UNIVERSITY OF CALIFORNIA,
IRVINE

Parameterization and Concise Representation in Graph Algorithms: Leaf powers, Subgraphs with Hereditary Properties, and Activity-on-edge Minimization

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Elham Havvaei

2021

DEDICATION

To my loving family and friends

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank my advancement and dissertation committee members, Professor Michael T. Goodrich, Professor Sandy Irani, Professor Vijay V. Vazirani, Professor Dan Hirschberg and Matthew D. Foreman, who were generous with their expertise and precious time. A special thanks to Professor David Eppstein, my committee chairman for advising and supporting me through my PhD studies.

I would like to thank all my collaborators, Professor David Eppstein, Daniel Frishberg and Siddharth Gupta for their invaluable and pleasant partnership.

I would like to thank my fellow graduate students in the Center for Algorithms and Theory of Computation for all the enlightening conversations and fun times we have had together.

I would like to thank the Donald Bren School of Information and Computer Science, at University of California, Irvine for their funding support. Additionally, much of the research in this dissertation was supported in part by NSF Grants CCF-1618301 and CCF-1616248.

VITA

Elham Havvaei

EDUCATION

Doctor of Philosophy in Computer Science **2021**
University of California, Irvine *Irvine, CA*

Master of Science in Computer Science **2016**
University of Central Florida *Orlando, FL*

Bachelor of Science in Computational Sciences **2012**
Amirkabir University of Technology *Tehran, Iran*

RESEARCH EXPERIENCE

Graduate Research Assistant **2016–2021**
University of California, Irvine *Irvine, CA*

Graduate Research Assistant **2013–2016**
University of Central Florida *Orlando, FL*

TEACHING EXPERIENCE

Teaching Assistant **2020–2021**
University of California, Irvine *Irvine, CA*

Teaching Assistant **2016–2018**
University of California, Irvine *Irvine, CA*

Teaching Assistant **2013–2016**
University of Central Florida *Orlando, FL*

PUBLICATIONS

- Elham Havvaei, David Eppstein, Siharth Gupta. "*Parameterized Complexity of Finding Subgraphs with Hereditary Properties on Hereditary Graph Classes*", under review [56].
- Elham Havvaei, David Eppstein, Daniel Frishberg. "*Simplifying Activity-on-Edge Graphs*", 17th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2020) [53].

- Elham Havvaei, David Eppstein. "*Parameterized Leaf Power Recognition via Embedding into Graph Products*", 13th International Symposium on Parameterized and Exact Computation (2019) [57].

- Elham Havvaei, Narsingh Deo. "*A Game-Theoretic Approach for Detection of Overlapping Communities in Dynamic Complex Networks.*", International Journal of Mathematical and Computational Methods 1 (2016): 313-324 [75].

ABSTRACT OF THE DISSERTATION

Parameterization and Concise Representation in Graph Algorithms: Leaf powers, Subgraphs with Hereditary Properties, and Activity-on-edge Minimization

By

Elham Havvaei

Doctor of Philosophy in Computer Science

University of California, Irvine, 2021

Professor David Eppstein, Chair

Parameterized complexity provides an important framework to deal with hard problems by restricting some problem parameter to be a fixed constants. Problems which are categorized as fixed-parameter tractable with respect to some parameters are problems that can be solved in polynomial time, if such parameters are bounded by a fixed value.

In this dissertation, in Chapter 2, we first study the problem of recognizing k-leaf powers and representing k-leaf roots. A graph G is a k-leaf power of a tree if its vertices correspond to leaves of the tree and a pair of leaves have distance at most k if and only if the corresponding vertices in G are adjacent. Then, the tree is a k-leaf root of G. A graph is a k-leaf power if it has at least one k-leaf root. We show recognizing k-leaf powers parameterized by k and the degeneracy of the input graph is fixed parameter tractable. This is the first result in the literature studying this problem in the paradigm of parameterized complexity and providing a polynomial-time algorithm working on multiple values of k.

Following this line of work, in Chapter 3, we study the parameterized complexity of the problem of finding induced subgraphs with hereditary properties under the condition that the input graph belongs to a hereditary graph class, as well. In this work, we provide a framework that settles the parameterized complexity of various graph classes.

To show the importance of graph representation, we emphasize that our proposed technique for recognition of k-leaf powers for graphs of bounded degeneracy heavily relies on our representation of k-leaf roots as a subgraph of the graph product of the input graph and a cycle graph of size k. Additionally, in Chapter 4, we study the problem of simplifying activity-on-edge graphs, which provides an insight on how graph representation can further help data analysis such as enabling a better understanding of the flow of project schedules. In an activity-on-edge graph, vertices represent project milestones and edges represent the tasks/activities of the project. We simplify such representations of project schedules to enhance visualization of an abstract timeline of the potential critical paths of the project by optimally minimizing the number of vertices while maintaining the reachability relations among tasks.

Chapter 1

Introduction

Graphs and graph-theoretical techniques have been widely used in various fields, such as biology [96, 86], social sciences [17, 74] and road networks [54, 55, 58]. Additionally, the literature is rich with problems that have been proved to be NP-hard, which and are therefore presumed to be unsolvable in polynomial time. Given the complexity of these problems, researchers often take measures such as designing approximations, heuristics, or parameterized algorithms to deal with these problems. In Chapter 2 and 3, we study the important problems of recognizing k-leaf powers and finding subgraphs with hereditary properties, from the point of view of parameterized complexity.

1.1 Parameterized Complexity

The use of parameterized complexity theory, developed by Downey and Fellows [47], has grown considerably as a refined way of dealing with hard problems. Problems in which some parameters are fixed or bounded are called parameterized problems. Measuring the complexity of a problem as a function of the parameters provides a finer classification of inherently hard problems than the classic NP-hard categorization. A problem is *fixed-parameter tractable* (FPT) with respect to a parameter x of the input if the problem can be decided in time $f(x)n^{O(1)}$ where f is a computable function depending only on parameter x, n is the size of the input, and the exponent of n is a constant, independent of x. Downey and Fellows have further introduced the complexity classes of parameterized problems known as the W hierarchy. It is presumed that problems complete for W[t] for $t \geq 1$ are fixed-parameter intractable. For more information on parameterized complexity, we refer the reader to [42, 49].

A natural parameter for the study of the parameterized complexity of a problem is the size of the solution. Problems such as k-vertex cover [25, 33, 32] and k-feedback vertex set [34] have been parameterized by the size of the desired output set, k. In Chapter 3, we use the size of the solution to parameterize the underlying problem.

Another useful parameter which targets sparse graphs is *degeneracy* [92] as it implies that every graph of size n and degeneracy d has at most $(n-1)d$ edges. A graph is k-degenerate if every induced subgraph has a vertex of degree at most k. Degeneracy of a graph is defined as the smallest value of k for which the graph is k-degenerate. Equivalently, degeneracy may be defined as the smallest d for which an ordering of vertices of the graph exists in which each vertex has d earlier neighbors according to that ordering. Various problems in the literature are parameterized by the degeneracy of the input graph [60, 6, 28].

Treewidth is another graph sparsity parameter which was initially introduced by Bertelé, Brioschi [13] and Halin [72] under the name *S-function* and later rediscovered by Robertson and Seymour [103]. A tree decomposition of graph $G(V, E)$ consists of a tree T such that:

- Each vertex $X_i \in T$ (called a bag) is a subset of vertices of G
- For each edge $e(u, v) \in E$, there exists a bag X_i in T where both $u, v \in X_i$; and
- For each vertex $v \in V$, the bags containing v induce a nonempty subtree of T.

The width of a tree decomposition is the size of its largest bag, minus one. The treewidth of a graph is the minimum width over all tree decompositions of the graph. Many well-known graph classes such as series-parallel graphs and outerplanar graphs have bounded treewidth. Additionally, there exist many NP-complete problems that have linear-time algorithms on graphs of bounded treewidth using dynamic programming techniques [15, 14].

Another graph parameter is the *clique-width*, introduced by Courcelle, Engelfriet and Rozenberg [40]. The clique-width of a graph G is the minimum number of labels needed to construct G using the following four graph operations: creation of a new vertex with a label, vertex disjoint union, connecting vertices with specified labels and relabeling vertices. Graphs of bounded clique-width form a more general graph class than the class of graphs with bounded treewidth.

1.2 Recognizing Sparse k-leaf Powers

Graph representation has been widely used in biology to model and further solve important problems. Relevantly to our work, graph representation plays an important role in similarity and evolutionary analysis of species. Given the similarity data among species, one can represent the data as a graph in which the vertices are the species and a pair of vertices is adjacent if and only if they are similar enough. Analyzing these graphs has an important application in phylogeny for the problem of building a phylogenetic tree out of a similarity graph. Such analysis provides a rich understanding of how biological entities evolve and act over time. A phylogenetic tree represents the evolutionary interrelationships among species and has a fundamental role in illustration of the origin and the evolution of species or entities [105, 100]. The leaves of the phylogenetic tree represent species under study, and for a pair of leaves, their parent represent their most recent common ancestor. The internal vertices of the tree can be seen as the species that are long-extinct.

Motivated by the search for the phylogenetic trees, Nishimura, Ragde and Thilikos introduced the concept of leaf power graphs [99]. Formally speaking, a k-leaf power of a tree T is a graph G whose vertices are the leaves of T, where a pair of vertices in G is adjacent if and only if the corresponding leaves in T are within distance k of each other. Then, G is a k-leaf power and T is a k-leaf root of G. A k-leaf power graph and its k-leaf root can represent the similarity graph among species and its phylogenetic tree, respectively, with similarity threshold k.

In Chapter 2, we investigate the problem of recognizing sparse k-leaf powers from the lens of parameterized complexity theory. Our contribution to this problem is to provide two different algorithms to recognize k-leaf powers in linear time, parameterized by k and the degeneracy of the input graph. For our first algorithm, we express the property of being a k-leaf power in monadic second-order logic and, using Courcelle's theorem [40], we prove that recognizing k-leaf powers is fixed-parameter tractable when parameterized by k and the degeneracy of the input graph. Our second algorithm takes a dynamic programming approach to provide a better dependence on the parameters. The complexity of the dynamic programming algorithm is $O\big(n(wk)^{O(w)}\big)$ where w is the treewidth of the input graph. For $k \geq 7$, recognition of k-leaf powers for general graphs is still an open problem. However, our decision to explore this problem from the point of view of parameterized complexity delivers substantial progress in recognition of sparse k-leaf powers.

1.3 Finding Subgraphs with Hereditary Properties

Following the work on graph parameterization, in Chapter 3, we investigate parameterized complexity of another inherently hard problem, which is to determine, in an input graph G belonging to a hereditary graph class, whether there exist k vertices of G that induce a subgraph satisfying a particular hereditary property. This problem for general input graphs, not necessarily belonging to a hereditary property class, has been already proved to be NP-complete [91].

Our contribution to this problem is to extend the parameterized complexity results proved by Khot and Raman [84]. They found that if the underlying property includes all trivial graphs but not all complete graphs, or vice versa, then the problem is W[1]-complete and it is fixed-parameter tractable, otherwise. Our contribution aims to extend parameterized complexity of the problem in cases for which the problem is W[1]-complete for general input graphs, by integrating the constraint that the input graph also belongs to a hereditary graph class. Our motivation to extend these results is based on our observation that there exist problems that are NP-complete or intractable on general graphs or some graph classes but become tractable when the input graph is restricted to some graph classes. For instance, k-clique is W[1]-complete for general graphs [48], and NP-complete for multiple-interval graphs, [26], but it is fixed-parameter tractable for multiple-interval graphs [63].

To deal with this problem, we partition hereditary properties into four classes named AA, AS, SA, SS. As the first letter, "A" (respectively, "S") indicates that the property includes (excludes) all (some) complete graphs. As the second letter, "A" (respectively, "S") indicates that the property includes (excludes) all (some) independent sets. The focus of our work is to solve the problem for the cases that the hereditary property belongs to either AS or SA. The other cases are already known to be fixed-parameter tractable [84], regardless of the properties of the input graph.

This categorization enables us to use Ramsey's theorem to prove our tractablity results. Informally speaking, Ramsey's theorem indicates that if a graph is large enough, it has either a large clique or a large independent set. Additionally, for our hardness results we design polynomial-time parameterized reductions to the k-independent set problem, which is known to be NP-complete or W[1]-complete for various graph classes. Our technique settles the parameterized complexity of the problem for numerous important graph classes and removes the need for long and tedious hardness proofs. For an input graph G and hereditary properties Π_G and Π, let $G \in \Pi_G$ and our desired subgraph belong to Π. Our results show, when both Π and Π_G are the properties of being planar, bipartite, triangle-free, or co-bipartite, then we settle the parameterized complexity of the problem as FPT. Additionally, if Π_G is the property of being a unit-disk graph, C_4-free, or $K_{1,4}$-free, and Π is the property of being either planar or bipartite, then we show the problem is $\mathsf{W}[1]$-complete.

1.4 Simplifying Activity-on-Edge Graphs

The well-structured and concise representation of graphs or a subgraphs give insight into the structure and recognition of important graph classes and further help toward the betterment of graph visualization. In Chapter 2, we deeply studied the structure and characteristics of the k-leaf root as an embedded subgraph of the graph product of the input graph and a k-vertex cycle. Discovering and understanding this representation of the k-leaf root as a subgraph of such a graph product was the key insight that paved our way toward the recognition of k-leaf powers and reconstruction of k-leaf roots in both our proposed algorithms. Similarly, in Chapter 4, we follow this line of work and use concise graph representation to improve visualization of project schedules by simplifying activity-on-edge graphs. An activity-on-edge graph (AOE) has vertices that represent project milestones and edges that represent activities. AOEs are very similar to Activity-on-Node graphs (AONs). In an AON graph, project schedule information is presented differently, with vertices specifying activities and edges representing the logical progression of the dependencies among activities. AONs have several applications such as modeling minimal and maximal time lags among activities [98].

Although AON networks seem to be more natural to represent project schedules, they are unsuitable for visually representing timelines of potential critical paths where the time length of tasks/activities is yet unknown. Therefore, to resolve this issue, we choose to employ AOEs in order to represent the project schedules and the possible critical paths. Given an activity-on-edge graph, our focus is on finding an AOE with the minimum possible number of vertices that has the same critical paths as the input graph. We provide a polynomial-time algorithm to transform the input AOE into an optimal one and prove correctness and optimality.

Chapter 2

Parameterized Leaf Power Recognition via Embedding into Graph Products [1]

2.1 Introduction

As stated in Chapter 1, leaf powers are a class of graphs that were introduced in 2002 by Nishimura, Ragde and Thilikos [99], extending the notion of graph powers. For a graph G, the kth power graph G^k has the same set of vertices as G but a different notion of adjacency: two vertices are adjacent in G^k if there is a path of at most k edges between them in G. Determining whether a graph is a kth power of another graph is known to be NP-complete, for $k \geq 2$ [89]. However deciding whether a graph G is the second power of a graph H is decidable in polynomial time when H belongs to various graph classes such as bipartite graphs [88], block graphs [90], cactus graphs [68] and cactus block graphs [51]. Besides, it is

[1]The material in this chapter is included with permission from Springer [57].

possible to decide in linear time if a graph is the power of a tree [30]. The leaf powers are defined in the same way from trees, but only including the leaves of the trees as vertices. The kth leaf power of a tree T has the leaves of T as its vertices, with two vertices adjacent in the leaf power if there is a path of at most k edges between them in T. A given graph G is a k-leaf-power graph when there exists a tree T for which G is the kth leaf power. In this case, T is a *k-leaf root* of G. In general, the k-leaf root may have vertices and edges that are not part of the input graph. For example, Figure 2.1 shows a 3-leaf power alongside one of its 3-leaf roots. Nishimura, Ragde and Thilikos, further, derived the first polynomial-time algorithms to recognize k-leaf powers for $k = 3$ and $k = 4$ [99].

One application of recognizing leaf powers arises as a formalization of a problem in computational biology, the reconstruction of evolutionary history and evolutionary trees from information about the similarity between species [35, 64, 2]. In this problem, the common ancestry of different species can be represented by an evolutionary or phylogenetic tree, in which each vertex represents a species and each edge represents a direct ancestry relation between two species. We only have full access to living species, the species at the leaves of the tree; the other species in the tree are typically long-extinct, and may be represented physically only through fossils or not at all. If we suppose that we can infer, from observations of living species, which ones are close together (within some number k of steps in this tree) and which others are not, then we could use an algorithm for leaf power recognition to infer a phylogenetic tree consistent with this data.

2.1.1 New Results

In this chapter, presenting two different algorithms, we prove that the k-leaf powers of degeneracy d can be recognized in time that is fixed-parameter tractable when parameterized by k and d. Here, the degeneracy of a graph is the maximum, over its subgraphs, of the

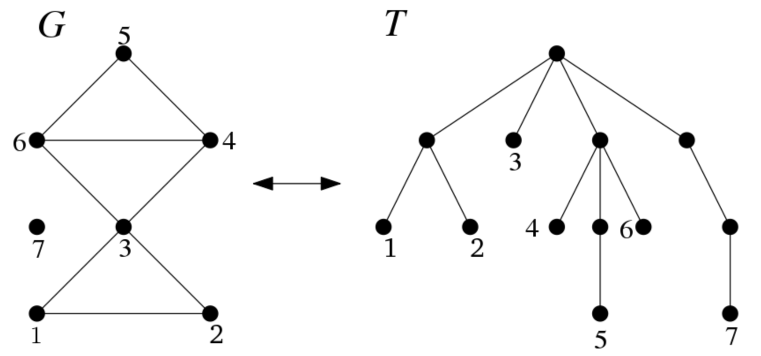

Figure 2.1: A 3-leaf power graph G and one of its 3-leaf roots T.

minimum degree of any subgraph.

Our first algorithm makes ample use of Courcelle's theorem [40] while the second employs a dynamic programming method to provide a time complexity with a better dependence on the parameters. Although the second algorithm is more efficient, we retain the description of the first algorithm as it was the source of our inspiration to devise a more practical method to prove the fixed-parameter tractability of k-leaf powers, and as we feel that our technique of using graph products (which we use in both algorithms) can have broader applications.

Both algorithms have running time polynomial (in fact linear) in the size of the input graph, multiplied by a factor that depends non-polynomially on k and d. We also apply the same methods to a more general problem in which each edge of the input graph is labeled by a range of distances, constraining the corresponding pair of leaves in the leaf root to have a distance in that range.

Relevantly for our work, Golumbic and Rotics showed that unit interval graphs are of unbounded clique-width [69]. A graph is an interval graph if and only if all its vertices can be mapped into intervals on a straight line such that two vertices are adjacent when the corresponding intervals intersect each other. In the unit interval graphs, each interval has a unit length. As shown by Brandstädt and Hundt, unit interval graphs belong to the class of leaf powers, which implies that leaf powers also have unbounded clique-width [19, 20].

However, it is known that the k-leaf powers have bounded clique-width when k is bounded [71]. A wide class of graph problems (those expressible in a version of monadic second order logic quantifying over only vertex sets, MSO_1) can be solved in fixed-parameter time for graphs of bounded clique-width, via Courcelle's theorem. However we have been unable to express the recognition of leaf powers in MSO_1. Instead, our algorithm uses a more powerful version of monadic second order logic allowing quantification over edge sets, MSO_2. As we prove later in Section 2.5, leaf powers with bounded degeneracy have bounded treewidth, allowing us to apply a form of Courcelle's theorem for MSO_2 for graphs of bounded treewidth.

However, there is an additional complication that makes it tricky to apply these methods to leaf power recognition. As stated in Section 2.1, the tree that we wish to find, for which our given input graph is a leaf power, will in general include vertices and edges that are not part of the input, but MSO_2 can only quantify over subsets of the existing vertices and edges of a graph, not over sets of vertices and edges that are not subsets of the input. To work around this problem, we apply Courcelle's theorem not to the given graph G itself, but to a *graph product* $G \boxtimes C_k$ where C_k is a k-vertex cycle graph. In Section 2.3, we prove that a leaf root (the tree for which G is a leaf power, if there is one) can be embedded as a subgraph of this product, that it can be recognized by an MSO_2 formula applied to this product, and that this product has bounded treewidth whenever G is a k-leaf power of bounded degeneracy. In this way we can recognize G as a leaf power, not by applying Courcelle's theorem to G, but by applying it to the graph product.

Thus, our algorithm combines the following ingredients:

- Our embedding of the k-leaf root as a subgraph of the graph product $G \boxtimes C_k$.
- Our logical representation of k-leaf roots as subgraphs of graph products.
- Courcelle's theorem, which provides general-purpose algorithms for testing MSO_2 formulas on graphs of bounded treewidth.

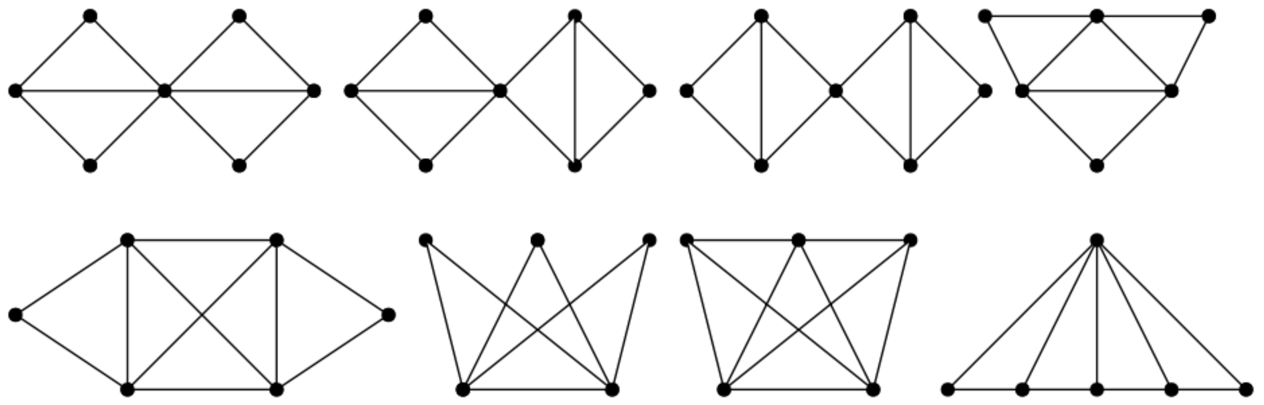
Figure 2.2: A graph is a 4-leaf power if and only if it is chordal and does not contain any of the graphs above as a subgraph.

- The fact that leaf powers of bounded degeneracy also have bounded treewidth.
- The fact that, by taking a product with a graph of bounded size, we preserve the bounded treewidth of the product.

Our algorithm runs in fixed-parameter tractable time when parameterized by k and the degeneracy d of the given input graph. In particular, it runs in linear time when k and d are both constant.

Our results provide the first known efficient algorithms for recognizing k-leaf powers for $k \geq 7$, for graphs of bounded degeneracy. Our method of embedding the k-leaf roots into graph products considerably simplifies our task of designing a logical formula for recognizing k-leaf powers.More generally, such embedding appears likely to apply to other graph problems involving network design (the addition of edges to an existing graph, rather than the identification of a special subgraph of the input). Later in Section 2.7, we also profit from the same embedding into a product as a key step in our dynamic programming algorithm to decide whether a graph is a k-leaf power.

2.1.2 Related Work

Polynomial-time algorithms are known for recognizing k-leaf powers for $k \leq 6$.

- A graph is a 2-leaf power if it is a disjoint union of cliques, so this class of graphs is trivial to recognize.
- There exist various ways to characterize 3-leaf powers [99, 21, 45, 102], some of which lead to efficient algorithms. For instance, one way to determine if a graph is a 3-leaf power is to check whether it is bull-, dart- and gem-free and chordal [45]. The chordal graphs have a known recognition algorithm, and testing for the existence of any of the other forbidden induced subgraphs is polynomial, because they all have bounded size.
- Similarly, there are various known ways to characterize 4-leaf powers [99, 102, 46, 23]. One is that a graph is a 4-leaf power if and only if it is chordal and does not contain any of the graphs depicted in Figure 2.2 as induced subgraphs [102]. Again, this leads to a polynomial-time recognition algorithm, because all of these graphs have bounded size.
- k-leaf powers can be recognized in polynomial time if the $(k-2)$- Steiner root problem can be solved in polynomial time. Chang and Ko, in 2007, provided a linear-time recognition algorithm for 3-Steiner root problem [29]. This implies that 5-leaf powers can be recognized in linear time. Besides, Brandstädt, Le and Rautenbach provided a forbidden induced subgraph characterization for the distance-hereditary 5-leaf powers [22].
- Ducoffe has recently extended result of Chang and Ko [29] and provided a polynomial-time recognition algorithm of 4-Steiner powers [50] which as stated, it leads to a polynomial-time recognition of 6-leaf powers.

Polynomial-time structural characterization of k-leaf powers for $k \geq 7$ is still an open problem.

Throughout the literature, there exist many structural characterizations of leaf powers which provide potentially useful insight into this class of graphs. It is known, for instance, that all leaf powers are strongly chordal, but the converse is not always true. Further, Kennedy, Lin and Yan showed that strictly chordal graphs are always k-leaf powers for $k \geq 4$; these are the chordal graphs that are also, dart- and gem-free. They provided a linear-time algorithm to construct k-leaf roots of strictly chordal graphs [83].

For all $k \geq 2$, every k-leaf power is also a $(k+2)$-leaf power. A $(k+2)$-leaf root of any k-leaf-power can be obtained from its k-leaf root, by subdividing all edges incident to leaves. However, the problems of recognizing k-leaf powers for different values of k do not collapse: for all $k \geq 4$, there exists a k-leaf power which is not a $(k+1)$-leaf power [24].

2.1.3 Organization

This chapter is organized as follows. We begin in Section 2.2 with some preliminary definitions and a survey of the relevant background material for our results. In Section 2.3 we describe how to embed leaf roots into graph products , a construction used in both of our algorithms. We provide a logical formulation of the leaf power recognition problem in Section 2.4, and in Section 2.5 we use this formulation for our first algorithm for the problem. We generalize the problem to leaf powers with restricted distance ranges on each input graph edge in Section 2.6. Our dynamic programming algorithm for leaf powers is presented in Section 2.7. We conclude with some general observations in Section 2.8.

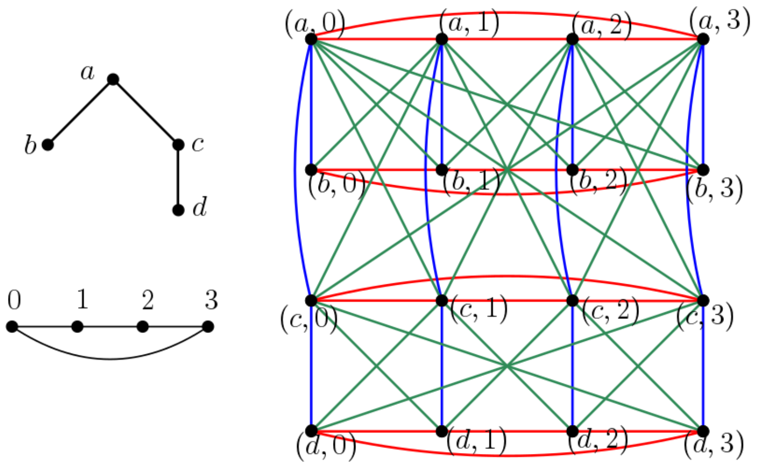

Figure 2.3: The graph on the right is the strong product of a four-vertex path graph (top left) and a four-vertex cycle graph (bottom left). The colors indicate the partition of the edges into vertical, horizontal, and diagonal subsets.

2.2 Preliminaries

2.2.1 Definitions

Throughout this chapter, we let $G(V, E)$ denote a simple undirected graph (typically, the input to the leaf power recognition problem). If u and v are two vertices in V that are adjacent in G, we let $e(u, v)$ denote the edge connecting them.

The strong product of graphs G_1 and G_2, denoted as $G_1 \boxtimes G_2$, is a graph whose vertices are ordered pairs of a vertex from G_1 and a vertex from G_2. In it, two distinct vertices (u_1, u_2) and (v_1, v_2) are adjacent if and only if for all $i \in \{1, 2\}$, $u_i = v_i$ or u_i and v_i are adjacent in G_i. Figure 2.3 shows an example, the strong product of a four-vertex path graph with a four-vertex cycle graph. When we construct a strong product, we will classify the edges of the product into three subsets:

- We call an edge from (u_1, u_2) to (v_1, v_2) a *vertical edge* if $u_2 = v_2$. The edges of this type form $|V(G_2)|$ disjoint copies of G_1 as subgraphs of the product.

- We call an edge from (u_1, u_2) to (v_1, v_2) a *horizontal edge* if $u_1 = v_1$. The edges of this type form $|V(G_1)|$ disjoint copies of G_2 as subgraphs of the product.

- We call the remaining edges, for which $u_1 \neq v_1$ and $u_2 \neq v_2$, *diagonal edges.* The subgraph composed of the diagonal edges forms a different kind of graph product, the *tensor product* $G_1 \times G_2$.

We may think of these three edge sets as forming an (improper) edge coloring of the graph product. In Figure 2.3 these edge sets are colored blue, red and green, respectively.

2.2.2 Courcelle's Theorem

By considering graphs as logical structures, their properties can be expressed in first-order and second-order logic. In first-order logic, graph properties are expressed as logical formulas wherein the variables range over vertices and the predicates include equality and adjacency relations. Second-order logic is an extension of first-order logic with the power to quantify over relations. Particularly, many natural graph properties can be described in monadic second-order logic, which is a restriction of second-order logic in which only unary relations (sets of vertices or edges) are allowed [39].

There exist two variations of monadic second-order logic: MSO_1 and MSO_2. In MSO_1, quantification is allowed only over sets of vertices, while MSO_2 allows quantification over both sets of vertices and sets of edges. MSO_2 is strictly more expressive; there are some properties, such as Hamiltonicity [38], which are expressible in MSO_2 but not in MSO_1. A graph property is MSO_2-*expressible* if there exists an MSO_2 formula to express it, in which case the corresponding class of graphs becomes MSO_2-*definable.*

The algorithmic connection between treewidth and monadic second-order logic is given by Courcelle's theorem, according to which every property definable in monadic second-order logic can be tested in linear time on graphs of bounded treewidth [37]. Later, Courcelle, Makowsky and Rotics extended this theorem to the class of graphs with bounded clique-width when the underlying property is MSO_1-definable [41]. In our application of Courcelle's theorem, we will use an MSO_2 formula with a free variable HORIZONTAL, an edge set, which we will use to pass to the formula certain information about the structural decomposition of the graph it is operating on. This extension of Courcelle's theorem to formulas with a constant number of additional free variables, whose values are assigned through some extra-logical process prior to applying the theorem, is non-problematic and standard. Courcelle's theorem is the foundation of many fixed-parameter tractable algorithms [10, 70, 59, 79], as it proves that properties expressible in MSO_1 or MSO_2 are fixed-parameter tractable with respect to the clique-width or treewidth (respectively) of the input graph.

However, even in MSO_2, it is only possible to quantify over subsets of vertices and edges that belong to the graph to which the logical formula is applied. Much of the difficulty of the leaf power problem rests in this restriction. If we could quantify over edges and vertices that were not already present, we could construct a formula that asserts the existence of sets of vertices and edges forming a leaf root of a given graph, and then add clauses to the formula that ensure that the quantified sets describe a valid leaf root. However, we are not allowed such quantification, because in general the leaf root has vertices and edges that do not belong to our input graph. To apply Courcelle's theorem to leaf power recognition, we must instead find a way to express the property of being a leaf power using only quantification over subsets of vertices and edges of the graph to which we apply the theorem. For this reason, the problem of leaf power recognition forms an important test case for the ability to express graph problems in MSO logic.

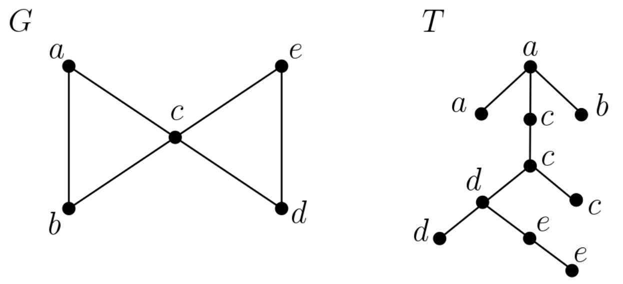

Figure 2.4: A 4-leaf power graph G (left), and one of its leaf roots T (right). Each leaf of T is labeled by the vertex of G that it represents, and each internal node of T is labeled by its closest leaf node. When there are ties at a node (as for instance at the root of T) the choice of label is made arbitrarily among the closest leaf nodes whose labels appear among the children of the node.

2.3 Embedding Leaf Roots into Graph Products

In this section, we show that every k-leaf power has a k-leaf root that can be embedded in the graph product $G \boxtimes C_k$. Let G be a k-leaf power graph, and T be a k-leaf root of G. If G is not connected, we can handle each of its connected components independently; in this way, we can assume from now on, without loss of generality that G is a connected graph with at least three vertices, and that T is a leaf root chosen arbitrarily among the possible k-leaf roots of T. It follows from these assumptions that T is a tree, because every edge in G must be represented by a path in T. Because T has at least three leaves, it has at least one interior node; we choose one of these nodes arbitrarily to be the root of T. Additionally, every vertex or edge of T participates in a path of length at most k between two leaves, representing an edge of G. For, if some vertices and edge do not participate in these paths, removing all non-participating vertices and edges from T would produce a smaller leaf root, without creating any new leaves. But this removal would disconnect pairs of leaves on the opposite sides of any removed edge, contradicting the assumption that G is connected.

As the first step of the embedding, we provide a subroutine that takes as input, a graph and a k-leaf root of the form, mentioned above and embeds it in $G \boxtimes C_k$ as a subgraph. Our leaf-power recognition algorithm does not employ this subroutine, as it does not have access

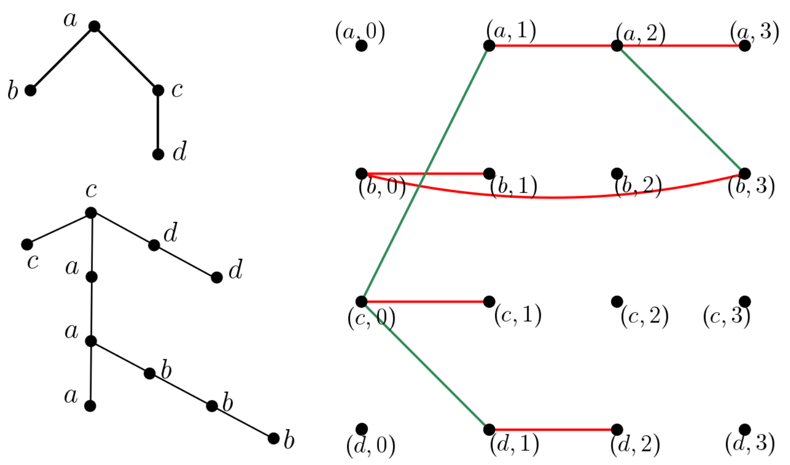

Figure 2.5: The graph on the bottom left is a 4-leaf root T of graph G (top left). T can be embedded in the strong product $G \boxtimes C_4$ (right), by mapping each vertex u of T to the pair (v, i) where v is the label of u and i is the depth of u (modulo k).

to the k-leaf root. This subroutine solely fulfills the purpose of proving that the k-leaf root of this form can be embedded in the graph product. For that, we label the vertices of T with the names of vertices in G. Each vertex of T will get a label in this way; some labels will be used more than once. In particular, we label each leaf of T by the vertex of G represented by that leaf. Then, as shown in Figure 2.4, we give each non-leaf node of T the same label as its closest leaf. If there are two or more closest leaves, we choose one arbitrarily among the labels already applied to the children of the given interior node. In this way, when the same label appears more than once, the tree nodes having that label form a connected path in T.

As we now show, these labels, together with the depths of the nodes modulo k, can be used to embed the k-leaf root T into the strong product $G \boxtimes C_k$, where C_k denotes a k-vertex cycle graph.

Lemma 2.1. *If G is a connected k-leaf power graph on three or more vertices, and T is any k-leaf root of G, then T can be embedded as a subtree of the strong product $G \boxtimes C_k$. Additionally, the embedding can be chosen in such a way that each horizontal cycle in the strong product (the product of a vertex v of G with C_k) contains exactly one leaf of the embedded copy of T, the leaf representing v.*

Proof. We map a vertex u of T to the pair (v, i) where v is the label assigned to u (the name of a vertex in G) and i is the depth of u (its distance from the root of T), taken modulo k. This pair is one of the vertices of the strong product, so we have mapped vertices of T into vertices of the strong product. An example of such embedding can be seen in Figure 2.5. Because G is assumed to be connected, each node of T participates in at least one path of length at most k between two leaves of T, representing an adjacency of G; it follows that the label for each node of T is at most $k - 1$ steps away from the node, and that each path of same-labeled nodes in T has length at most $k - 1$. As a consequence, when we take depths modulo k, none of these paths can wrap around the cycle and cover the same vertex of the graph product more than once. That is, our mapping from T to $G \boxtimes C_k$ is one-to-one. Because each leaf of T is labeled with the vertex of G that it represents, this mapping has the property described in the lemma, that each horizontal cycle in the strong product contains exactly one leaf of the embedded copy of T, the leaf representing the vertex whose product with C_k forms that particular horizontal cycle.

We must also show that this mapping from T to $G \boxtimes C_k$ maps each pair of vertices that are adjacent in T into a pair of vertices that are adjacent in $G \boxtimes C_k$. Recall that adjacency in $G \boxtimes C_k$ is the conjunction of two conditions: two vertices in the product are adjacent if their first coordinates are equal or adjacent in G and their second coordinates are equal or adjacent in C_k. Because every two adjacent vertices in T have depths that differ by one, the second coordinates of their images in the product will always be adjacent in C_k. It remains to show that, when two vertices are adjacent in T, their images in the product have first coordinates that are equal or adjacent in G. That is, the labels of the two adjacent vertices in T should be equal or adjacent.

Rephrasing what we still need to show, it is the following: whenever two adjacent vertices in T have different labels, those labels represent adjacent vertices in G.

To see that this is true, consider two adjacent vertices u_1 and its parent u_2 in T, labeled by two different vertices v_1 and v_2 in G. As we already stated at the start of this section, the assumption of the lemma that G is connected implies that edge u_1u_2 in T participates in at least one path P of length at most k between two leaves, corresponding to an adjacency in G. But because v_1 and v_2 are represented by the closest leaves to u_1 and u_2 (respectively) the length of the path in T between the leaves representing v_1 and v_2 must be at most equal to the length of P. Therefore, there is a path of length at most k between the leaves representing v_1 and v_2, so v_1 and v_2 are adjacent in the k-leaf power G, as required. □

Based on this embedding, we can prove the following characterization of leaf powers, which we will use in our application of Courcelle's theorem to the problem. It is important, for this characterization, that we express everything intrinsically in terms of the properties of the graph product $G \boxtimes C_k$, its edge coloring, and its subgraphs, without reference to the given graph G.

Lemma 2.2. *A given connected graph G on three or more vertices is a k-leaf power if and only if the product $G \boxtimes C_k$ has a subgraph T with the following properties:*

1. *T is 1-degenerate (i.e., a forest).*
2. *Every vertex of $G \boxtimes C_k$ is connected by horizontal edges of the product to exactly one leaf of T.*
3. *Two vertices of $G \boxtimes C_k$ are the endpoints of a non-horizontal edge of the product if and only if the corresponding leaves of T (given according to Property 2) are the distinct endpoints of a path of length at most k in T.*

Proof. A subgraph obeying these properties is a forest (Property 1), whose leaves can be placed into one-to-one correspondence with the vertices of G (Property 2, using the fact that the horizontal cycles of the product correspond one-to-one with vertices of G). It has a path of length at most k between two leaves if and only if the corresponding vertices of G are adjacent (Property 3). So if it exists, it is a k-leaf root of G and G is a k-leaf power.

In the other direction, if G is a connected k-leaf power, let T be a k-leaf root of G. Then, according to Lemma 2.1, T can be embedded as a subtree of $G \boxtimes C_k$ (Property 1), with exactly one leaf for each horizontal cycle (Property 2), that forms a k-leaf root of G (Property 3). So when G is a k-leaf power, a subgraph T obeying the properties of the lemma exists. $\square$

2.4 Logical Expression

In this section, we describe how to express the components of Lemma 2.2, our characterization of the products $G \boxtimes C_k$ that contain a k-leaf root of G, in monadic second-order logic. Our logical formula will involve a free variable *horizontal*, the subset of edges of the given graph (assumed to be of the form $G \boxtimes C_k$) that are horizontal in the product (that is, edges that connect two copies of the same vertex in G). We will also assume that V and E refer to the vertices and edges of the graph $G \boxtimes C_k$. In our logical formulas, we will express the type of each quantified variable (whether it is a vertex, edge, set of vertices, or set of edges) by annotating its quantifier with a membership or subset relation. For instance, "$\forall x \in V : \ldots$" quantifies x as a vertex variable. We will express the incidence predicate between an edge e and a vertex v (true if v is an endpoint of e, false otherwise) by $e \multimap v$. Because our formulas will also use equality as a predicate, we will express the equality between names of formulas and their explicit logical formulation using a different symbol, $\equiv$. In our formulas, predicates (equality, incidence, and adjacence) will be considered to bind more tightly than logical connectives, allowing us to omit parentheses in many cases.

A subgraph of the given graph may be represented by its set S of edges. In this representation, adjacency between two vertices a and b may be expressed by the formula

$$\text{ADJACENT}(a, b, S) \equiv \exists e \in S : (e \multimap a \wedge e \multimap b).$$

The following formula expresses the property that the neighbors of vertex ℓ in subgraph S include at most one vertex from a set X:

$$\text{LEAF}(\ell, X, S) \equiv \forall c, d \in X : \Big(\big(\text{ADJACENT}(\ell, c, S) \wedge \text{ADJACENT}(\ell, d, S)\big) \rightarrow c = d\Big).$$

This allows us to express the acyclicity of a subgraph S in terms of 1-degeneracy: every nonempty subset X of vertices contains a leaf.

$$\text{ACYCLIC}(S) \equiv \forall X \subset V : (\exists x \in X) \rightarrow \exists \ell \in X : \text{LEAF}(\ell, X, S).$$

This already allows us to express the first condition of Lemma 2.2. We will also use a predicate for whether two vertices p and q are connected by horizontal edges. This is true if for every subset C of vertices containing p and excluding q, there exists a horizontal edge, connecting a vertex of C to a vertex not in C.

$$\text{ALIGNED}(p, q) \equiv \forall C \subset V : \big(p \in C \wedge \neg(q \in C)\big) \rightarrow \exists h \in \text{HORIZONTAL} : \exists y, z \in V : \ \big(y \in C \wedge \neg(z \in C) \wedge h \multimap y \wedge h \multimap z\big).$$

This allows us to express a predicate for the property that vertex ℓ is a leaf of subgraph S on the same horizontal level as another vertex v (that is, ℓ is the representative leaf for v's level):

$$\text{REPRESENTATIVE}(v, \ell, S) \equiv \text{LEAF}(\ell, V, S) \wedge \text{ALIGNED}(v, \ell).$$

The second part of Lemma 2.2 is that every level has exactly one representative leaf:

$$\begin{aligned}&\textsc{represented}(S) \equiv \big(\forall v \in V : \exists \ell \in V : \textsc{representative}(v, \ell, S)\big)\wedge\\ &\Big(\forall v, \ell_1, \ell_2 \in V : \big(\textsc{representative}(v, \ell_1, S) \wedge \textsc{representative}(v, \ell_2, S)\big)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to \ell_1 = \ell_2\Big).\end{aligned}$$

Unlike for the previous formulas, there is no way of expressing the existence of a path of length k from u to v in subgraph S, for a non-fixed k, in MSO_2. We need a different formula $\textsc{path}_k$ for each k. We do not require these paths to be simple, as this would only complicate the formula without simplifying our use of it. However it is essential for our application to the third condition of Lemma 2.2 that we require our paths to have distinct endpoints.

$$\begin{aligned}&\textsc{path}_k(u, v, S) \equiv \exists w_1, w_2, \ldots w_{k-1} \in V : \exists e_1, e_2, \ldots e_k \in S :\\ &\qquad\qquad \neg(u = v) \wedge e_1 \multimap u \wedge e_1 \multimap w_1 \wedge e_2 \multimap w_1 \wedge \cdots \wedge e_k \multimap w_{k-1} \wedge e_k \multimap v.\end{aligned}$$

Other than the inequality of the two endpoints, this formula allows repetitions of vertices and edges within each path. In particular, it allows w_i and w_{i+1} to be equal to each other, repeating one endpoint of an edge twice and omitting the other endpoint. Because we allow repetitions in this way, this formulation of the $\textsc{path}$ predicate has the following convenient property:

Lemma 2.3. *For all $k \geq 1$ and all u, v, and S, we have that*

$$\textsc{path}_k(u, v, S) \to \textsc{path}_{k+1}(u, v, S).$$

Proof. Let $w_1, \ldots w_{k-1}$ and $e_1, \ldots e_k$ be the vertices and edges witnessing the truth of $\textsc{path}_k(u, v, S)$, let $w_k = v$, and let $e_{k+1} = e_k$. Then $w_1, \ldots, w_k$ and $e_1, \ldots, e_{k+1}$ witness the truth of $\textsc{path}_{k+1}(u, v, S)$. □

Corollary 2.1. *Two vertices u and v of a subgraph S of a given graph obey the predicate* $\textsc{path}_k(u, v, S)$ *if and only if they are distinct and their distance in S is at most k.*

This allows us to express the final part of Lemma 2.2, the requirement that each two vertices are connected by a non-horizontal edge if and only if their representatives are connected by a short path:

$$\textsc{root}_k(S) \equiv \forall u, v \in V : \Big(\big(\exists u', v' \in V\ \exists e \in E : \textsc{aligned}(u, u') \wedge \textsc{aligned}(v, v') \wedge e \multimap u' \wedge e \multimap v' \wedge \neg(e \in \textsc{horizontal})\big) \longleftrightarrow \exists x, y \in V : \big(\textsc{representative}(u, x, S) \wedge \textsc{representative}(v, y, S) \wedge \textsc{path}_k(x, y, S)\big)\Big).$$

Lemma 2.4. *There exists an* MSO_2 *formula that is modeled by a graph $G \boxtimes C_k$ and its set* $\textsc{horizontal}$ *of horizontal edges exactly when $G \boxtimes C_k$ meets the conditions of Lemma 2.2.*

Proof. The formula is

$$\exists S : \big(\textsc{acyclic}(S) \wedge \textsc{represented}(S) \wedge \textsc{root}_k(S)\big).$$

A subgraph defined by a set S of its edges meets the first condition of the lemma if $\textsc{acyclic}(S)$ is true, it meets the second condition of the lemma if $\textsc{represented}(S)$ is true, and it meets the third condition of the lemma if $\textsc{root}_k(S)$ is true. □

Corollary 2.2. *The property of a graph G being k-leaf power can be expressed as an* MSO_2 *formula of $G \boxtimes C_k$ and of the set* $\textsc{horizontal}$ *of horizontal edges of this graph product.*

2.5 Fixed-Parameter Tractability of Leaf Powers

In this section, by using Courcelle's theorem, we provide our main result that recognizing k-leaf powers is fixed-parameter tractable when parameterized by k and the degeneracy of the input graph.

In order to apply Courcelle's theorem to the graph product $G \boxtimes C_k$ we need to bound its treewidth.

Lemma 2.5. *If G has treewidth t and H has a bounded number of vertices s then $G \boxtimes H$ has treewidth at most $s(t+1)-1$.*

Proof. Given any tree-decomposition of G with width t, we can form a decomposition of $G \boxtimes H$ by using the same tree, and placing each vertex (v, w) of $G \boxtimes H$ (where v and w are vertices of G and H respectively) into the same bag as vertex v of G. The size of the largest bag of the tree-decomposition of G is $t+1$, so the size of the largest bag of the resulting tree-decomposition of the graph product is $s(t+1)$. The treewidth is one less than the size of the largest bag. □

Corollary 2.3. *If G has a bounded treewidth and k is bounded, then $G \boxtimes C_k$ also has bounded treewidth.*

This gives us our main theorem:

Theorem 2.1. *For fixed constants k and d, it is possible to recognize in linear time (with fixed-parameter tractable dependence on k and d) whether a graph of degeneracy at most d is a k-leaf power.*

Proof. As stated in subsection 2.1.2, all leaf powers are chordal graphs and it is known for a chordal graph, treewidth is equal to maximum clique number minus one [103]. This implies that treewidth of leaf powers is equal to their degeneracy. Further, from Corollary 2.3, $G \boxtimes C_k$ also has bounded treewidth. Therefore, by applying Courcelle's theorem to the MSO_2 formula of Corollary 2.2 we obtain the result. $\square$

2.6 Edges Labeled by Distance Ranges

It is perhaps of interest to generalize k-leaf powers to a more general version in which each edge of the input graph G has a weight range $[k_1, k_2]$ where $2 \leq k_1 \leq k_2$ and K is the upper bound on k_2 over all the edges. We say that G is a *labeled K-leaf power* if G has a K-leaf root T in which, for each edge uv of G, the corresponding leaves of T are at a distance that is within the range used to label edge uv. As with the unlabeled version of the problem, for non-adjacent pairs of vertices of G, the corresponding leaves should be at distance more than K. The original k-leaf power is a restricted variant of this general version in which all edges have a fixed weight range $[1, k]$ and $K = k$.

One motivation for this comes from the phylogenetic tree applications of k-leaf powers. If we know some information about the evolutionary distance between species, and wish to reconstruct the evolutionary tree, the information we know may be more fine-grained than merely that the distance is big or small. The ranges on each edge allow us to model this fine-grained information and by doing so restrict the trees that can be generated to more accurately reflect the data. As we show in this section, our parameterized algorithms can be extended to the more general problem of recognizing labeled K-leaf powers.

Recall that we are already modeling some labeling information on the graph product $G_1 \boxtimes G_2$, in the logic of graphs, as the free set variable HORIZONTAL. We will similarly need to model the edge weight range labels logically. To do so, we extend the weights on the edges of G to the weights on the edges of a graph product using the following definition. Suppose that we are considering the graph product $G_1 \boxtimes G_2$ where G_1 and G_2 are weighted and unweighted, respectively. Recall that, in this product, two distinct vertices (u_1, u_2) and (v_1, v_2) are adjacent if and only if for all $i \in \{1, 2\}$, $u_i = v_i$ or u_i and v_i are adjacent in G_i. A vertical or diagonal edge is an edge with endpoints (u_1, u_2) and (v_1, v_2), for which $u_1 \neq v_1$. In this case, we assign the vertical or diagonal edge weight ω if the edge connecting u_1 and v_1 has weight ω, in G_1.

We have the following analogue of Lemma 2.1 for the weighted case:

Lemma 2.6. *If G is a weighted connected K-leaf power graph on three or more vertices, and T is any K-leaf root of G, then T can be embedded as a subtree of the strong product $G \boxtimes C_K$. Additionally, the embedding can be chosen in such a way that each horizontal cycle in the strong product (the product of a vertex v of G with C_K) contains exactly one leaf of the embedded copy of T, the leaf representing v.*

Proof. The weighted graph product has the same underlying graph as the unweighted product, and the weighted K-leaf root is a special case of the unweighted K-leaf root, so this follows immediately from Lemma 2.1, which provides an embedding into the graph power of every K-leaf root. □

We can now provide the following characterization of K-leaf powers.

Lemma 2.7. *A given connected weighted graph G on three or more vertices is a K-leaf power if and only if the product $G \boxtimes C_K$ has a subgraph T with the following properties:*

1. *T is* 1*-degenerate (i.e., a forest).*
2. *Every vertex of $G \boxtimes C_K$ is connected by horizontal edges of the product to exactly one leaf of T.*

3. *If two vertices of $G \boxtimes C_K$ are the endpoints of a non-horizontal edge of the product with weight $[k_1, k_2]$ then the corresponding leaves of T (given according to Property 2) are the distinct endpoints of a path of length at least k_1 and at most k_2 in T.*

4. *If two distinct leaves of T are at distance at most K then there exists a non-horizontal edge of the product with two endpoints vertices, aligned to each leaf.*

Proof. The proof follows the same lines as the proof of Lemma 2.2, modified only to take into account the edge weights. □

In order to express the components of Lemma 2.7 in monadic second-order logic, we reuse formulas ACYCLIC and REPRESENTED from Lemma 2.2 for the first and second parts of Lemma 2.7, respectively.

To express the third part, we introduce K^2 edges sets I_{k_1,k_2} where $2 \leq k_1 \leq k_2 \leq K$. An edge e of the product, with two endpoints (u_1, u_2) and (v_1, v_2) belongs to I_{k_1,k_2} if and only if $u_1 \neq u_2$, $v_1 \neq v_2$ and it has weight $[k_1, k_2]$. This allows us the express the requirement that if two vertices are connected by a non-horizontal edge with weight $[k_1, k_2]$ then their representatives are connected by a path with a length in the range $[k_1, k_2]$:

$$\begin{aligned}
&\textsc{edge}_{k_1,k_2}(S) \equiv \forall u, v \in V : \Big(\big(\exists e \in E : e \multimap u \wedge e \multimap v \wedge (e \in I_{k_1,k_2}) \big) \\
&\longrightarrow \exists x, y \in V : \big(\textsc{representative}(u, x, S) \wedge \textsc{representative}(v, y, S) \wedge \\
&\qquad\qquad\qquad \textsc{path}_{k_2}(x, y, S) \wedge \neg\textsc{path}_{k_1-1}(x, y, S) \big) \Big).
\end{aligned}$$

The last part of Lemma 2.7 can be expressed as follows:

$$\textsc{nonedge}_K(S) \equiv \forall u, v \in V : \Big(\big(\exists x, y \in V : \textsc{representative}(u, x, S) \wedge \textsc{representative}(y, v, S) \wedge \textsc{path}_K(x, y, S)\big) \longrightarrow \big(\exists u', v' \in V \; \exists e \in E : e \multimap u' \wedge e \multimap v' \wedge \textsc{aligned}(u, u') \wedge \textsc{aligned}(v, v') \wedge \neg(e \in \textsc{horizontal})\big)\Big)$$

Lemma 2.8. *There exists an* MSO_2 *formula that is modeled by a graph* $G \boxtimes C_K$ *and its set* HORIZONTAL *of horizontal edges and* K^2 *edge sets* I_{k_1,k_2} *exactly when* $G \boxtimes C_K$ *meets the conditions of Lemma 2.7.*

Proof. The formula is

$$\exists S : \big(\textsc{acyclic}(S) \wedge \textsc{represented}(S) \wedge \textsc{edge}_{2,2}(S) \wedge \textsc{edge}_{2,3}(S) \wedge \cdots \wedge \textsc{edge}_{K,K}(S) \wedge \textsc{nonedge}_K(S)\big).$$

A subgraph defined by a set S of its edges meets the first condition of the Lemma 2.7 if $\textsc{acyclic}(S)$ is true, it meets the second condition of the lemma if $\textsc{represented}(S)$ is true, it meets the third condition of the lemma if $\textsc{edge}_{k_1,k_2}(S)$ is true for all $2 \leq k_1 \leq k_2 \leq K$, and it meets the forth condition of the lemma if $\textsc{nonedge}(S)$ is true. □

Corollary 2.4. *The property of a weighted graph* G *being* K*-leaf power can be expressed as an* MSO_2 *formula of* $G \boxtimes C_K$*, of the set* HORIZONTAL *of horizontal edges and of the* K^2 *edge sets* I_{k_1,k_2} *of this graph product.*

As proved in Lemma 2.5, if G has a bounded treewidth and K is fixed, then $G \boxtimes C_K$ also has a bounded treewidth. This fact enables us to provide the following theorem for the general leaf power problem.

Theorem 2.2. *For fixed constants* K *and* d*, it is possible to recognize in linear time (with* *fixed-parameter tractable dependence on K and d) whether a graph of degeneracy at most d is a K-leaf power.*

Proof. The proof follows the same outline as the proof of Theorem 2.1, modified only to use the weighted versions of the lemmas above in place of their unweighted versions. □

2.7 Dynamic Programming Algorithm

Many graph problems, including a vast number of NP-hard problems, have been shown to be solvable in polynomial time when given a tree decomposition of constant width [8, 14, 15]. Dynamic programming on tree decomposition of graphs is an underlying technique to devise such algorithms, restricted to graphs of bounded treewidth [14]. Indeed, our application of Courcelle's theorem relies on such an algorithm to evaluate whether a logical formula is modeled by the given graph. In this section, we present a direct dynamic programming algorithm to decide whether the input graph is a k-leaf power.

Dynamic programming algorithms often use a variant of tree decomposition, called *nice* tree decomposition. A nice tree decomposition of graph G is a rooted tree decomposition T of G in which each bag X_i is one of the following:

- a *leaf* bag in which $|X_i| = 1$,
- a *forget* bag with one child X_j, where $X_i \subset X_j$ and $|X_j| - |X_i| = 1$,
- an *introduce* bag with one child X_j, where $X_j \subset X_i$ and $|X_i| - |X_j| = 1$, or
- a *join* bag with two children X_j and $X_{j'}$, where $X_i = X_j = X_{j'}$,

For a forget bag we call $X_j \backslash X_i$ the *forgotten vertex.* Given a graph G and its tree decomposition of width w, one can construct a nice tree decomposition of equal width in linear time [85].

Our algorithm uses these restrictions on tree decompositions, but we need others as well. Therefore, we will define an *extra* nice tree decomposition. In comparison with nice tree decomposition, an extra nice tree decomposition has one more type of bag, an *edge-associated* bag. An edge-associated bag X_i has a child X_j where $X_i = X_j$ and exactly one edge $e(u, v)$, $u, v \in X_i$, is associated with X_i. Using a nice tree decomposition of G, we can simply construct such tree decomposition in the following way: for each pair of adjacent pairs u and v in bag X_i, if $e(u, v)$ is not yet associated to a bag, create a new bag $X_{i'}$ as a new parent of X_i where $X_{i'} = X_i$ and associate edge $e(u, v)$ to $X_{i'}$. The old parent of X_i, if it exists, is now the parent of $X_{i'}$.

Our algorithm is run over a *mixed* decomposition of graphs G and graph product H. Given an extra nice decomposition of G of width w, for each vertex v in bag X_i, add all vertices $(v, r) \in H$ for $0 \leq r < k$. Hence, the size of each bag of the mixed decomposition is at most wk. Our second algorithm can therefore be viewed as using the same graph product technique that our first algorithm used, applied directly in a dynamic programming algorithm rather than indirectly via Courcelle's theorem.

2.7.1 Local Picture of a k-leaf root

Intuitively, for each bag of mixed decomposition M, we describe a local picture which describes a subtree of a k-leaf root T upon existence. This description allows us to check whether the big picture, T, is a k-leaf root of G. For a bag X_i let G_i and H_i be a set of vertices of X_i that belongs to G and H, respectively.

A local picture of T at bag X_i consists of the following ingredients:

- A partition of H_i into connected components (with one more partition set for vertices of H_i, not participating in T).

- A distance matrix between each pair of vertices in the same component. Each coefficient of the matrix will store either a number between 1 and k (the distance between two vertices), or a special flag ∞ to represent a finite distance greater than k.
- A designated root vertex for each component, the vertex that will become the closest to the root of T.
- For each vertex v of G_i in X_i, a corresponding vertex (v, i) chosen as the leaf representative of v in H_i.

To reduce the number of local pictures that we need to consider, consistently with the embedding of Section 2.3, we will restrict our attention to local pictures in which the vertices (v, i) of H_i associated with a single vertex v of H_i are either part of a single component or not in any component, and have distances within that component consistent with their distances along the cycle C_k. We will associate with each remaining local picture a Boolean variable. We will set this variable to True if there is a subtree of H within the bags descending from X_i that is consistent with the local picture and with the requirement that it be part of a leaf root of G. Otherwise, we set this variable to False. In order to enforce the requirement that the local picture be consistent with being part of a leaf root, we only consider local pictures such that, for the distances in each component, the pairs of representative vertices at distance at most k are adjacent in G and pairs with distance ∞ are non-adjacent. Adjacent vertices in G_i whose representatives belong to different components are allowed, however, as their distance will be checked at a higher level of the tree decomposition where their components merge. If these conditions are not met, we set the associated Boolean variable of the local picture to False.

We process M in post-order from leaves to the root of M computing for each bag and each local picture the Boolean variable for that local picture. This bottom-up ordering ensures that the variables for local pictures of the child or children of a bag are known before we try to compute the variables at the bag itself. After computing these values, G will be a k-leaf power if and only if there exists a local picture at the root bag whose associated Boolean variable is true. If G is a k-leaf power, one can form a k-leaf root by creating a vertex as the root of the k-leaf root and connect it to the root of each component of the True local picture, with an appropriate number of edges (at most k edges for each connection). Further, such ordering allows us to remember the distance to the nearest forgotten leaf as μ_v for each non-leaf vertex v of each local picture for distance-checking purposing. In another word, μ_v stores the distance from v to the nearest forgotten leaf that is no more present in the current local picture. When the bottom-up traversal of M reaches a bag X_i, one of the following cases occurs:

- X_i may be a leaf of M. In this case, it contains a vertex $v \in G$ alongside all vertices (v, r), $0 \leq r < k$. A local picture is set to True if and only if it has one component, a single chain of vertices with the appropriate distances, ending at the vertex designated as the representative of v.

- X_i may be a forget bag. In this case, it has one child X_j where $X_i \subset X_j$ and $H_j \wedge (X_j \backslash X_i) = \{(v, r)\}, 0 \leq r < k$. A local picture ℓ at X_i is set to True if and only if it is formed by removing vertices (v, r) (a chain of vertices representing $v \in G_j$) from a True local picture ℓ' of X_j.The removal of such chain of vertices may result in more number of components in the corresponding True local picture ℓ. If a removed vertex has a child other than the one in the chain, that child becomes the root of a new component in ℓ. Further, as the designated leaf of such chain is forgotten, there might be a need to update μ_u for a vertex u in ℓ within the vicinity ($< k$) of the forgotten leaf.

- X_i may be an introduce bag. In this case, it has one child X_j where $X_j \subset X_i$ and $G_i \wedge (X_i \backslash X_j) = \{v\}$. A local picture at X_i is set to True if and only if it can be formed from one of the True local pictures of X_j by adding one more component which is a path $(v, r), \ldots, (v, r')$, $0 \leq r, r' < k$. Because the subtree descending from X_i does not contain any edge-associated bags for edges incident with v, this component cannot be connected to any of the existing components in the local picture in X_j.

- X_i may be an edge-associated bag. In this case, it has one child X_j where $X_i = X_j$ and there exists an edge $e(u, v)$ associated to bag X_i. A local picture L at X_i is set True if and only if either there exists an exact True copy of the local picture at X_j, or using the edge $e(u, v)$, L can be formed from a True local picture at X_j by connecting a root x of one component to a vertex w of another component. Such connection can be made if the resulting local picture obeys the distance matrix and also the distance from each forgotten leaf of one component to a (forgotten or existing) leaf of another component is greater than k as their corresponding vertices in G cannot be adjacent given the definition of extra nice decomposition (when a vertex is forgotten, it cannot be reintroduced as the bags containing that vertex form a nonempty connected subtree).

- X_i may be a join bag. In this case, it has two children X_j and $X_{j'}$ where $X_i = X_j = X_{j'}$. A local picture L_1 at X_i has its value set to True if and only if there exist True local pictures L_2 and L'_2 at X_j and $X_{j'}$, respectively, that when combined together, they form L_1. To find such a combination, we consider all pairs of local pictures for L_2 and L'_2 at X_j and $X_{j'}$ and construct a bipartite graph F. One side of bipartition includes vertices of $H_i \in X_i$, each with two neighbors, representing the two subtrees, the vertex belongs to in the local pictures L_2 and L'_2. L_1 can be formed if and only if F is a forest, its subtrees are subtrees of F and the combined local picture obeys the distance matrix at L_1 and no forgotten or existing leaf of L_2 get a distance at most k to a forgotten leaf of L'_2 or vice versa.

2.7.2 Analysis

To analyze our dynamic programming algorithm, we need to understand the number of local pictures that are possible in each bag of the tree decomposition. We can perform this analysis by combining the following factors, each of which depends only on the width w and leaf power parameter k of the given input.

- For each vertex v of G_i, there are $O(k^2)$ choices for the representative vertex and the length of the path using vertices (v, i) in the component of this representative vertex. The total number of such choices for all vertices of G_i is $k^{O(w)}$.
- Given these choices of paths, there are $w^{O(w)}$ ways of connecting the paths into components and selecting the vertex closest to the root within each component.
- Within a component that connects c paths, there are $(ck)^{O(w)}$ choices of distance matrix for the whole component consistent with the distances within each path and with the assumption that the distances come from a tree.

Therefore, there are $(wk)^{O(w)}$ local pictures considered by our algorithm for each bag. The time for the algorithm is dominated by the join bags; there are $n - 1$ of these bags, and in each such bag we consider a number of pairs of local pictures bounded by the square of the number of local pictures per bag. Each pair of local pictures in the two child bags takes time polynomial in w and k to check for whether it is consistent and to find the corresponding local picture in the join bag. So the total time for our dynamic programming algorithm is $O\big(n(wk)^{O(w)}\big)$.

2.8 Conclusion

We have provided two fixed-parameter algorithms to recognize k-leaf powers (and generalized K-leaf powers) for graphs of bounded degeneracy. In both methods we use embeding of a k-leaf root of a k-leaf power graph in the graph product of the input graph and a k-vertex cycle C_k. Our first algorithm finds a logical characterization of the leaf roots that are embedded in this way, and applies Courcelle's theorem to determine the existence of a subgraph of the graph product that meets our characterization.

Our methods of using low-treewidth supergraphs to represent vertices and edges that are not part of the input graph, and of using graph products to find these supergraphs helped us to solve the problem directly using dynamic programming rather than by applying Courcelle's theorem. Additionally, these methods may be useful in other graph problems. For instance, the same graph product technique would have greatly simplified the application of Courcelle's theorem in our recent work on planar split thickness [59]: a graph G has planar split thickness k if and only if $G \boxtimes K_k$ has a planar subgraph S such that, for each non-horizontal edge of the product, the endpoints of the edge are aligned with the endpoints of an edge in S. In reducing the logical complexity of problems such as these, our first method also makes it more likely that faster model checkers for restricted fragments of MSO logic [9] can be applied to our problem.

Our dynamic programming algorithm has significantly better dependence on its parameters than our first, logic-based algorithm. However, its dependence is still not singly exponential. We leave whether this is possible as open for future research.

Chapter 3

Parameterized Complexity of Finding Subgraphs with Hereditary Properties on Hereditary Graph Classes [1]

3.1 Introduction

In this chapter, we study the parameterized complexity of finding k-vertex induced subgraphs in a given hereditary class of graphs, within larger graphs belonging to a different hereditary class of graphs. A prototypical instance of the induced subgraph problem is the k-clique problem, which asks whether a given graph G has a clique of size k. Although k-clique is W[1]-complete for general graphs [48], and NP-complete even when the input graph is constrained to be a multiple-interval graph, [26], it is fixed-parameter tractable in this special case [63]. This example, of a W[1]-complete problem for general graphs which becomes FPT on constrained inputs, motivates us to seek additional examples of this phenomenon, and

[1]The material in this chapter is from unpublished work with Eppstein and Gupta [56].

more broadly to attempt a classification of induced subgraph problems which can determine in many cases whether a constrained induced subgraph problem is tractable or remains hard.

We formalize a graph property as a set Π of the graphs that have the property. A property is *nontrivial* if it is neither empty nor contains all the graphs, and more strongly it is *interesting* if infinitely many graphs have the property and infinitely many graphs do not have the property. A nontrivial graph property Π is *hereditary* if it is closed under taking induced subgraphs. That is, if Π is hereditary and a graph G belongs to Π, then every induced subgraph of G also belongs to Π. Given a hereditary property Π, let $\overline{\Pi}$ be the complementary property, the set of graphs which do not belong to Π. The *forbidden* set $\mathcal{F}_\Pi$ of Π is the set of graphs that are minimal for $\overline{\Pi}$: they belong to $\overline{\Pi}$, but all of their proper induced subgraphs belong to Π. For a hereditary property Π, a graph G belongs to $\overline{\Pi}$ if and if G has no induced subgraph in $\mathcal{F}_\Pi$. Khot and Raman [84] studied the parameterized complexity of the following unified formulation of the induced-subgraph problem, without constraints on the input graph: Given a graph G, an interesting hereditary property Π and a positive integer k, the problem $P(G, \Pi, k)$ asks whether there exists an induced subgraph of G of size k that belongs to Π. They proved a dichotomy theorem for this problem: If Π includes all trivial graphs (graphs with no edges) but not all complete graphs, or vice-versa, then the problem is $\mathsf{W}[1]$-complete. However, in all remaining cases, the problem is FPT.

Our work studies the parameterized complexity of the problem $P(G, \Pi, k)$, in cases for which it is $\mathsf{W}[1]$-complete for general graphs, under the constraint that the input graph G belongs to a hereditary graph class Π_G. (Note that Π_G should be a different class than Π, for otherwise the problem is trivial: just return any k-vertex induced subgraph of the input.) Given a graph G, the interesting hereditary properties Π_G and Π, and an integer k, we denote our problem by $P(G, \Pi_G, \Pi, k)$. The main tool that we use for finding efficient algorithms for $P(G, \Pi_G, \Pi, k)$ is Ramsey's theorem, which allows us to prove the existence of either large cliques or large independent sets in arbitrary graphs, allowing some combinations of input graph size and parameter to be answered immediately without performing a search. For the cases where we find hardness results, we do so by reductions from $P(G, \Pi_G, \mathsf{IS}, k)$ to $P(G, \Pi_G, \Pi, k)$, where IS is the property of being an independent set. We believe our framework has interest in its own right, as a way to settle a wide class of induced-subgraph properties while avoiding the need to develop many tedious hardness proofs for individual problems.

3.1.1 Our Contributions

We partition interesting hereditary properties into four classes named AA, AS, SA, and SS as follows. A hereditary property Π belongs to:

- AA, if it includes all complete graphs and all independent sets.
- AS, if it includes all complete graphs but excludes some independent sets.
- SA, if it excludes some complete graphs but includes all independent sets.
- SS, if it excludes some complete graphs as well as some independent sets.

By Ramsey's theorem, an interesting hereditary property cannot belong to SS. The interesting cases for the problem $P(G, \Pi_G, \Pi, k)$ with respect to Π are either $\Pi \in \mathsf{SA}$ or $\Pi \in \mathsf{AS}$. In the other two cases, when $\Pi \in \mathsf{AA}$ or $\Pi \in \mathsf{SS}$ the problem $P(G, \Pi_G, \Pi, k)$ is known to be fixed-parameter tractable regardless of Π_G [84] . We prove the following results related to the problem $P(G, \Pi_G, \Pi, k)$, for these interesting cases:

- If $\Pi_G \in \mathsf{AS}$ and $\Pi \in \mathsf{SA}$ or vice versa, then the problem $P(G, \Pi_G, \Pi, k)$ is solvable in polynomial time (Theorem 3.1). Although the exponent of the polynomial depends in general on Π, some classes Π_G for which subgraph isomorphism is in FPT also have polynomial-time algorithms for $P(G, \Pi_G, \Pi, k)$ whose exponent is fixed independently of Π (Theorem 3.2). The key insight for these problems is that these assumptions cause $\Pi_G \cap \Pi$ to be a finite set, limiting the value of k and making it possible to perform a brute-force search for an induced subgraph while remaining within polynomial time.

A class of problems of this form that has been extensively studied involves finding cliques in sparse graphs or sparse classes such as planar graphs; beyond being polynomial for any fixed hereditary sparse AS or class of graphs, it is FPT for general graphs when parameterized by degeneracy, a parameter describing the sparsity of the given graph [62]. Another example problem of this type that is covered by this result is finding planar induced subgraphs of co-bipartite graphs; here, Π is the property of being planar, in SA, and Π_G is the property of being co-bipartite, in AS. Similarly, this result covers finding a k-vertex bipartite or triangle-free induced subgraph of a co-bipartite graph, or finding a k-vertex co-bipartite induced subgraph of a planar, bipartite, or triangle-free graph.

- If both Π_G and Π belong either to AS or both belong to SA, then the problem $P(G, \Pi_G, \Pi, k)$ is in FPT (Theorem 3.3). The insight that leads to this result is that large-enough graphs in Π_G necessarily contain k-vertex cliques (for properties in AS) or independent sets (for properties in SA), which also belong to Π. Therefore, the only instances for which a more complicated search is needed are those for which k is large enough relative to G that the existence of a k-vertex clique or independent set cannot be guaranteed. For that range of the parameter k, the search complexity is in FPT.

  Problems of this type that have been studied previously include finding independent sets in sparse graph families, as well as finding planar induced subgraphs of sparse classes of graphs [18]. Finding a k-vertex graph that belongs to one of the four classes of forests, planar graphs, bipartite graphs, or triangle-free graphs, as an induced subgraph of a graph G that belongs to another of these three classes, belongs to the problems of this type.

- If $\Pi_G \in \mathsf{SS}$, then the problem $P(G, \Pi_G, \Pi, k)$ is solvable in polynomial time (Theorem 3.4). This case is trivial: there can be only finitely many graphs in Π_G and we can precompute the answers to each one.

- In the remaining cases, $\Pi_G \in \mathsf{AA}$, while Π belongs to AS or to SA. These cases include both problems known to be polynomial, such as finding independent sets in various classes of perfect graphs, problems known to be FPT, including several other cases of independent sets [43], and problems known to be hard for parameterized computation, such as finding independent sets in unit disk graphs [94]. Therefore, we cannot expect definitive results that apply to all cases of this form, as we obtained in the previous cases. Instead, we provide partial results suggesting that in many natural cases the complexity of $P(G, \Pi_G, \Pi, k)$ is controlled by the complexity of the simpler problem of finding independent sets:

  – If Π_G is closed under duplication of vertices (strong products with complete graphs), and Π contains the graphs $n \cdot K_{\chi(\Pi)}$ (disjoint unions of complete graphs with the maximum chromatic number for Π), then $P(G, \Pi_G, \Pi, k)$ is as hard as $P(G, \Pi_G, \mathsf{IS}, k)$ (Theorem 3.5).

    Families Π_G that meet these conditions, for which finding independent sets is $\mathsf{W}[1]$-complete, include the property of being a unit disk graph, the property of being C_4-free, and the property of being $K_{1,4}$-free. Families Π that meet these conditions include the property of being either planar or bipartite. Therefore, $P(G, \Pi_G, \Pi, k)$ is also $\mathsf{W}[1]$-complete in these families.

  – If $\Pi_G \in \mathsf{AA}$ and is closed under joins with disjoint unions of cliques, and if Π contains all joins of an independent set with a disjoint union of cliques that have chromatic number at most $\chi(\Pi) - 1$, then $P(G, \Pi_G, \Pi, k)$ is as hard as $P(G, \Pi_G, \mathsf{IS}, k)$ (Theorem 3.6).

3.1.2 Other Related Work

Before the investigation of the parameterized complexity of $P(G, \Pi, k)$, Lewis and Yannakakis had studied the dual of this problem, the Node Deletion problem, for interesting hereditary properties, which is defined as follows: Given a graph G and an interesting hereditary property Π, find the minimum number of nodes to delete from G such that the resulting graph belongs to Π. They proved that the Node Deletion problem is NP-complete [91]. Cai [27] studied the parameterized version of Node Deletion and proved that the problem is FPT, parameterized by the number of deleted vertices, for an interesting hereditary property with a finite forbidden set.

Related to our line of work on the parameterized complexity of hereditary properties, finding an independent set with the maximum cardinality (MIS) on a general graph, has been proved to be NP-hard even for planar graphs of degree at most three [65], unit disk graphs [36], and C_4-free graphs [4]. Fellows, Hermelin, Rosamond and Vialette proved that finding a k-Independent Set is W[1]-hard for 2-interval graphs while its complementary problem, k-clique, as mentioned before is FPT for multiple-interval graphs [63].

3.2 Preliminaries

Throughout the chapter, we consider finite undirected graphs. Given a graph G, we denote its vertex set and edge set by $V(G)$ and $E(G)$, respectively. For a vertex $v \in V(G)$, we denote the set of all adjacent vertices of v in G by $N_G(v)$, i.e. $N_G(v) = \{u \in V(G) \mid \{u, v\} \in E(G)\}$. The degree of a vertex $v \in V(G)$ in G is denoted by $\deg_G(v)$. Given a vertex set $S \subseteq V(G)$, $G[S]$ represents the subgraph of G induced by S. The maximum clique size of G is denoted by $\omega(G)$. The maximum clique size of a graph property Π, denoted by $\omega(\Pi)$, is the maximum clique size of any graph $G \in \Pi$. The *chromatic number*, $\chi(G)$, of G is the minimum number of colors needed to color the vertices such that no two adjacent vertices get the same color. The chromatic number, $\chi(\Pi)$, of a graph property Π is the maximum chromatic number of any graph $G \in \Pi$.

Let Π be a hereditary graph property. If $\Pi \in \mathsf{AS}$ or $\Pi \in \mathsf{SS}$, then we denote the size of the smallest independent set that does not belong to Π by i_Π. Similarly, if $\Pi \in \mathsf{SA}$ or $\Pi \in \mathsf{SS}$, then we denote the number of vertices in the smallest clique that does not belong to Π by c_Π. Observe that, $c_\Pi = \omega(\Pi) + 1$. We denote the property of being an independent set (the family of all all independent sets) as IS.

3.3 Tractability Results

In this section, we identify pairs of hereditary properties Π_G and Π for which the problem $P(G, \Pi_G, \Pi, k)$ is either in P or FPT. Our proofs use Ramsey numbers which we begin by defining. For any positive integers r and s, there exists a minimum positive integer $R(r, s)$ such that any graph on at least $R(r, s)$ vertices contains either a clique of size r or an independent set of size s. It is well-known that $R(r, s) \leq \binom{r+s-2}{r-1}$ [73]. It will also be convenient in our analysis to have a notation for the time to test whether a given k-vertex graph (typically, a subgraph of our given graph G) has property Π; we let $t_\Pi(k)$ denote this time complexity.

Theorem 3.1. *If $\Pi_G \in \mathsf{AS}$ and $\Pi \in \mathsf{SA}$ or vice versa, then the problem $P(G, \Pi_G, \Pi, k)$ is solvable in polynomial time.*

Proof. We give a proof for the case when $\Pi_G \in \mathsf{AS}$ and $\Pi \in \mathsf{SA}$. The proof for the other case is symmetric under reversal of the roles of cliques and independent sets. Recall that every graph on $R(c_\Pi, i_{\Pi_G})$ vertices contains either a clique of size c_Π, too large to have property Π, or it contains an independent set of size i_{Π_G}, too large to have property Π_G. Therefore, If

$k \geq R(c_\Pi, i_{\Pi_G})$, it is impossible for a k-vertex induced subgraph of a graph G in Π_G to also have property Π, because such a subgraph would either have a large clique (contradicting the membership of the subgraph in Π) or a large independent set (contradicting the membership of G in Π_G). Therefore, for such large values of k, an algorithm for $P(G, \Pi_G, \Pi, k)$ can simply answer No without doing any searching.

If $k < R(c_\Pi, i_{\Pi_G})$, then we can use a brute force search to test whether there exists a k-vertex induced subgraph having property Π. Specifically, we enumerate all k-vertex subsets of the vertices of G, construct the induced subgraph for each subset, and test whether any of these induced subgraphs belongs to Π. Given a representation of G for which we can test adjacency in constant time, the time to construct each subgraph is $O(k^2)$, so the total time taken by this search is

$$\binom{n}{k}\left(O(k^2) + t_\Pi(k)\right) \leq n^r\left(O(r^2) + t_\Pi(r)\right),$$

where $r = R(c_\Pi, i_{\Pi_G}) - 1$. As the right hand side of this time bound is a polynomial of n without any dependence on k, this is a polynomial time algorithm. Thus, the problem $P(G, \Pi_G, \Pi, k)$ is solvable in polynomial time. □

Although polynomial, the time bound of Theorem 3.1 has an exponent r that depends on Π and Π_G, and may be large. An alternative approach, which we outline next, may lead to better algorithms for properties Π_G for which the induced subgraph isomorphism problem is in FPT, as it is for instance for planar graphs [52] or more generally for nowhere-dense families of graphs [97].

Theorem 3.2. *If $\Pi_G \in$ AS and $\Pi \in$ SA or vice versa, and induced subgraph isomorphism is in* FPT *in Π_G with time $t_{\text{sgi}}(n, k)$ to find k-vertex induced subgraphs of n-vertex graphs, then the problem $P(G, \Pi_G, \Pi, k)$ is solvable in polynomial time $O(t_{\text{sgi}}(n, r))$, for the same constant r (depending on Π and Π_G but not on k or G) as in Theorem 3.1.*

Proof. If $k > r$, we answer No immediately as in Theorem 3.1. Otherwise, we generate all k-vertex graphs, test each of them for having property Π, and if so apply the subgraph isomorphism algorithm for graphs with property Π_G to G and the generated graph. There are $2^{O(r^2)}$ graphs to generate, testing for property Π takes time $t_\Pi(r)$ for each one, and testing for being an induced subgraph of G takes time $t_{\mathrm{sgi}}(n, r)$ for each one, so the time is as stated. □

In particular, these problems can be solved in linear time for planar graphs.

Theorem 3.3. *If both Π_G and Π belong to* AS*, or if both belong to* SA*, then the problem $P(G, \Pi_G, \Pi, k)$ is in* FPT*.*

Proof. We give a proof for the case when both Π_G and Π belong to AS. The proof for the other case is again symmetric under reversal of the roles of cliques and independent sets. For a graph $G \in \Pi_G$ that is large enough that $|V(G)| \geq R(k, i_{\Pi_G})$, it must be the case that G contains a clique C of size k, for it cannot contain an independent set of size i_{Π_G} without violating the assumption that it belongs to Π_G. Because Π is assumed to be in AS, it contains all cliques, so this k-vertex clique belongs to Π. Therefore, for graphs with this many vertices, it is safe to answer Yes. There is a small subtlety here, in that we do not know an efficient method to calculate $R(k, i_{\Pi_G})$, and an inefficient method would unnecessarily increase the dependence of our time bounds on the parameter k. However, we can use the inequality

$$R(k, i_{\Pi_G}) \leq \binom{k + i_{\Pi_G} - 2}{k - 1}$$

to get a bound on this number that is easier to calculate. Our algorithm can simply test whether $|V| \geq \binom{k+i_{\Pi_G}-2}{k-1}$, and if so we return Yes without doing any searching.

If $|V(G)| < \binom{k+i_{\Pi_G}-2}{k-1}$, then constructing and checking all induced subgraphs of G of size k

to detect whether there exists such a subgraph belonging to Π takes time

$$\binom{k + i_{\Pi_G} - 2}{k-1}^k \left(O(k^2) + t_\Pi(k)\right),$$

a time complexity that is bounded by a function of k but independent of n. As the times for both cases are of the appropriate form, the problem $P(G, \Pi_G, \Pi, k)$ is in FPT. $\square$

The following corollaries can be directly obtained from Theorem 3.1 and Theorem 3.3.

Corollary 3.1. *If Π_G is the property of being co-bipartite and Π is the property of being a forest, planar, bipartite or triangle-free (or vice versa), then the problem $P(G, \Pi_G, \Pi, k)$ is solvable in polynomial time.*

Corollary 3.2. *If Π_G and Π are the properties of being planar, bipartite or triangle-free, then the problem $P(G, \Pi_G, \Pi, k)$ is FPT.*

For completeness, we state the following (trivial) theorem:

Theorem 3.4. *If $\Pi_G \in \mathsf{SS}$, then the problem $P(G, \Pi_G, \Pi, k)$ is solvable in polynomial time.*

Proof. We have $|V(G)| < R(c_{\Pi_G}, i_{\Pi_G})$, because otherwise G has either a clique of size c_{Π_G} or a trivial graph of size i_{Π_G}, a contradiction. Because $V(G)$ is bounded, there are only finitely many valid inputs to the problem $P(G, \Pi_G, \Pi, k)$ and we can precompute the solutions to each one. $\square$

Table 3.1 briefly summarizes the results of Theorems 3.1, 3.3 and 3.4.

	$\Pi \in \mathsf{SA}$	$\Pi \in \mathsf{AS}$
$\Pi_G \in \mathsf{AS}$	If $k < R(c_\Pi, i_{\Pi_G})$ check all induced subgraphs of size k, otherwise return No	If $\lvert V(G)\rvert < \binom{k+i_{\Pi_G}-2}{k-1}$ check all induced subgraphs of size k, otherwise return Yes
$\Pi_G \in \mathsf{SA}$	If $\lvert V(G)\rvert < \binom{k+c_{\Pi_G}-2}{k-1}$ check all induced subgraphs of size k, otherwise return Yes	If $k < R(c_{\Pi_G}, i_\Pi)$ check all induced subgraphs of size k, otherwise return No
$\Pi_G \in \mathsf{SS}$	$\lvert V(G)\rvert < R(c_{\Pi_G}, i_{\Pi_G})$, precompute all possible inputs	

Table 3.1: Summary of Theorems 3.1, 3.3 and 3.4.

3.4 Hardness from strong products

In this section, we prove some hardness results for the problem $P(G, \Pi_G, \Pi, k)$, when $\Pi_G \in \mathsf{AA}$ and $\Pi \in \mathsf{SA}$.

3.4.1 Hardness from strong products with cliques

To formulate the first of these results in full generality, we need some definitions. The *strong product* $G \boxtimes H$ is defined as a graph whose vertex set $V(G) \times V(H)$ consists of the ordered pairs of a vertex in G and a vertex in H, with two of these ordered pairs (u, v) and (u', v') adjacent if u and u' are adjacent or equal, and v and v' are adjacent or equal. In particular, the strong product with a complete graph, $G \boxtimes K_i$, can be thought of as making i copies of each vertex in G, with two copies of the same vertex always adjacent, and with adjacency between copies of different vertices remaining the same as in G. We use the notation $n \cdot K_i$ to denote the disjoint union of n copies of an i-vertex complete graph; this is the strong product of an n-vertex independent set with an i-vertex clique.

Observation 3.1. *Given a graph G on n vertices, there exists an independent set of G of size at least $n/\chi(G)$.*

Namely, the large independent set of the observation can be chosen as the largest color class of any optimal coloring of G.

Theorem 3.5. *Let $\Pi_G \in \mathsf{AA}$ be a hereditary property which is closed under strong products with complete graphs, and let $\Pi \in \mathsf{SA}$ be a hereditary property such that, for all n, the graph $n \cdot K_{\chi(\Pi)}$ belongs to Π. Then, the problem $P(G, \Pi_G, \Pi, k)$ is as hard as $P(G, \Pi_G, \mathsf{IS}, k)$.*

Proof. We describe a polynomial-time parameterized reduction from instances of $P(G, \Pi_G, \mathsf{IS}, k)$ to equivalent instance of $P(G, \Pi_G, \Pi, k')$, where k' depends only on k (and not on G). The reduction transforms the graph G of the instance into a new graph $G' = G \boxtimes K_{\chi(\Pi)}$, and transforms the parameter k into a new parameter value $k' = k \cdot \chi(\Pi)$.

As we have assumed that Π_G is closed under strong products with complete graphs, it follows that $G' \in \Pi_G$, so the reduction produces a valid instance of $P(G, \Pi_G, \Pi, k')$. To show that this instance is equivalent to the starting instance, we show that G has an independent set of size k if and only if G' has an induced subgraph of size k' belonging to Π.

($\Rightarrow$) Let I be an independent set of G of size k, and let $X = I \boxtimes K_{\chi(\Pi)}$ be the subgraph of G' induced by the set of all copies of vertices in I. Then $|V(X)| = k'$ and, as a graph of the form $k \cdot K_{\chi(\Pi)}$, X belongs to Π by assumption.

($\Leftarrow$) Let $H \in \Pi$ be an induced subgraph of G' of size k'. By Observation 3.1, it has an independent set I' of size $k'/\chi(\Pi) \geq k$. This independent set can include at most one copy of each vertex in G, so the set I of vertices in G whose copies are used in I' must also have size $\geq k$. Further, I is independent, for any edge between its vertices would be copied as an edge in G', contradicting the assumption that we have an independent set in G'. Therefore, I is an independent set of size $\geq k$ in G, as desired.

□

The families of unit-disk graphs, C_4-free graphs, and $K_{1,4}$-free graphs all belong to AA, and are closed under strong products with complete graphs. Finding independent sets is also known to be complete for unit-disk graphs [94, 95], C_4-free graphs [16], and $K_{1,4}$-free graphs [78]. Moreover, the families of planar graphs and of bipartite graphs both have the property that $n \cdot K_{\chi(\Pi)} \in \Pi$. For instance, in planar graphs, the graph $n \cdot K_{\chi(\Pi)}$ consists of n disjoint copies of K_4, a planar graph, and forming disjoint unions preserves planarity. Therefore, we have the following corollary:

Corollary 3.3. *If Π_G is the property of being (a) unit-disk, (b) C_4-free, or (c) $K_{1,4}$-free , and Π is the property of being either planar or bipartite, then the problem $P(G, \Pi_G, \Pi, k)$ is* $\mathsf{W}[1]$*-complete.*

3.4.2 Hardness from joins with cliques

The *join* of two graphs $G + H$ is a graph formed from the disjoint union of G and H by adding edges from each vertex of G to each vertex of H. The reduction that we consider in this section involves the join with a disjoint union of cliques, $G + t \cdot K_c$. That is, starting from G we add t cliques of size c, with each vertex in G connected to all vertices in these cliques.

Observation 3.2. *Given a graph G and two positive integers t and c, the maximum clique size of $G + t \cdot K_c$ is $\omega(G) + c$.*

Theorem 3.6. *Let $\Pi_G \in \mathsf{AA}$ be a hereditary property which is closed under joins with disjoint unions of cliques, and $\Pi \in \mathsf{SA}$ be a hereditary property which includes all subgraphs $I + n \cdot K_{\omega(\Pi)-1}$ for an independent set I and positive integer n. Then the problem $P(G, \Pi_G, \Pi, k)$ is as hard as $P(G, \Pi_G, \mathsf{IS}, k)$.*

Proof. We first construct a new graph $G' = G + r \cdot K_c$, where $r = R(\omega(\Pi) + 1, k)$ and $c = \omega(\Pi) - 1$, and a new parameter value $k' = k + rc$. By the assumption that Π_G is closed under joins with disjoint unions of cliques, $G' \in \Pi_G$. Now, we show that G has an independent set of size k if and only if G' has an induced subgraph of size k' belonging to Π.

($\Rightarrow$) Let I be an independent set of G of size k. Consider the induced subgraph $I + r \cdot K_c$ of G', formed by including all vertices that were added to G. This subgraph has size $k' = k + rc$, and by assumption it belongs to Π.

($\Leftarrow$) Let $H \in \Pi$ be an induced subgraph of G' of size k'. The vertices of H can be partitioned into two sets $S_1 \subset V(G)$ and $S_2 \subset r \cdot K_c$. The following two cases can occur:

- If S_1 is not an independent set, let uv be an edge in S_1. Then S_2 must have at most $c - 1$ vertices in each clique of $r \cdot K_c$, for if it contained all c vertices of one of these cliques, then these c vertices together with u and v would form a clique of size $\omega(\Pi) + 1$, which is disallowed in Π. Therefore, S_2 has at most $r(c-1)$ vertices, and to obtain total size k', S_1 must have at least $k + r$ vertices. By the definition of r and by Ramsey's theorem, S_1 has either a clique of size $\omega(\Pi) + 1$ (again, an impossibility) or an independent set of size k, as desired.
- If S_1 is an independent set, we observe that, even if S_2 includes all of the vertices added to G to form G', it has only rc vertices. Therefore, to obtain total size k', S_1 must have at least k vertices, and contains an independent set of size k, as desired.

$\square$

There are many families Π_G that meet the requirements on Π_G in this theorem, but do not meet the requirements of Theorem 3.5: this will be true, for instance, when the forbidden subgraphs of Π_G do not include disjoint unions of cliques, and are co-connected (so they cannot be formed by joins, which produce co-disconnected graphs) but at least one of these graphs contains two adjacent twin vertices (with the same neighbors other than each other). The requirement on Π in this theorem is met, for instance, by the family Π of bipartite graphs. In this case, $\omega(\Pi) = 2$, so the graphs $I + n \cdot K_{\omega(\Pi)-1}$ are just complete bipartite graphs, which are of course bipartite.

As an example, finding k-independent sets in $\overline{K_{1,3}}$-free graphs (the complements of claw-free graphs) is known to be NP-complete, from the completeness of the same problem in triangle-free graphs [101]. Theorem 3.6 then shows that finding k-vertex bipartite induced subgraphs of $\overline{K_{1,3}}$-free graphs is also NP-complete. However, we cannot use this method to prove parameterized hardness for this example, because the k-independent set problem in $\overline{K_{1,3}}$-free graphs can be solved in FPT by applying an FPT algorithm for $(k-1)$-independent sets in triangle-free graphs [43] to the sets of non-neighbors of all vertices.

3.5 Conclusion

We have further narrowed down the parameterized complexity of the problem $P(G, \Pi, k)$ for the case when it is $\mathsf{W}[1]$-complete. In particular, restricting the input graph G to belong to a hereditary graph class Π_G helps us to settle parameterized complexity of numerous graph classes circumventing long and tedious reduction proofs. It remains an open problem to determine the parameterized complexity of the problem $P(G, \Pi_G, \Pi, k)$ when $\Pi_G \in \mathsf{AA}$ without any restrictions. It would be also interesting to investigate this problem under other graph parameters beyond the size of the solution.

Chapter 4

Simplifying Activity-on-Edge Graphs[1]

4.1 Introduction

The *critical path method* is used in project modeling to describe the tasks of a project, along with the dependencies among the tasks; it was originally developed as PERT by the United States Navy in the 1950s [93]. A dependency graph is used to identify bottlenecks, and in particular to find the longest path among a sequence of tasks, where each task has a required length of time to complete (this is known as the *critical path*).

In this chapter we consider a phase in planning a given project in which we do not yet know the time lengths of each task. We are interested in the problem of visualizing an abstract timeline of the potential critical paths (i.e., paths that could be critical depending on the lengths of the tasks) of the project, represented abstractly as a partially ordered set of tasks. The most common method of visualizing partially ordered sets, as an *activity-on-node graph* (a transitively reduced directed acyclic graph with a vertex for each task) is unsuitable for this aim, because it represents each task as a point instead of an object that can extend over

[1]The material in this chapter is from a published work with Eppstein and Frishberg [53].

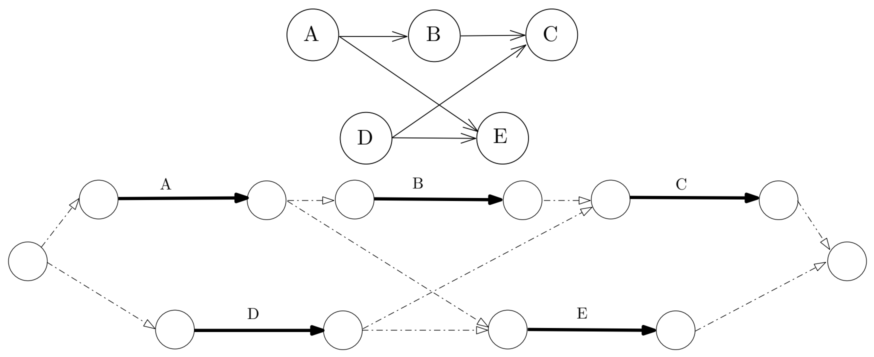

Figure 4.1: An activity-on-node graph, above, and its naively expanded activity-on-edge graph, below, with solid arrows as task edges and empty arrows as unlabeled edges.

a span of time in a timeline. To resolve this issue, we choose to represent each task as an edge in a directed acyclic graph. In this framework, the endpoints of the task edges have a natural interpretation, as the *milestones* of the project to be scheduled. Additional *unlabeled edges* do not represent tasks to be performed within the project, but constrain certain pairs of milestones to occur in a certain chronological order. The resulting *activity-on-edge graph* can then be drawn in standard upward graph drawing style [44, 12, 67, 66, 1]. Alternatively, once the lengths of the tasks are known and the project has been scheduled, this graph can be drawn in leveled style [82, 76], where the level of each milestone vertex represents the time at which it is scheduled.

It is straightforward to expand an activity-on-node graph into an activity-on-edge graph by expanding each task vertex of the activity-on-node graph into a pair of milestone vertices connected by a task edge, with the starting milestone of each task retaining all of the incoming unlabeled edges of the activity-on-node graph and the ending milestone retaining all of the outgoing edges. It is convenient to add two more milestones at the start and end of the project, connected respectively to all milestones with no incoming edges and from all milestones with no outgoing edges. The size of the resulting activity-on-edge graph is linear in the size of the activity-on-node graph. An example of such a transformation is depicted in Figure 4.1.

However, the graphs that result from this naive expansion are not minimal. Often, one can

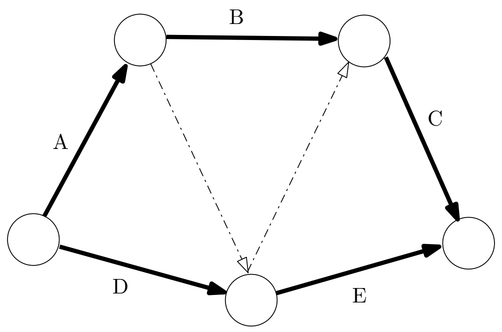

Figure 4.2: A simplification of the graph from Figure 4.1.

merge some pairs of milestones (for instance the ending milestone of one task and the starting milestone of another task) to produce a simpler activity-on-edge graph (such as the one for the same schedule in Figure 4.2). Despite having fewer milestones, this simpler graph can be equivalent to the original, in the sense that its potential critical paths (maximal sequences of tasks that belong to a single path in the graph) are the same. By being simpler, this merged graph should aid in the visualization of project schedules. In this chapter we formulate and provide an $O(mn^2)$-time algorithm (where n is the number of milestones and m is the number of unlabeled edges) for the problem of optimal simplification of activity-on-edge graphs.

4.1.1 New Results

We describe a polynomial-time algorithm that, given an activity-on-edge graph (i.e., a directed acyclic graph with a subset of its edges labeled as tasks), produces a directed acyclic graph that preserves the potential critical paths of the graph and has the minimum possible number of vertices among all critical-path-preserving graphs for the given input. Our algorithm is agnostic about the weights of the tasks. In more general terms, the resulting graph has the following properties:

- The task edges in the given graph correspond one-to-one with the task edges in the new graph.

- The new graph has the same dependency (reachability) relation among task edges as the original graph.
- The new graph has the same potential critical paths as the original graph.
- The number of vertices of the graph is minimized among all graphs with the first three properties.

Our algorithm repeatedly applies a set of local reduction rules, each of which either merges a pair of adjacent vertices or removes an unlabeled edge, in arbitrary order. When no rule can be applied, the algorithm outputs the resulting graph.

We devote the rest of this section to related work and then describe the preliminaries in Section 4.2. We then present the algorithm in Section 4.3 and show in Section 4.4 that its output preserves the potential critical paths of the input, and in Section 4.5 that it has the minimum possible number of vertices. We also show that the output is independent of the order in which the rules are applied. We discuss the running time in Section 4.6 and conclude with Section 4.7.

4.1.2 Related work

Constructing clear and aesthetically pleasing drawings of directed acyclic graphs is an old and well-established task in graph drawing, with many publications [104, 12, 11, 77]. The work in this line that is most closely relevant for our work involves upward drawings of unweighted directed acyclic graphs [44, 67, 66, 1] or leveled drawings of directed acyclic graphs that have been given a level assignment [82, 76] (an assignment of a y-coordinate to each vertex, for instance representing its height on a timeline).

Although multiple prior publications use activity-on-edge graphs [87, 31, 5, 81] and even consider graph drawing methods specialized for these graphs [106], we have been unable to locate prior work on their simplification. This problem is related to a standard computational problem, the construction of the transitive reduction of a directed acyclic graph or equivalently the covering graph of a partially ordered set [3]. We note in addition our prior work on augmenting partially ordered sets with additional elements (preserving the partial order on the given elements) in order to draw the augmented partial order as an upward planar graph with a minimum number of added vertices [61].

The PERT method may additionally involve the notion of "float", in which a given task may be delayed some amount of time (depending on the task) without any effect on the overall time of the project [7, 80]. We do not consider constraints of this form in the present work, although the unlabeled edges of our output can in some sense be seen as serving a similar purpose.

4.2 Preliminaries

We first define an activity-on-edge graph. The graph can be a multigraph to allow tasks that can be completed in parallel to share both a start and end milestone when possible.

Definition 4.1. *An* activity-on-edge graph (AOE) *is a directed acyclic multigraph $G = (V, E)$, where a subset of the edges of E, denoted $\mathcal{T}$, are labeled as* task edges*. The labels, denoting tasks, are distinct, and we identify each edge in $\mathcal{T}$ with its label.*

Definition 4.2. *Given an AOE G with tasks $\mathcal{T}$, for all $T \in \mathcal{T}$, let $\mathrm{St}_G(T)$ be the start vertex of T, and let $\mathrm{End}_G(T)$ be the end vertex of T.*

When the considered graph is clear from context, we omit the subscript G and write $\mathrm{St}(T)$ and $\mathrm{End}(T)$. It may be that $\mathrm{St}(T) = \mathrm{St}(T')$, or $\mathrm{End}(T) = \mathrm{End}(T')$, or $\mathrm{End}(T) = \mathrm{St}(T')$ with $T \neq T'$.

To define potential critical paths formally, we introduce the following notation.

Definition 4.3. *Given an AOE G with tasks $\mathcal{T}$, for all $T, T' \in \mathcal{T}$ with $T \neq T'$, say that T* has a path to T' *in G if there exists a path from* $\mathrm{End}(T)$ *to* $\mathrm{St}(T')$, *or if* $\mathrm{End}(T) = \mathrm{St}(T')$, *and write $T \rightsquigarrow_G T'$.*

Definition 4.4. *Given an AOE G with tasks $\mathcal{T}$, a* potential critical path *is a sequence of tasks $P = (T_1, \ldots, T_k)$, where for all $i = 1, \ldots, k-1$, $T_i \rightsquigarrow_G T_{i+1}$, and where P is not a subsequence of any other sequence with this property.*

Our algorithm will apply a set of transformation rules to an input AOE of a *canonical* form.

Definition 4.5. *A* canonical AOE *is an AOE which is naively expanded from an activity-on-node graph.*

Every AOE G can be transformed into a canonical AOE with the same reachability relation on its tasks. First, we start by computing the reachability relation of the tasks. The transitive closure of the resulting reachability matrix gives an activity-on-node graph (which is quadratic, in the worst case, in the size of the original AOE). Then, this activity-on-node graph can be converted to a canonical AOE as described in Section 4.1.

Definition 4.6. *Two AOE graphs G and H are* equivalent, *i.e. $G \equiv H$, if G and H have the same set of tasks—i.e., there is a label-preserving bijection between the task edges of G and those of H—and, with respect to this bijection, G and H have the same set of potential critical paths.*

Definition 4.7. *An AOE G is* optimal *if G minimizes the number of vertices for its equivalence class: i.e., if for every AOE $H \equiv G$, $|V(H)| \geq |V(G)|$.*

We now formally define our problem.

Problem 1. *Given a canonical AOE G, find some optimal AOE H with $H \equiv G$.*

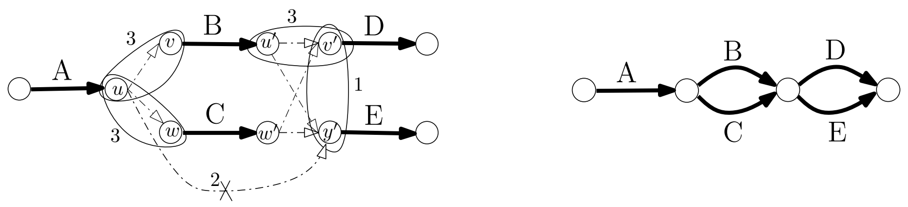

Figure 4.3: On the left, an AOE in which each of rules 1-3 can be applied, and on the right, the corresponding graph output by the algorithm.

4.3 Simplification Rules

Our algorithm takes a canonical AOE and greedily applies a set of rules until no more rules can be applied. Given an AOE $G = (V, E)$ and given two distinct vertices $u, v \in V$, the simplification rules used by our algorithm are:

1. if u and v have no outgoing task edges and have precisely the same outgoing neighbors, merge u and v. Symmetrically, if u and v have no incoming task edges and have precisely the same incoming neighbors, merge u and v.
2. If u has an unlabeled edge to v, and u has another path to v, remove the edge (u, v).
3. If u has an unlabeled edge to v and the following conditions are satisfied, merge u and v:
   - rule 2 is not applicable to the edge (u, v).
   - if u has an outgoing task, then v has no incoming edge other than (u, v).
   - if v has an incoming task, then u has no outgoing edge other than (u, v).
   - every incoming neighbor of v has a path to every outgoing neighbor of u.

Figure 4.3 depicts an AOE graph and the graph output by the algorithm after applying all possible rules. Vertices u and v can be merged by rule 3, since there is no other path from u to v to apply rule 2 (satisfying the first condition in the application of rule 3), u and v

have no outgoing and no incoming task, respectively, and v has no incoming neighbor other than u. Therefore, the conditions of rule 3 are true (the second and third hold vacuously). Further, vertices u' and v' can be merged since the first three conditions for applying rule 3 are satisfied and there exists a path from w' to y', satisfying the last condition.

It will be convenient for the proofs in Section 4.5 to give a name to the output of the algorithm:

Definition 4.8. *An* output AOE, *denoted $\mathcal{A}$, is any AOE obtained from a canonical AOE G by a sequence of applications of rules 1, 2, and 3, to which none of these rules can still be applied.*

We will show (Theorem 4.2) that $\mathcal{A}$ does not depend on the order in which the rules are applied.

4.4 Correctness

In this section we prove the correctness of our algorithm (its output graph is equivalent to its input graph).

We begin with preserving potential critical paths. We show that the rules never change the existence or nonexistence of a path from one task to another, and that this implies preservation of potential critical paths.

Lemma 4.1. *Given two AOEs G and H with the same set of tasks $\mathcal{T}$, G and H have the same reachability relation $\rightsquigarrow$ on the tasks if and only if $G \equiv H$.*

Proof. Trivially, we have $T \rightsquigarrow_G T'$ (or $T \rightsquigarrow_H T'$) if and only if T is earlier than T' in some potential critical path of G (or H). Therefore, preservation of potential critical paths is equivalent to preservation of the reachability relation. □

Lemma 4.2. *The output of the algorithm is equivalent to its input.*

Proof. We show the invariant that given tasks T and T', $T \rightsquigarrow T'$ at a given iteration of the algorithm if and only if $T \rightsquigarrow T'$ at the next iteration. From this, and from the fact that the rules never change the set of tasks, it follows that the output of the algorithm has the same reachability relation on its tasks as the input, and then the lemma follows from Lemma 4.1.

The invariant is true because merging a pair of vertices (rules 1 and 3) never disconnects a path, and no edge is ever removed (by rule 2) between two vertices unless another path exists between the two vertices. In particular, the end vertex of T still has a path to the start vertex of T' after the application of any of the rules.

For the other direction, removing an edge never introduces a new path. Furthermore, if vertices u and v are merged by applying rule 1, and if some vertex w has a path to some vertex z through the newly merged uv, then the condition of rule 1 ensures that w has a path, through u or v, to z before the merge. Similarly, suppose u and v are merged by applying rule 3. Then if w has a path to z through uv, then (abusing notation) either $w \rightsquigarrow u$ and $v \rightsquigarrow z$ before the merge, so $w \rightsquigarrow z$ (via the edge (u, v)), or for some incoming neighbor x of v and outgoing neighbor y of u, $w \rightsquigarrow x$ and $y \rightsquigarrow z$. In this case, by the conditions of the rule, $w \rightsquigarrow z$ before the merge. □

Lemma 4.3. *Any intermediate graph that results from applying rules of the algorithm to an input canonical AOE graph, is acyclic.*

Proof. Given Definition 4.1 and Definition 4.5, the canonical AOE input G is acyclic. Now we show none of the rules can create a cycle after being applied to an intermediate acyclic graph G'. This is obvious for rule 2 as it removes edges. Suppose for a contradiction that merging vertices u and v creates a cycle. The cycle must involve the new vertex resulting from the merge. For rule 1, this implies the existence of a cycle in G' either from u or v to itself which is a contradiction. For rule 3, it implies the existence of a cycle in G' including the unlabeled edge (u, v) or a cycle including an incoming neighbor of v and an outgoing neighbor of u, which is a contradiction. □

Corollary 4.1. *Any graph $\mathcal{A}$ output by the algorithm is acyclic.*

4.5 Optimality

In this section we prove the optimality of our algorithm: it uses as few vertices as possible. Let $\mathcal{A}$ be any output AOE. Let *Opt* be any optimal AOE such that $\mathcal{A} \equiv$ *Opt*. Our proof relies on an injective mapping from the vertices of $\mathcal{A}$ to the vertices of *Opt*. The existence of this mapping shows that $\mathcal{A}$ has at most as many vertices as *Opt*, and therefore has the optimal number of vertices. Once we have identified the vertices of $\mathcal{A}$ with the vertices of *Opt* in this way, we show that, for a given input, any two graphs output by the algorithm (but not necessarily *Opt*) must have the same unlabeled edges. Since the task edges are determined, and since the injective mapping to *Opt* determines the vertices, determining the unlabeled edges implies the order-independence of our algorithm's choice of simplification rules.

Before defining the mapping between $\mathcal{A}$ and *Opt*, we establish some facts about the structure of $\mathcal{A}$.

Lemma 4.4. *For every unlabeled edge (u, v) in any output AOE $\mathcal{A}$, there exist tasks T and T' such that $u = \mathrm{End}(T)$ and $v = \mathrm{St}(T')$.*

Proof. By Definition 4.8, $\mathcal{A}$ is produced by the algorithm from some canonical AOE G. This property holds for G by Definition 4.5. As every rule of the algorithm either removes an unlabeled edge or merges two vertices, and never creates a new edge or vertex, the proof is complete. □

Corollary 4.2. *Every vertex in an output AOE $\mathcal{A}$ has an incident task edge.*

We can now define a mapping from the vertices of $\mathcal{A}$ to those of *Opt*:

Definition 4.9. *Given an output AOE $\mathcal{A}$ with task set $\mathcal{T}$, and given an optimal AOE Opt with $\mathcal{A} \equiv Opt$, let $M : V(\mathcal{A}) \to V(Opt)$ be the following mapping: for every $v \in V(\mathcal{A})$:*

- *Let $M(v) = \mathrm{St}_{Opt}(T)$, for some $T \in \mathcal{T}$ for which $v = \mathrm{St}_{\mathcal{A}}(T)$, if such a task exists.*
- *Let $M(v) = \mathrm{End}_{Opt}(T)$, where $v = \mathrm{End}_{\mathcal{A}}(T)$, otherwise.*

As shown in Corollary 4.2, every vertex in $\mathcal{A}$ has an incident task edge, and by Definition 4.6, $\mathcal{A}$ and *Opt* have the same set of tasks. Therefore, this mapping is well-defined (up to its arbitrary choices of which task to use for each v). To prove that M is injective, we will use the fact that since $\mathcal{A} \equiv Opt$, $\mathcal{A}$ and *Opt* have the same reachability relation (by Lemma 4.1 and Lemma 4.2).

The heart of the proof that M is injective lies in showing that if two tasks do not share a vertex in $\mathcal{A}$, the corresponding tasks also do not share the corresponding vertices in *Opt*. From this it follows that M cannot map distinct vertices in $\mathcal{A}$ to the same vertex in *Opt*.

Lemma 4.5. *Given an output AOE $\mathcal{A}$, and an optimal AOE $Opt \equiv \mathcal{A}$, with task set $\mathcal{T}$, let T and T' be two distinct tasks in $\mathcal{T}$. If $\mathrm{St}_{\mathcal{A}}(T) \neq \mathrm{St}_{\mathcal{A}}(T')$, then $\mathrm{St}_{Opt}(T) \neq \mathrm{St}_{Opt}(T')$. If $\mathrm{End}_{\mathcal{A}}(T) \neq \mathrm{End}_{\mathcal{A}}(T')$, then $\mathrm{End}_{Opt}(T) \neq \mathrm{End}_{Opt}(T')$.*

Proof. Suppose for a contradiction that $\mathrm{St}_{\mathcal{A}}(T) \neq \mathrm{St}_{\mathcal{A}}(T')$, but $\mathrm{St}_{Opt}(T) = \mathrm{St}_{Opt}(T')$ (the other case is symmetrical). Let $u = \mathrm{St}_{\mathcal{A}}(T)$ and $v = \mathrm{St}_{\mathcal{A}}(T')$. Consider the following (exhaustive) cases for u and v:

1. u and v have no incoming edges

2. u or v has an incoming unlabeled edge, but neither u nor v has an incoming task edge

3. u or v has an incoming task edge A

In case 1, applying rule 1 results in merging u and v. However, since $\mathcal{A}$ is the output of the algorithm, no rule can be applied to $\mathcal{A}$. This is a contradiction.

In case 2, u and v cannot have the same incoming neighbors or else rule 1 would apply. We may assume without loss of generality that there exist a vertex w and an unlabeled edge (w, u), such that the edge (w, v) does not exist. By Lemma 4.4, there exists a task A where $w = \mathrm{End}_{\mathcal{A}}(A)$. Since $A \rightsquigarrow_{\mathcal{A}} T$ and $\mathcal{A} \equiv \mathit{Opt}$ (by Lemma 4.2), then by Lemma 4.1, $A \rightsquigarrow_{\mathit{Opt}} T$, so $A \rightsquigarrow_{\mathit{Opt}} T'$, since $\mathrm{St}_{\mathit{Opt}}(T) = \mathrm{St}_{\mathit{Opt}}(T')$. Again by Lemma 4.1, $A \rightsquigarrow_{\mathcal{A}} T'$, so there is a path P from w to v. If $|P| = 1$, then this contradicts that (w, v) does not exist. Suppose $|P| > 1$. Then we show there exist some vertex $w' \neq w$ and an unlabeled edge (w', v). The following cases are exhaustive:

(a) P contains a path from u to v. As such a path to v exists and v has no incoming task edge, there exist a vertex w' and an unlabeled edge (w', v) $(w' \neq u)$, not belonging to P unless rule 3 can be applied to vertex v and its incoming neighbor in path P .

(b) P does not contain a path from u to v. As $|P| > 1$, an unlabeled edge (w', v) belonging to path P exists.

Given the existence of (w', v), by Lemma 4.4, there exists a task B where $w' = \mathrm{End}_{\mathcal{A}}(B)$. $B \rightsquigarrow_{\mathcal{A}} T'$, so by reasoning similar to the above, $B \rightsquigarrow_{\mathcal{A}} T$. Then, one can apply rule 2 and either remove edge (w', v) in case a or (w, u) in case b (Figure 4.4); this contradicts the definition of $\mathcal{A}$.

In case 3, we can assume without loss of generality that u has an incoming task A; consequently, $u = \mathrm{End}_{\mathcal{A}}(A)$. Then, by Lemma 4.1, we have $A \rightsquigarrow_{\mathit{Opt}} T'$ and thus $A \rightsquigarrow_{\mathcal{A}} T'$ via a path P.

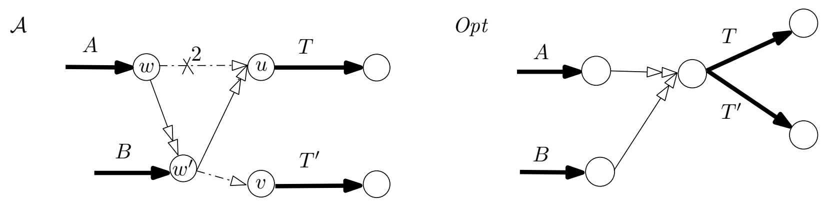

Figure 4.4: Lemma 4.5, case 2, subcase b (double arrows indicate paths).

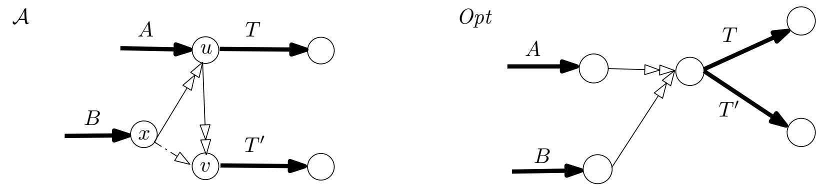

Figure 4.5: Lemma 4.5, case 3, subcase b.

Consider the following cases for P:

(a) P contains a task edge B

(b) P is a sequence of unlabeled edges

In case a, by Lemma 4.1 $B \rightsquigarrow_{Opt} T'$, and thus $B \rightsquigarrow_{Opt} T$, and therefore $B \rightsquigarrow_{\mathcal{A}} T$. This creates a cycle between u and $\mathrm{End}_{\mathcal{A}}(B)$, contradicting Corollary 4.1.

In case b, illustrated in Figure 4.5, since rule 3 cannot be applied (if it could, this would contradict the definition of $\mathcal{A}$), there exist a vertex x not on the path from u to v, and an edge (x, v) (a task edge or an unlabeled edge). Therefore, there exists a task B where either $v = \mathrm{End}_{\mathcal{A}}(B)$ or by Lemma 4.4, $x = \mathrm{End}_{\mathcal{A}}(B)$. Considering Opt and applying Lemma 4.1, $B \rightsquigarrow_{Opt} T$ so $B \rightsquigarrow_{\mathcal{A}} T$. This path either creates a cycle in $\mathcal{A}$ or allows for removing edge (x, v) by rule 2, which is a contradiction.

Thus if $\mathrm{St}_{\mathcal{A}}(T) \neq \mathrm{St}_{\mathcal{A}}(T')$, then $\mathrm{St}_{Opt}(T) \neq \mathrm{St}_{Opt}(T')$.

□

Lemma 4.6. *Given an output AOE $\mathcal{A}$, and an optimal AOE $Opt \equiv \mathcal{A}$, with task set $\mathcal{T}$, let T and T' be two distinct tasks in $\mathcal{T}$. If $\mathrm{End}_{\mathcal{A}}(T) \neq \mathrm{St}_{\mathcal{A}}(T')$, then $\mathrm{End}_{Opt}(T) \neq \mathrm{St}_{Opt}(T')$.*

Proof. The proof, which is in Appendix **??**, uses essentially the same approach as the proof of Lemma 4.5: supposing that the two vertices are the same, then using the fact that $\mathcal{A}$ and *Opt* have the same reachability relation on their tasks, and the definition of $\mathcal{A}$ as having no more rules to apply, to derive a contradiction.

□

There is one remaining technicality: we have defined an optimal AOE as being acyclic; the question arises whether one could reduce the number of vertices by allowing (unlabeled) cycles. However, this is not the case; it is easy to see that any unlabeled cycle can be merged into one vertex, reducing the number of vertices, without changing the reachability relation on the tasks.

We are ready to prove our main results.

Theorem 4.1. *Given a canonical AOE G, the algorithm produces an optimal AOE $Opt \equiv G$.*

Proof. Let $\mathcal{A}$ be the output AOE produced by the algorithm on G. Given any optimal AOE *Opt* and $\mathcal{A}$, the mapping M in Definition 4.9 is injective: suppose for a contradiction that u and v are distinct vertices in $\mathcal{A}$, and $w = M(u) = M(v)$. Then by the definition of M, either u, v, and w have the same incoming task, or u, v, and w have the same outgoing task, or there exist tasks T and T' such that (without loss of generality) $u = \mathrm{End}_{\mathcal{A}}(T), v = \mathrm{St}_{\mathcal{A}}(T')$, and $\mathrm{End}_{Opt}(T) = w = \mathrm{St}_{Opt}(T')$. By Lemmas 4.5 and 4.6, all three of these cases imply that $u = v$.

Therefore, $|V(Opt)| = |V(\mathcal{A})|$. Furthermore, $\mathcal{A} \equiv G$, by Lemma 4.2. The theorem follows. □

Theorem 4.2. *Given an input, the algorithm produces the same output regardless of the order in which the rules are applied.*

Proof. As stated in subsection 4.1.1, all task edges of an input canonical AOE G are present in any output of the algorithm and the mapping determines the vertices. Therefore, it suffices to show that any two graphs output by the algorithm have the same set of unlabeled edges. Suppose for a contradiction that $\mathcal{A}_1$ and $\mathcal{A}_2$ are two distinct outputs of the algorithm, resulting from applying different sequences of rules. By Theorem 4.1, the algorithm always produces an optimal AOE. Therefore, $|V_{\mathcal{A}_1}| = |V_{\mathcal{A}_2}| = |V_{Opt}|$. Since $\mathcal{A}_1 \neq \mathcal{A}_2$, there is an unlabeled edge (u, v) in $\mathcal{A}_1$ (without loss of generality) that is not in $\mathcal{A}_2$. By Lemma 4.4, there exist task edges T and T' such that $u = \text{End}_{\mathcal{A}_1}(T)$ and $v = \text{St}_{\mathcal{A}_1}(T')$. We have $T \rightsquigarrow_{\mathcal{A}_1} T'$. Since by Lemma 4.1 and Lemma 4.2, $\mathcal{A}_1$ and $\mathcal{A}_2$ both preserve the reachability relation of the tasks of G, we have $T \rightsquigarrow_{\mathcal{A}_2} T'$. Consider the cases for path P from T to T' in $\mathcal{A}_2$:

1. There exists a task A in P other than T and T'.
2. Path P is a sequence of unlabeled edges.

In case 1, we have $T \rightsquigarrow_{\mathcal{A}_2} A \rightsquigarrow_{\mathcal{A}_2} T'$ and therefore, $T \rightsquigarrow_{\mathcal{A}_1} A \rightsquigarrow_{\mathcal{A}_1} T'$. Then by rule 2, one can remove the edge (u, v), which contradicts the definition of $\mathcal{A}_1$.

In case 2, the length of P is at least two, and P contains a vertex w. By Lemma 4.4, there exist tasks A and B where $w = \text{End}_{\mathcal{A}_2}(A) = \text{St}_{\mathcal{A}_2}(B)$. Now, since $\mathcal{A}_1 \equiv \mathcal{A}_2$, both graphs are optimal, and both graphs are outputs of the algorithm, Lemma 4.6 implies that $\text{End}_{\mathcal{A}_1}(A) = \text{St}_{\mathcal{A}_1}(B)$. Call this vertex x. Then there exists a path from u to v, through x, by Lemma 4.1, and one can remove the edge (u, v) by rule 2. This contradicts the definition of $\mathcal{A}_1$. □

It is tempting to imagine that Theorem 4.2 implies uniqueness of the optimal AOE. However, this is not the case: the unlabeled edges of an optimal AOE are not determined by our bijection. Figure 4.6 shows an optimal AOE graph that our algorithm cannot produce (because it violates Lemma 4.4). One way to see the optimality is to expand the graph naively

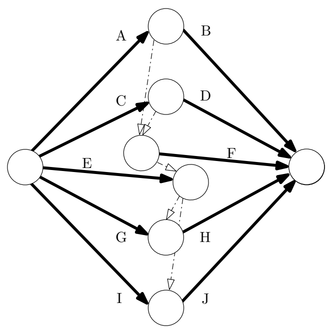

Figure 4.6: An optimal AOE graph that the algorithm cannot produce.

into a canonical AOE, apply the algorithm, and verify that the resulting number of vertices is the same.

4.6 Analysis

Let n be the number of vertices in a canonical AOE (which is linear in the number of tasks), and m the number of unlabeled edges. (The number of task edges is $O(m)$.) There are at most $O(n + m)$ iterations in the algorithm, because each iteration either merges two vertices or removes an edge, by applying one of the three rules. This requires finding an edge to remove ($O(m)$ potential edges) or two vertices to merge ($O(n^2)$ potential pairs), then performing the merge or the removal. Intuitively, our algorithm runs in polynomial time as it takes polynomial time to find and apply a rule.

We provide a faster implementation of our algorithm than the naive approach. The algorithm transforms a canonical AOE graph G into an optimal AOE graph by applying rules 1, 2 or 3. For simplicity, we label the vertices $1, \ldots, n$. At each iteration, compute a reachability matrix M for the current graph. $M[u][v]$ indicates whether there exist zero, one, or more than one paths from u to v. In order to compute M, for all u and v initialize $M[u][v] = 1$ if the edge (u, v) exists. Then sort the vertices in topological order (such an ordering exists according to Lemma 4.3). For each vertex v in this order, and for each vertex u, set $M[u][v]$

Algorithm 1: the proposed transformation algorithm.

```
Data: Canonical AOE $G$
Result: Optimal AOE $Opt$
1 while true do
2     Initialize and compute the reachability matrix $M$;
3     Remove, by rule 2, all unlabeled edges $(u, v)$ where $M[u][v] = 2$;
4     if rule 1 applies then
5         apply rule 1;
6     else if rule 3 applies then
7         apply rule 3;
8     else
9         return the graph;
```

to $\min(2, \sum_{w \in W}(M[u][w]))$, where W is the set of all vertices w such that either $w = v$ or there exists an edge (w, v). This procedure takes $O(nm)$ time. Algorithm 1 provides a summary.

Given the reachability matrix, an unlabeled edge (u, v) is removed by rule 2 in $O(1)$ time, if $M[u][v] \geq 2$. Therefore, checking rule 2 for all edges takes $O(m)$ time.

Without loss of generality, for rule 1, we only consider merges of pairs of vertices with the same outgoing neighbors. This requires, for each vertex u with no outgoing task edge, a sorted list of outgoing neighbors (S[u]). To obtain such lists for all vertices, list unlabeled edges as pairs of vertices and sort all the pairs with two bucket sorts: first over the first elements of the pairs, then over the second elements. Breaking the sorted list into chunks of pairs with the same first element (say u), gives the outgoing neighbors of u, in the second elements of the pairs, in a numerically sorted order. This takes $O(m)$ time. Then find pairs of vertices to merge, if any exist: first, bucket sort vertices based on their out-degree. Vertices in different buckets cannot be merged by rule 1. For each bucket b containing vertices with degree d ($0 \leq d < n$), call MergeDetection(b, d):

The vertices in each resulting bucket have the same outgoing neighbors and can be merged by rule 1. As each vertex with degree d appears in one bucket in each of $d + 1$ iterations,

```
Function MergeDetection(bucket $a, i$):
    if $i = 0$ then
        return bucket $a$
    else
        bucket sort vertices $v$ of $a$ based on S$[v][i]$
        foreach newly created bucket $a'$ do
            MergeDetection($a', i-1$)
```

this sort takes $O(\sum_v(deg(v))) = O(m)$ time. Upon merging vertices u and v, name the new vertex $\min(u, v)$.

To check rule 3, for each vertex v, compute $I(v)$: the intersection of the reachable sets of the incoming neighbors of v. This takes $O(mn)$ time.

Consider only those unlabeled edges (u, v) that meet the preconditions of rule 3 concerning the existence of outgoing and incoming tasks of u and v respectively. Test whether the last point in rule 3 applies to edge (u, v) by testing in $O(n)$ time whether all outgoing neighbors of u are in $I(v)$.

Computing the reachability matrix takes $O(mn)$ time, and using this matrix to check for rule 2 takes $O(m)$ time per iteration. Checking for rule 1 or 3 takes $O(mn)$ time per iteration. Further, the outer loop in Algorithm 1 runs at most n times as it either merges two vertices or returns the output. This gives a total complexity of $O(mn^2)$ for our algorithm.

4.7 Conclusion

Our algorithm reduces the visual complexity of an activity-on-edge graph, making it easier to understand bottlenecks in a project. The algorithm repeatedly applies simple rules and therefore can be implemented easily. We have shown that the algorithm runs in $O(mn^2)$ time. One question for future work is whether this analysis is tight. Another question is whether some other algorithm could achieve an optimal graph more efficiently.

Furthermore, one can measure the complexity of a graph in other ways. One question for future work is whether one can minimize the number of edges in an AOE graph in polynomial time. Another question is whether one can, in polynomial time, convert an AOE graph G into a graph that (i) has the same potential critical paths as G, and (ii) has a plane drawing with fewer edge crossings than all other graphs satisfying (i). It would also be interesting to implement this algorithm and run it on realistic graphs arising in project planning, and to evaluate the visual complexity of the resulting graphs in terms of the measures described above.

Chapter 5

Conclusion

In Chapter 2, we have developed two parameterized algorithms for representing k-leaf roots and ultimately recognizing k-leaf powers with bounded degeneracy. We leave the problems of finding a better dependence on the parameters and recognizing the parameterized complexity of k-leaf powers for graphs of bounded clique-width as open problems.

Following the work on the paradigm of parameterized complexity, we have studied the parameterized complexity of finding subgraphs with hereditary properties on graphs belonging to hereditary graph classes and established a framework which settles the parameterized complexity of numerous graph classes by checking some characteristics of the underlying hereditary properties.

Further, we have proposed as $O(mn^2)$-time algorithm to simplify activity-on-edge graphs used for visualization of project schedules. Given an input activity-on-edge graph, the output of the algorithm is an activity-on-edge graph with the same critical paths as the input with the minimum possible number of vertices. We leave the problem of optimizing the number of unlabeled edges as an interesting open problem.

Bibliography

[1] S. Abbasi, P. Healy, and A. Rextin. Improving the running time of embedded upward planarity testing. *Information Processing Letters*, 110(7):274–278, 2010.

[2] R. Afshar, M. T. Goodrich, P. Matias, and M. C. Osegueda. Reconstructing binary trees in parallel. In C. Scheideler and M. Spear, editors, *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, pages 491–492. ACM, 2020.

[3] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM J. Comput.*, 1(2):131–137, 1972.

[4] V. E. Alekseev. The effect of local constraints on the complexity of determination of the graph independence number. In *Combinatorial-algebraic methods in applied mathematics*, pages 3–13. Gor\cprime kov. Gos. Univ., Gorki, 1982.

[5] C. Alexander, D. Reese, and J. Harden. Near-critical path analysis of program activity graphs. In *Proceedings of International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 308–317. IEEE, 1994.

[6] N. Alon and S. Gutner. Linear time algorithms for finding a dominating set of fixed size in degenerated graphs. *Algorithmica*, 54(4):544, 2009.

[7] D. Arditi and T. Pattanakitchamroon. Selecting a delay analysis method in resolving construction claims. *International J. Project Management*, 24(2):145–155, 2006.

[8] S. Arnborg, J. Lagergren, and D. Seese. Easy problems for tree-decomposable graphs. *J. Algorithms*, 12(2):308–340, 1991.

[9] M. Bannach and S. Berndt. Practical access to dynamic programming on tree decompositions. In Y. Azar, H. Bast, and G. Herman, editors, *26th Annual European Symposium on Algorithms (ESA 2018)*, volume 112 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:13, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[10] M. J. Bannister and D. Eppstein. Crossing minimization for 1-page and 2-page drawings of graphs with bounded treewidth. *J. Graph Algorithms Appl.*, 22(4):577–606, 2018.

[11] O. Bastert and C. Matuszewski. Layered drawings of digraphs. In M. Kaufmann and D. Wagner, editors, *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes in Computer Science*, pages 87–120. Springer-Verlag, 2001.

[12] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice-Hall, 1999.

[13] U. Bertelè and F. Brioschi. On non-serial dynamic programming. *J. Comb. Theory, Ser. A*, 14(2):137–148, 1973.

[14] H. L. Bodlaender. Dynamic programming on graphs with bounded treewidth. In T. Lepistö and A. Salomaa, editors, *Automata, Languages and Programming, 15th International Colloquium, ICALP88, Tampere, Finland, July 11-15, 1988, Proceedings*, volume 317 of *Lecture Notes in Computer Science*, pages 105–118. Springer, 1988.

[15] H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybernet.*, 11(1-2):1–21, 1993.

[16] É. Bonnet, N. Bousquet, P. Charbit, S. Thomassé, and R. Watrigant. Parameterized complexity of independent set in H-free graphs. *Algorithmica*, 82(8):2360–2394, 2020.

[17] S. P. Borgatti, A. Mehra, D. J. Brass, and G. Labianca. Network analysis in the social sciences. *science*, 323(5916):892–895, 2009.

[18] G. Borradaile, D. Eppstein, and P. Zhu. Planar induced subgraphs of sparse graphs. *J. Graph Algorithms & Applications*, 19(1):281–297, 2015.

[19] A. Brandstädt and C. Hundt. Ptolemaic graphs and interval graphs are leaf powers. In E. S. Laber, C. F. Bornstein, L. T. Nogueira, and L. Faria, editors, *LATIN 2008: Theoretical Informatics, 8th Latin American Symposium, Búzios, Brazil, April 7-11, 2008, Proceedings*, volume 4957 of *Lecture Notes in Computer Science*, pages 479–491. Springer, 2008.

[20] A. Brandstädt, C. Hundt, F. Mancini, and P. Wagner. Rooted directed path graphs are leaf powers. *Discrete Math.*, 310(4):897–910, 2010.

[21] A. Brandstädt and V. B. Le. Structure and linear time recognition of 3-leaf powers. *Inform. Process. Lett.*, 98(4):133–138, 2006.

[22] A. Brandstädt, V. B. Le, and D. Rautenbach. A forbidden induced subgraph characterization of distance-hereditary 5-leaf powers. *Discrete Math.*, 309(12):3843–3852, 2009.

[23] A. Brandstädt, V. B. Le, and R. Sritharan. Structure and linear-time recognition of 4-leaf powers. *ACM Trans. Algorithms*, 5(1):A11:1–A11:22, 2009.

[24] A. Brandstädt and P. Wagner. On k-versus $(k+1)$-leaf powers. In B. Yang, D. Du, and C. A. Wang, editors, *Combinatorial Optimization and Applications, Second International Conference, COCOA 2008, St. John's, NL, Canada, August 21-24, 2008. Proceedings*, volume 5165 of *Lecture Notes in Computer Science*, pages 171–179. Springer, 2008.

[25] J. F. Buss and J. Goldsmith. Nondeterminism within P. *SIAM J. Comput.*, 22(3):560–572, 1993.

[26] A. Butman, D. Hermelin, M. Lewenstein, and D. Rawitz. Optimization problems in multiple-interval graphs. In N. Bansal, K. Pruhs, and C. Stein, editors, *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 268–277. SIAM, 2007.

[27] L. Cai. Fixed-parameter tractability of graph modification problems for hereditary properties. *Inf. Process. Lett.*, 58(4):171–176, 1996.

[28] L. Cai, S. M. Chan, and S. O. Chan. Random separation: A new method for solving fixed-cardinality optimization problems. In *International Workshop on Parameterized and Exact Computation*, pages 239–250. Springer, 2006.

[29] M.-S. Chang and M.-T. Ko. The 3-Steiner root problem. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 109–120. Springer, 2007.

[30] M.-S. Chang, M.-T. Ko, and H.-I. Lu. Linear-time algorithms for tree root problems. *Algorithmica*, 71(2):471–495, 2015.

[31] P. Chaudhuri and R. K. Ghosh. Parallel algorithms for analyzing activity networks. *BIT Comput. Sci. Sect.*, 26(4):418–429, 1986.

[32] J. Chen, I. A. Kanj, and W. Jia. Vertex cover: Further observations and further improvements. *J. Algorithms*, 41(2):280–301, 2001.

[33] J. Chen, I. A. Kanj, and G. Xia. Improved parameterized upper bounds for vertex cover. In R. Kralovic and P. Urzyczyn, editors, *Mathematical Foundations of Computer Science 2006, 31st International Symposium, MFCS 2006, Stará Lesná, Slovakia, August 28-September 1, 2006, Proceedings*, volume 4162 of *Lecture Notes in Computer Science*, pages 238–249. Springer, 2006.

[34] J. Chen, Y. Liu, S. Lu, B. O'Sullivan, and I. Razgon. A fixed-parameter algorithm for the directed feedback vertex set problem. In C. Dwork, editor, *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, pages 177–186. ACM, 2008.

[35] Z.-Z. Chen, T. Jiang, and G. Lin. Computing phylogenetic roots with bounded degrees and errors. *SIAM J. Comput.*, 32(4):864–879, 2003.

[36] B. N. Clark, C. J. Colbourn, and D. S. Johnson. Unit disk graphs. *Discret. Math.*, 86(1-3):165–177, 1990.

[37] B. Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Inform. and Comput.*, 85(1):12–75, 1990.

[38] B. Courcelle. On the expression of graph properties in some fragments of monadic second-order logic. In N. Immerman and P. G. Kolaitis, editors, *Descriptive Complexity and Finite Models: Proceedings of a DIMACS Workshop, January 14–17, 1996, Princeton University*, volume 31 of *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.*, pages 33–62. American Mathematical Society, Providence, RI, 1997.

[39] B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In *Handbook of graph grammars and computing by graph transformation, Vol. 1*, pages 313–400. World Scientific, River Edge, NJ, 1997.

[40] B. Courcelle, J. Engelfriet, and G. Rozenberg. Handle-rewriting hypergraph grammars. *J. Comput. System Sci.*, 46(2):218–270, 1993.

[41] B. Courcelle, J. A. Makowsky, and U. Rotics. Linear time solvable optimization problems on graphs of bounded clique-width. *Theory Comput. Syst.*, 33(2):125–150, 2000.

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

[43] K. K. Dabrowski, V. V. Lozin, H. Müller, and D. Rautenbach. Parameterized algorithms for the independent set problem in some hereditary graph classes. In C. S. Iliopoulos and W. F. Smyth, editors, *Combinatorial Algorithms - 21st International Workshop, IWOCA 2010, London, UK, July 26-28, 2010, Revised Selected Papers*, volume 6460 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2010.

[44] G. Di Battista, R. Tamassia, and I. G. Tollis. Area requirement and symmetry display of planar upward drawings. *Discrete and Computational Geometry*, 7(4):381–401, 1992.

[45] M. Dom, J. Guo, F. Hüffner, and R. Niedermeier. Error compensation in leaf root problems. In R. Fleischer and G. Trippen, editors, *Algorithms and Computation, 15th International Symposium, ISAAC 2004, Hong Kong, China, December 20-22, 2004, Proceedings*, volume 3341 of *Lecture Notes in Computer Science*, pages 389–401. Springer, 2004.

[46] M. Dom, J. Guo, F. Hüffner, and R. Niedermeier. Extending the tractability border for closest leaf powers. In D. Kratsch, editor, *Graph-Theoretic Concepts in Computer Science, 31st International Workshop, WG 2005, Metz, France, June 23-25, 2005, Revised Selected Papers*, volume 3787 of *Lecture Notes in Computer Science*, pages 397–408. Springer, 2005.

[47] R. G. Downey and M. R. Fellows. Fixed-parameter tractability and completeness I: basic results. *SIAM J. Comput.*, 24(4):873–921, 1995.

[48] R. G. Downey and M. R. Fellows. Fixed-parameter tractability and completeness II: on completeness for W[1]. *Theor. Comput. Sci.*, 141(1&2):109–131, 1995.

[49] R. G. Downey and M. R. Fellows. *Fundamentals of Parameterized Complexity.* Texts in Computer Science. Springer, 2013.

[50] G. Ducoffe. The 4-Steiner root problem. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 14–26. Springer, 2019.

[51] G. Ducoffe. Finding cut-vertices in the square roots of a graph. *Discrete Applied Mathematics*, 257:158–174, 2019.

[52] D. Eppstein. Subgraph isomorphism in planar graphs and related problems. *J. Graph Algorithms Appl.*, 3(3):1–27, 1999.

[53] D. Eppstein, D. Frishberg, and E. Havvaei. Simplifying activity-on-edge graphs. In S. Albers, editor, *17th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2020, June 22-24, 2020, Tórshavn, Faroe Islands*, volume 162 of *LIPIcs*, pages 24:1–24:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[54] D. Eppstein and M. T. Goodrich. Studying (non-planar) road networks through an algorithmic lens. In W. G. Aref, M. F. Mokbel, and M. Schneider, editors, *16th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS 2008, November 5-7, 2008, Irvine, California, USA, Proceedings*, page 16. ACM, 2008.

[55] D. Eppstein and S. Gupta. Crossing patterns in nonplanar road networks. In E. G. Hoel, S. D. Newsam, S. Ravada, R. Tamassia, and G. Trajcevski, editors, *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2017, Redondo Beach, CA, USA, November 7-10, 2017*, pages 40:1–40:9. ACM, 2017.

[56] D. Eppstein, S. Gupta, and E. Havvaei. Parameterized complexity of finding subgraphs with hereditary properties on hereditary graph classes. *arXiv preprint arXiv:2101.09918*, 2021.

[57] D. Eppstein and E. Havvaei. Parameterized leaf power recognition via embedding into graph products. *Algorithmica*, 82(8):2337–2359, 2020.

[58] D. Eppstein and H. Khodabandeh. On the edge crossings of the greedy spanner. *arXiv preprint arXiv:2002.05854*, 2020.

[59] D. Eppstein, P. Kindermann, S. Kobourov, G. Liotta, A. Lubiw, A. Maignan, D. Mondal, H. Vosoughpour, S. Whitesides, and S. Wismath. On the planar split thickness of graphs. *Algorithmica*, 80(3):977–994, 2018.

[60] D. Eppstein, M. Löffler, and D. Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In *International Symposium on Algorithms and Computation*, pages 403–414. Springer, 2010.

[61] D. Eppstein and J. A. Simons. Confluent Hasse diagrams. *J. Graph Algorithms Appl.*, 17(7):689–710, 2013.

[62] D. Eppstein, D. Strash, and M. Löffler. Listing all maximal cliques in large sparse real-world graphs in near-optimal time. *J. Experimental Algorithmics*, 18(3):3.1, 2013.

[63] M. R. Fellows, D. Hermelin, F. A. Rosamond, and S. Vialette. On the parameterized complexity of multiple-interval graph problems. *Theor. Comput. Sci.*, 410(1):53–61, 2009.

[64] W. M. Fitch and E. Margoliash. Construction of phylogenetic trees. *Science*, 155(3760):279–284, 1967.

[65] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[66] A. Garg and R. Tamassia. Upward planarity testing. *Order*, 12(2):109–133, 1995.

[67] A. Garg and R. Tamassia. On the computational complexity of upward and rectilinear planarity testing. *SIAM J. Comput.*, 31(2):601–625, 2001.

[68] P. A. Golovach, D. Kratsch, D. Paulusma, and A. Stewart. Finding cactus roots in polynomial time. In V. Mäkinen, S. J. Puglisi, and L. Salmela, editors, *Combinatorial Algorithms - 27th International Workshop, IWOCA 2016, Helsinki, Finland, August 17-19, 2016, Proceedings*, volume 9843 of *Lecture Notes in Computer Science*, pages 361–372. Springer, 2016.

[69] M. C. Golumbic and U. Rotics. On the clique-width of some perfect graph classes. *Int. J. Found. Comput. Sci.*, 11(3):423–443, 2000.

[70] M. Grohe. Computing crossing numbers in quadratic time. In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing*, pages 231–236, New York, 2001. ACM.

[71] F. Gurski and E. Wanke. The clique-width of tree-power and leaf-power graphs. In A. Brandstädt, D. Kratsch, and H. Müller, editors, *Graph-Theoretic Concepts in Computer Science, 33rd International Workshop, WG 2007, Dornburg, Germany, June 21-23, 2007. Revised Papers*, volume 4769 of *Lecture Notes in Computer Science*, pages 76–85. Springer, 2007.

[72] R. Halin. S-functions for graphs. *J. Geometry*, 8(1-2):171–186, 1976.

[73] F. Harary. *Graph theory.* Addison-Wesley, 1991.

[74] E. Havvaei and N. Deo. A game-theoretic approach for detection of overlapping communities in dynamic complex networks. *International Journal of Mathematical and Computational Methods*, 1:313–324, 2016.

[75] E. Havvaei and N. Deo. A game-theoretic approach for detection of overlapping communities in dynamic complex networks. *CoRR, vol. abs/1603.00509*, 2016.

[76] P. Healy, A. Kuusik, and S. Leipert. A characterization of level planar graphs. *Discrete Math.*, 280(1-3):51–63, 2004.

[77] P. Healy and N. S. Nikolov. Hierarchical Graph Drawing. In R. Tamassia, editor, *Handbook of Graph Drawing and Visualization*, pages 409–453. CRC Press, 2014.

[78] D. Hermelin, M. Mnich, and E. J. van Leeuwen. Parameterized complexity of induced graph matching on claw-free graphs. *Algorithmica*, 70(3):513–560, 2014.

[79] P. Hliněný. Branch-width, parse trees, and monadic second-order logic for matroids. *J. Combin. Theory Ser. B*, 96(3):325–351, 2006.

[80] J. L. Householder and H. E. Rutland. Who owns float? *J. Construction Engineering and Management*, 116(1):130–133, 1990.

[81] W. Huang and L. Ding. Project-scheduling problem with random time-dependent activity duration times. *IEEE Transactions on Engineering Management*, 58(2):377–387, 2011.

[82] M. Jünger and S. Leipert. Level Planar Embedding in Linear Time. In J. Kratochvıl, editor, *Graph Drawing: 7th International Symposium*, volume 1731 of *Lecture Notes in Computer Science*, pages 72–81. Springer-Verlag, 1999.

[83] W. Kennedy, G. Lin, and G. Yan. Strictly chordal graphs are leaf powers. *J. Discrete Algorithms*, 4(4):511–525, 2006.

[84] S. Khot and V. Raman. Parameterized complexity of finding subgraphs with hereditary properties. *Theor. Comput. Sci.*, 289(2):997–1008, 2002.

[85] T. Kloks. *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer, 1994.

[86] O. Kuchaiev, T. Milenković, V. Memišević, W. Hayes, and N. Pržulj. Topological network alignment uncovers biological function and phylogeny. *Journal of the Royal Society Interface*, 7(50):1341–1354, 2010.

[87] V. G. Kulkarni. A compact hash function for paths in PERT networks. *Operations Research Letters*, 3(3):137–140, 1984.

[88] L. C. Lau. Bipartite roots of graphs. *ACM Transactions on Algorithms*, 2(2):178–208, 2006.

[89] V. B. Le and N. N. Tuy. Hardness results and efficient algorithms for graph powers. In C. Paul and M. Habib, editors, *Graph-Theoretic Concepts in Computer Science, 35th International Workshop, WG 2009, Montpellier, France, June 24-26, 2009. Revised Papers*, volume 5911 of *Lecture Notes in Computer Science*, pages 238–249, 2009.

[90] V. B. Le and N. N. Tuy. The square of a block graph. *Discret. Math.*, 310(4):734–741, 2010.

[91] J. M. Lewis and M. Yannakakis. The node-deletion problem for hereditary properties is NP-complete. *J. Comput. Syst. Sci.*, 20(2):219–230, 1980.

[92] D. R. Lick and A. T. White. k-degenerate graphs. *Canadian J. of Mathematics*, 22:1082–1096, 1970.

[93] D. G. Malcolm, J. H. Roseboom, C. E. Clark, and W. Fazar. Application of a technique for research and development program evaluation. *Operations Research*, 7(5):646–669, 1959.

[94] D. Marx. Efficient approximation schemes for geometric problems? In G. S. Brodal and S. Leonardi, editors, *Algorithms - ESA 2005, 13th Annual European Symposium, Palma de Mallorca, Spain, October 3-6, 2005, Proceedings*, volume 3669 of *Lecture Notes in Computer Science*, pages 448–459. Springer, 2005.

[95] D. Marx. Parameterized complexity of independence and domination on geometric graphs. In H. L. Bodlaender and M. A. Langston, editors, *Parameterized and Exact Computation, Second International Workshop, IWPEC 2006, Zürich, Switzerland, September 13-15, 2006, Proceedings*, volume 4169 of *Lecture Notes in Computer Science*, pages 154–165. Springer, 2006.

[96] O. Mason and M. Verwoerd. Graph theory and networks in biology. *IET systems biology*, 1(2):89–119, 2007.

[97] J. Nešetřil and P. Ossona de Mendez. 18.3 The Subgraph Isomorphism Problem and Boolean Queries. In *Sparsity: Graphs, Structures, and Algorithms*, volume 28 of *Algorithms and Combinatorics*, pages 400–401. Springer-Verlag, 2012.

[98] K. Neumann and C. Schwindt. Activity-on-node networks with minimal and maximal time lags and their application to make-to-order production. *Operations-Research-Spektrum*, 19(3):205–217, 1997.

[99] N. Nishimura, P. Ragde, and D. M. Thilikos. On graph powers for leaf-labeled trees. *J. Algorithms*, 42(1):69–108, 2002.

[100] R. D. Page and E. C. Holmes. *Molecular evolution: a phylogenetic approach.* John Wiley & Sons, 2009.

[101] S. Poljak. A note on stable sets and colorings of graphs. *Comment. Math. Univ. Carolinae*, 15:307–309, 1974.

[102] D. Rautenbach. Some remarks about leaf roots. *Discrete Math.*, 306(13):1456–1461, 2006.

[103] N. Robertson and P. D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986.

[104] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Trans. Systems, Man, and Cybernetics*, 11(2):109–125, 1981.

[105] C. R. Woese. Interpreting the universal phylogenetic tree. *Proceedings of the National Academy of Sciences*, 97(15):8392–8396, 2000.

[106] Z. Xu. Automatic layout of information in the AOE network. In *2010 International Conference on Mechanic Automation and Control Engineering.* IEEE, June 2010.