured separated by the expected edge density between O and $\{L, R\}$.

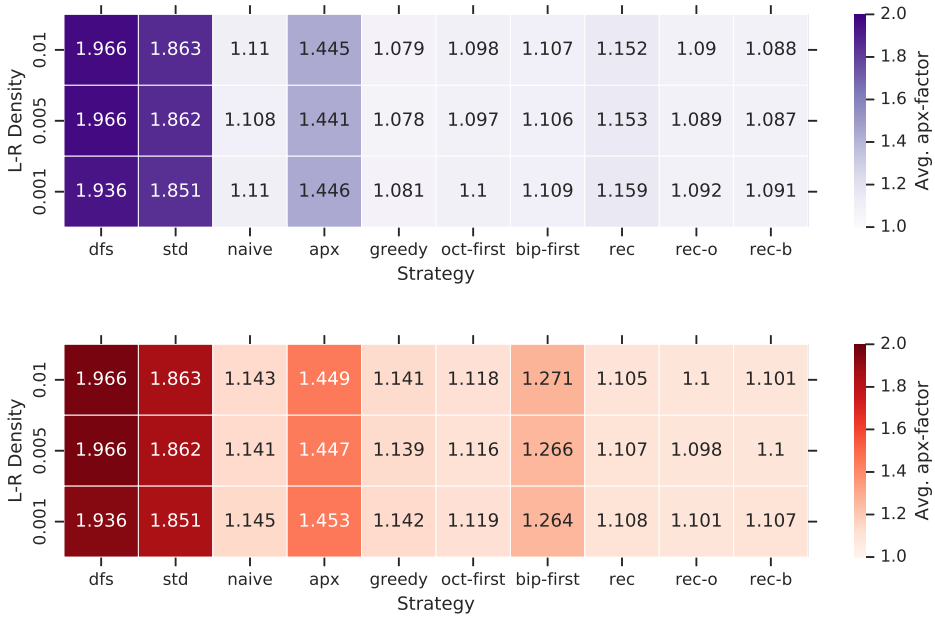

Figure 5.11 The mean approximation ratios over all synthetic graphs with 4-million expected edges in the prescribed setting and in the procured setting, separated by the expected edge density between L and R.

5.6.3 Real-world Experiments

Lastly, we ran our collection of algorithms on a corpus of 130 real-world graphs compiled from various scientific, sociological, and technological domains. Since these graphs are much sparser than any of the synthetic graphs we tested (average degree less than 2 in some cases), finding the OCT set became a bottleneck in a few of the larger graphs. To remedy this, we used random edge selection in the heuristic OCT algorithm to keep runtimes low. Additionally, since the graphs are so sparse,

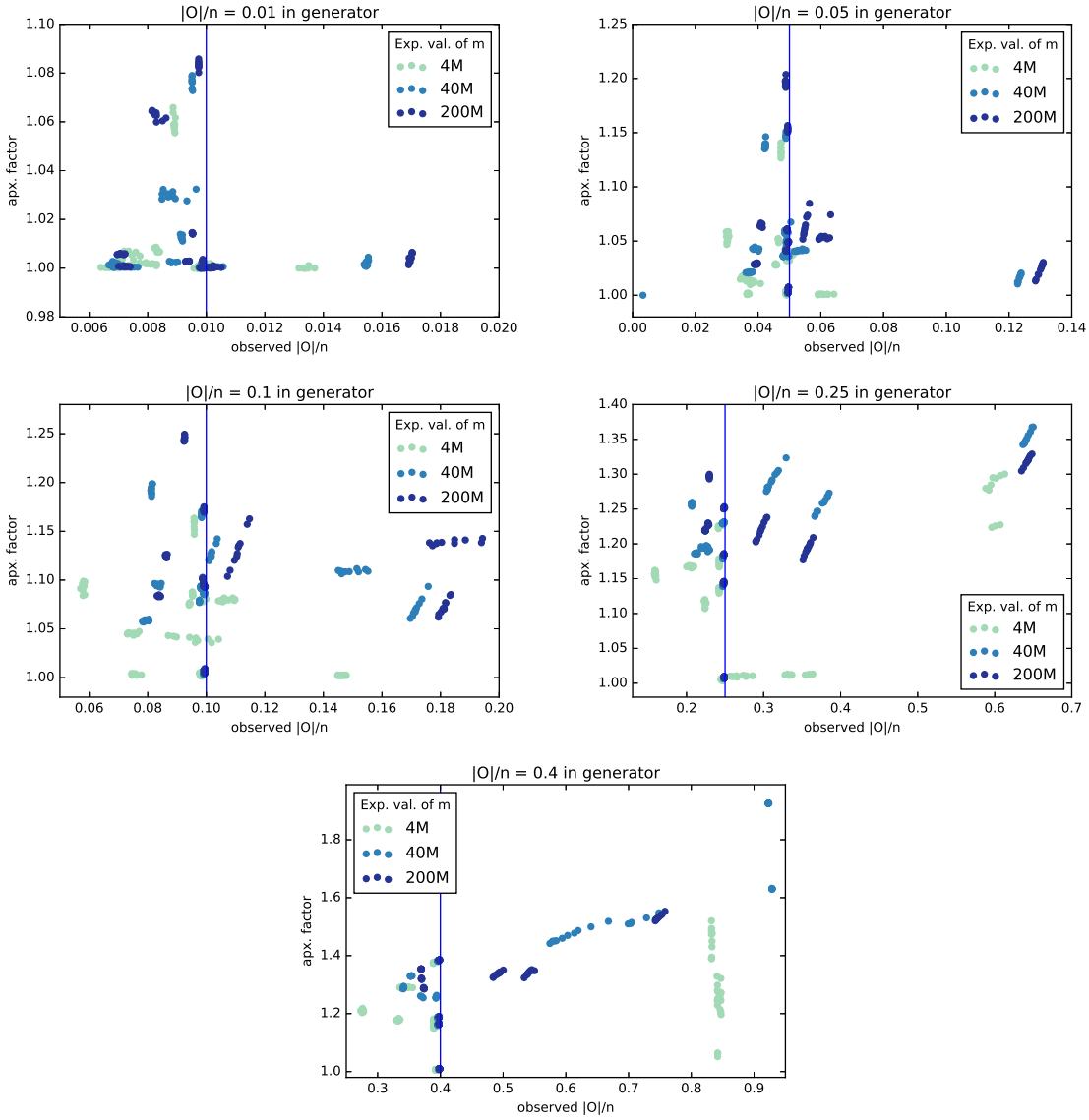

Figure 5.12 Correspondence between the size of the procured OCT set and solution quality across 4M-pro (green), 40M (blue), and 200M (red) corpuses; only graphs generated with parameters used in all experiments are included for consistency. The data is partitioned by prescribed relative OCT size $|O|/n$. We observe that performance is most consistent when the procured decomposition has OCT size close to the prescribed value.

this choice had minimal impact on the performance of the structural rounding algorithms. Overall, we see that structural rounding outperforms the 2-approximations. In Figure 5.13, we observe that structural rounding performs better in a large number of graphs across all sizes. Furthermore, we see that when graphs have small OCT sets, structural rounding consistently produces smaller solutions. Even on graphs with large OCT sets, structural rounding matches the solution quality of the 2-approximations, delivering well beyond its comparatively weak theoretical guarantees. Lastly, we note that the structural rounding algorithms finish reasonably quickly, taking less than 10

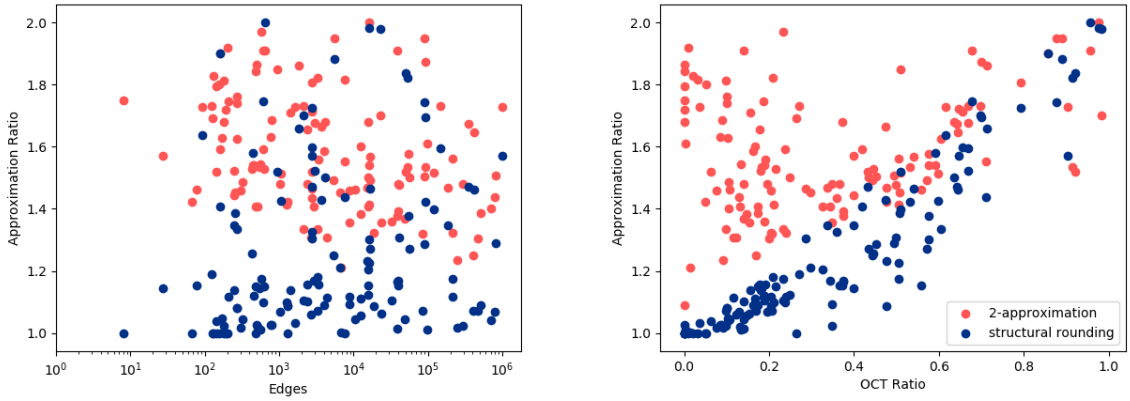

Figure 5.13 The best approximation ratios achieved by 2-approximations (in red) versus structural rounding (in blue) on graphs from the real-world corpus. The ratios are computed by comparing the best 2-approximation or structural rounding result with the worst 2-approximation. The 2-approximations considered are DFS, std-high, and std-low. The lifting algorithms used are `greedy`, `oct-first`, and `bip-first`. On the left, we plot these ratios against the number of edges in the graph on a log scale, while on the right, we plot the ratios against the proportion of the graph in the OCT set.

seconds to run on graphs with nearly a million edges and over half a million vertices.

5.7 Conclusion

In this work we provide the first implementation and empirical evaluation of structural rounding and demonstrate that structural rounding provides better approximations for VERTEX COVER than traditional approaches on a large corpus of widely varying graphs. These results highlight the importance of experimentally evaluating theoretical approaches. While the structural rounding framework for VERTEX COVER under vertex deletion to bipartite graphs has weaker theory guarantees than the traditional methods, its practical performance is much stronger. We introduced a suite of more sophisticated lifting algorithms but showed that the simple greedy approach was extremely competitive in most scenarios

While this paper concerns itself with editing to bipartite graphs and solving VERTEX COVER, there are an abundance of interesting and unexplored directions. One might consider editing to graph classes other than bipartite, e.g. classes suggested in [Dem19] including bounded degeneracy and bounded treewidth. The promise of more sophisticated lifting approaches on other problems such as INDEPENDENT SET, DOMINATING SET, and FEEDBACK VERTEX SET is also unclear. While we are interested in finding new lifting algorithms for problems other than VERTEX COVER, the success of `naïve` and `greedy` indicated in this paper leads us to believe that it is reasonable to anticipate similar results for other problems. Finally, we would be interested in comparing the effectiveness of structural rounding when using other edit types, namely edge deletion and edge contraction, with that of using vertex deletions.

CHAPTER

6

CONCLUSION & FUTURE WORK

In this dissertation, we developed and (when applicable) analyzed new graph modification algorithms for graphs near structural classes. We tackled three types of problems: finding an edit set via a parameterized approach, leveraging a given edit set to solve an enumeration problem, and using an edit set to efficiently and effectively obtain an approximate solution.

We began by designing a new parameterized algorithm for the k-Co-Path Set problem, which asks for a given value of k, if there exists a set of k edges whose deletion yields a graph where every connected component is a path. We made use of the Cut & Count framework to obtain a $O^*(4^{tw(G)})$ algorithm for the general problem, and combined this with branching and kernelization techniques to effectively bound the treewidth of the instances we work on. The resulting algorithm has complexity $O^*(1.588^k)$ for deciding k-Co-Path Set. It would be interesting to see if it is possible to extend our method to k-uniform hypergraphs or if parameterizing by the number of resulting disjoint paths and the size of the co-path set yields a sub-exponential fpt-algorithm.

Second, we consider the problem of enumerating maximal induced bicliques on near-bipartite graphs – those with small odd cycle transversals (sets of vertices whose deletion yields a bipartite graph). We developed two novel approaches for this problem when given an odd cycle transversal, the first extends from the best-known enumeration algorithm for bipartite graphs, while the second is based on placing restrictions on a new framework for enumerating maximal induced bicliques in general graphs. Empirical analysis identified unique cases where each of our algorithms was dominant. A more rigorous look at the performance of our algorithm suite as the size of the provided odd cycle transversal increases would yield further insight towards what constitutes being close to a structural class.

Finally, we then designed a suite of algorithms for carrying out the lifting phase of the structural

rounding framework. Structural rounding provides a new lens for designing approaches for graphs near a structural class, as we are again given an edit set, but now look to approximate the solution. In providing the first practical evaluation of structural rounding, we found that it vastly outperforms standard techniques in many settings when considering vertex deletion to bipartite graphs for VERTEX COVER. We also observed that the simpler lifting strategies are quite often extremely competitive with their more sophisticated counterparts. There is significant work to be done on experimentally evaluating structural rounding, including varying the edit operations, classes to edit to, and problems to approximate.

Overall, this dissertation extends the algorithmic frontier for approaches for graphs near structural classes and serves as a base for future exploration in this space. Research findings which answer the open problems mentioned above would continue to advance our knowledge of and increase the practicality of these types of algorithms.

BIBLIOGRAPHY

[AK04] Abu-Khzam, F. N. et al. “Kernelization Algorithms for the Vertex Cover Problem: Theory and Experiments.” *ALENEX/ANALC* **69** (2004).

[AK07] Abu-Khzam, F. N. et al. “Crown structures for vertex cover kernelization”. *Theory of Computing Systems* **41**.3 (2007), pp. 411–430.

[Aga05] Agarwal, A. et al. “$O(\sqrt{\log n})$) approximation algorithms for min UnCut, min 2CNF deletion, and directed cut problems”. *Proceedings of the thirty-seventh annual ACM Symposium on Theory of Computing*. ACM. 2005, pp. 573–581.

[Aga94] Agarwal, P. K. et al. “Can visibility graphs be represented compactly?” *Discrete & Computational Geometry* **12**.3 (1994), pp. 347–365.

[Aki16] Akiba, T. & Iwata, Y. “Branch-and-reduce exponential/fpt algorithms in practice: A case study of vertex cover”. *Theoretical Computer Science* **609** (2016), pp. 211–225.

[Ale04] Alexe, G. et al. “Consensus algorithms for the generation of all maximal bicliques”. *Discrete Applied Mathematics* **145**.1 (2004), pp. 11–21.

[Ban04] Bansal, N. et al. “Correlation clustering”. *Machine learning* **56**.1-3 (2004), pp. 89–113.

[Bjö07] Björklund, A. et al. “Fourier meets Möbius: fast subset convolution”. *Proceedings of the thirty-ninth annual ACM Symposium on Theory of Computing*. 2007, pp. 67–74.

[Che08a] Chen, J. et al. “A fixed-parameter algorithm for the directed feedback vertex set problem”. *Journal of the ACM (JACM)* **55**.5 (2008), p. 21.

[Che11] Chen, Z. et al. “An approximation algorithm for the minimum co-path set problem”. *Algorithmica* **60**.4 (2011), pp. 969–986.

[Che08b] Cheng, Y. et al. “The radiation hybrid map construction problem: recognition, hardness, and approximation algorithms”. Unpublished Manuscript. 2008.

[Chi85] Chiba, N. & Nishizeki, T. “Arboricity and subgraph listing algorithms”. *SIAM Journal on Computing* **14**.1 (1985), pp. 210–223.

[Cox90] Cox, D. et al. “Radiation hybrid mapping: a somatic cell genetic method for constructing high-resolution maps of mammalian chromosomes.” *Science* **250** (1990), pp. 245–50.

[Cyg11a] Cygan, M. et al. “Solving connectivity problems parameterized by treewidth in single exponential time”. *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*. IEEE. 2011, pp. 150–159.

[Cyg11b] Cygan, M. et al. “Solving connectivity problems parameterized by treewidth in single exponential time”. *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*. IEEE. 2011, pp. 150–159.

[Cyg15] Cygan, M. et al. *Parameterized algorithms*. Vol. 4. 8. Springer, 2015.

[Dab17] Dabrowski, K. K. et al. "Editing to a planar graph of given degrees". *Journal of Computer and System Sciences* **85** (2017), pp. 168–182.

[Dam06] Damaschke, P. "Parameterized enumeration, transversals, and imperfect phylogeny reconstruction". *Theoretical Computer Science* **351**.3 (2006), pp. 337–350.

[Daw01] Dawande, M. et al. "On bipartite and multipartite clique problems". *Journal of Algorithms* **41**.2 (2001), pp. 388–403.

[Dem19] Demaine, E. D. et al. "Structural Rounding: Approximation Algorithms for Graphs Near an Algorithmically Tractable Class". *27th Annual European Symposium on Algorithms (ESA 2019)*. Ed. by Bender, M. A. et al. Vol. 144. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 37:1–37:15.

[Dia05] Dias, V. M. et al. "Generating bicliques of a graph in lexicographic order". *Theoretical Computer Science* **337**.1-3 (2005), pp. 240–248.

[Dow12] Downey, R. G. & Fellows, M. R. *Parameterized complexity*. Springer Science & Business Media, 2012.

[Dra15] Drange, P. G. et al. "On the threshold of intractability". *Algorithms-ESA 2015*. Springer, 2015, pp. 411–423.

[Dra16] Drange, P. G. et al. "Compressing bounded degree graphs". *LATIN 2016: Theoretical Informatics*. Springer. 2016, pp. 362–375.

[Epp94] Eppstein, D. "Arboricity and bipartite subgraph listing algorithms". *Inf. Process. Lett.* **51**.4 (1994), pp. 207–211.

[Fen14] Feng, Q. et al. "Randomized parameterized algorithms for co-path set problem". *International Workshop on Frontiers in Algorithmics*. Springer, 2014, pp. 82–93.

[Fen15] Feng, Q. et al. "Kernelization and randomized Parameterized algorithms for Co-path Set problem". *J. Comb. Optim.* (2015). in press, DOI 10.1007/s10878-015-9901-y.

[Fom09] Fomin, F. et al. "On Two Techniques of Combining Branching and Treewidth". *Algorithmica* **54**.2 (2009), pp. 181–207.

[Fom10] Fomin, F. V. et al. "Algorithmic lower bounds for problems parameterized by clique-width". *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics. 2010, pp. 493–502.

[Fom14] Fomin, F. V. et al. "Tight bounds for parameterized complexity of cluster editing with a small number of clusters". *Journal of Computer and System Sciences* **80**.7 (2014), pp. 1430–1447.

[Gao10] Gao, X. et al. “A survey of graph edit distance”. *Pattern Analysis and applications* **13**.1 (2010), pp. 113–129.

[Gar79] Garey, M. & Johnson, D. *Computers and intractability: a guide to NP-completeness*. WH Freeman New York, 1979.

[Gas10] Gaspers, S. *Exponential Time Algorithms - Structures, Measures, and Bounds*. Verlag Dr. Mueller, 2010.

[Gél09] Gély, A. et al. “Enumeration aspects of maximal cliques and bicliques”. *Discrete Applied Mathematics* **157**.7 (2009), pp. 1447–1459.

[Goo18] Goodrich, T. D. et al. “Practical Graph Bipartization with Applications in Near-Term Quantum Computing”. *arXiv preprint arXiv:1805.01041* (2018).

[Gra04] Gramm, J. et al. “Automated generation of search tree algorithms for hard graph modification problems”. *Algorithmica* **39**.4 (2004), pp. 321–347.

[Gül04] Gülpinar, N. et al. “Extracting pure network submatrices in linear programs using signed graphs”. *Discrete Applied Mathematics* **137**.3 (2004), pp. 359–372.

[Hop73] Hopcroft, J. E. & Karp, R. M. “An nˆ5/2 algorithm for maximum matchings in bipartite graphs”. *SIAM Journal on Computing* **2**.4 (1973), pp. 225–231.

[Hor19] Horton, E. et al. *MI-bicliques: Version 2.0*. 10.5281/zenodo.3381532. 2019.

[Hüf05] Hüffner, F. “Algorithm engineering for optimal graph bipartization”. *International Workshop on Experimental and Efficient Algorithms*. Springer. 2005, pp. 240–252.

[Iwa14] Iwata, Y. et al. “Linear-time FPT algorithms via network flow”. *Proceedings of the twenty-fifth annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics. 2014, pp. 1749–1761.

[Jan13] Jansen, B. M. & Bodlaender, H. L. “Vertex cover kernelization revisited”. *Theory of Computing Systems* **53**.2 (2013), pp. 263–299.

[Kar05] Karakostas, G. “A better approximation ratio for the vertex cover problem”. *International Colloquium on Automata, Languages, and Programming*. Springer. 2005, pp. 1043–1050.

[Kar96] Karpinski, M. & Zelikovsky, A. *Approximating dense cases of covering problems*. Inst. für Informatik, 1996.

[Kay11] Kaytoue, M. et al. “Mining gene expression data with pattern structures in formal concept analysis”. *Information Sciences* **181**.10 (2011), pp. 1989–2001.

[KU08] Kaytoue-Uberall, M. et al. “Using formal concept analysis for the extraction of groups of co-expressed genes”. *Modelling, Computation and Optimization in Information Systems and Management Sciences*. Springer, 2008, pp. 439–449.

[Klo94] Kloks, T. *Treewidth, Computations and Approximations*. Vol. 842. LNCS. Springer, 1994.

[Klo19] Kloster, K. et al. “Mining Maximal Induced Bicliques using Odd Cycle Transversals”. *Proceedings of the 2019 SIAM International Conference on Data Mining*. SIAM. 2019, pp. 324–332.

[Kom18] Komusiewicz, C. “Tight running time lower bounds for vertex deletion problems”. *ACM Transactions on Computation Theory (TOCT)* **10**.2 (2018), p. 6.

[Kön31] König, D. “Graphen und matrizen”. *Mat. Fiz. Lapok* **38**.116-119 (1931), p. 60.

[Kra14] Kratsch, S. & Wahlström, M. “Compression via matroids: a randomized polynomial kernel for odd cycle transversal”. *ACM Transactions on Algorithms (TALG)* **10**.4 (2014), p. 20.

[Kri79] Krishnamoorthy, M. S. & Deo, N. “Node-deletion NP-complete problems”. *SIAM Journal on Computing* **8**.4 (1979), pp. 619–625.

[Kum99] Kumar, R. et al. “Trawling the Web for emerging cyber-communities”. *Computer networks* **31**.11-16 (1999), pp. 1481–1493.

[Kuz01] Kuznetsov, S. O. “On computing the size of a lattice and related decision problems”. *Order* **18**.4 (2001), pp. 313–321.

[Lav19] Lavallee, B. et al. *Structural Rounding: Version v1.0*. 10.5281/zenodo.3401541. 2019.

[Lew78] Lewis, J. M. “On the complexity of the maximum subgraph problem”. *Proceedings of the tenth annual ACM Symposium on Theory of Computing*. ACM. 1978, pp. 265–274.

[Lew80] Lewis, J. M. & Yannakakis, M. “The node-deletion problem for hereditary properties is NP-complete”. *Journal of Computer and System Sciences* **20**.2 (1980), pp. 219–230.

[Li07] Li, J. et al. “Maximal biclique subgraphs and closed pattern pairs of the adjacency matrix: A one-to-one correspondence and mining algorithms”. *IEEE Trans. Knowl. Data Eng.* **19**.12 (2007), pp. 1625–1637.

[Lok09] Lokshtanov, D. et al. “Simpler parameterized algorithm for OCT”. *International Workshop on Combinatorial Algorithms*. Springer. 2009, pp. 380–384.

[Lok11] Lokshtanov, D. et al. “Known algorithms on graphs of bounded treewidth are probably optimal”. *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms*. SIAM. 2011, pp. 777–789.

[Lok12] Lokshtanov, D. et al. “Subexponential parameterized odd cycle transversal on planar graphs”. *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2012)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2012.

[Mak04] Makino, K. & Uno, T. “New algorithms for enumerating all maximal cliques”. *Scandinavian Workshop on Algorithm Theory*. Springer. 2004, pp. 260–272.

[Mul87] Mulmuley, K. et al. “Matching is As Easy As Matrix Inversion”. *Combinatorica* **7**.1 (1987), pp. 105–113.

[Mus07] Mushlin, R. A. et al. “A graph-theoretical approach for pattern discovery in epidemiological research”. *IBM systems journal* **46**.1 (2007), pp. 135–149.

[Nas10] Nastos, J. & Gao, Y. “A novel branching strategy for parameterized graph modification problems”. *International Conference on Combinatorial Optimization and Applications*. Springer. 2010, pp. 332–346.

[Net] *network-corpus*. github.com/micrograviTas/network-corpus. 2018.

[Pan04] Panconesi, A. & Sozio, M. “Fast hare: A fast heuristic for single individual SNP haplotype reconstruction”. *International workshop on algorithms in bioinformatics*. Springer. 2004, pp. 266–277.

[Pap82] Papadimitrou, C. H. & Steiglitz, K. *Combinatorial optimization: algorithms and complexity*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[Pee03] Peeters, R. “The maximum edge biclique problem is NP-complete”. *Discrete Applied Mathematics* **131**.3 (2003), pp. 651–654.

[RI91] Richard III, C. et al. “A radiation hybrid map of the proximal long arm of human chromosome 11 containing the multiple endocrine neoplasia type 1 (MEN-1) and bcl-1 disease loci.” *Am. J. Hum. Genet.* **49**.6 (1991), pp. 1189–1196.

[Rob86] Robertson, N. & Seymour, P. “Graph minors. II. Algorithmic aspects of tree-width”. *J. Algorithms* **7**.3 (1986), pp. 309–322.

[Rob84] Robertson, N & Seymour, P. “Graph minors. III. Planar tree-width”. *Journal of Combinatorial Theory, Series B* **36**.1 (1984), pp. 49–64.

[San03] Sanderson, M. J. et al. “Obtaining maximal concatenated phylogenetic data sets from large sequence databases”. *Molecular biology and evolution* **20**.7 (2003), pp. 1036–1042.

[Sav82] Savage, C. “Depth-first search and the vertex cover problem”. *Information Processing Letters* **14**.5 (1982), pp. 233–235.

[Sch17] Schrock, J. et al. “Recall Performance for Content-Addressable Memory Using Adiabatic Quantum Optimization”. *Entropy* **19**.9 (2017), p. 500.

[Sha04] Shamir, R. et al. “Cluster graph modification problems”. *Discrete Applied Mathematics* **144**.1-2 (2004), pp. 173–182.

[Sha02] Sharan, R. “Graph modification problems and their applications to genomic research”. PhD thesis. Tel-Aviv University, 2002.

[Slo97] Slonim, D. et al. “Building human genome maps with radiation hybrids”. *J. Comp. Biol.* **4**.4 (1997), pp. 487–504.

[Sul16] Sullivan, B. D. & van der Poel, A. “A Fast Parameterized Algorithm for Co-Path Set”. *International Symposium on Parameterized and Exact Computation*. Springer. 2016.

[Sul19] Sullivan, B. D. et al. “Faster Biclique Mining in Near-Bipartite Graphs”. *Special Event on Analysis of Experimental Algorithms*. 2019.

[Tsu77] Tsukiyama, S. et al. “A new algorithm for generating all the maximal independent sets”. *SIAM Journal on Computing* **6**.3 (1977), pp. 505–517.

[Val79] Valiant, L. G. “The complexity of enumeration and reliability problems”. *SIAM Journal on Computing* **8**.3 (1979), pp. 410–421.

[Wer14] Wernicke, S. *On the algorithmic tractability of single nucleotide polymorphism (SNP) analysis and related problems*. diplom. de, 2014.

[Wil82] Wille, R. “Restructuring lattice theory: an approach based on hierarchies of concepts”. *Ordered sets*. Springer, 1982, pp. 445–470.

[Yan78] Yannakakis, M. “Node-and edge-deletion NP-complete problems”. *Proceedings of the tenth annual ACM Symposium on Theory of Computing*. ACM. 1978, pp. 253–264.

[Zha14a] Zhang, C. et al. “Radiation Hybrid Map Construction Problem Parameterized”. *Journal of Combinatorial Optimization* **27**.1 (2014), pp. 3–13.

[Zha16] Zhang, Y. et al. “K-anonymisation of social network by vertex and edge modification”. *International Journal of Embedded Systems* **8**.2-3 (2016), pp. 206–216.

[Zha14b] Zhang, Y. et al. “On finding bicliques in bipartite graphs: a novel algorithm and its application to the integration of diverse biological data types”. *BMC Bioinformatics* **15**.1 (2014), p. 110.