DISSERTATION

GENERALIZATIONS OF COMPARABILITY GRAPHS

Submitted by

Zhisheng Xu

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2022

Doctoral Committee:

Advisor: Ross McConnell

Francisco Ortega
Harvey Cutler
Alexander Hulpke

ABSTRACT

GENERALIZATIONS OF COMPARABILITY GRAPHS

In rational decision-making models, transitivity of preferences is an important principle. In a transitive preference, one who prefers x to y and y to z must prefer x to z. Many preference relations, including total order, weak order, partial order, and semiorder, are transitive. As a preference which is transitive yet not all pairs of elements are comparable, partial orders have been studied extensively. In graph theory, a comparability graph is an undirected graph which connects all comparable elements in a partial order. A transitive orientation is an assignment of direction to every edge so that the resulting directed graph is transitive. A graph is transitive if there is such an assignment. Comparability graphs are a class of graphs where clique, coloring, and many other optimization problems are solved by polynomial algorithms. It also has close connections with other classes of graphs, such as interval graphs, permutation graphs, and perfect graphs.

In this dissertation, we define new measures for transitivity to generalize comparability graphs. We introduce the concept of double threshold digraphs together with a parameter λ which we define as our degree of transitivity. We also define another measure of transitivity, β, as the longest directed path such that there is no edge from the first vertex to the last vertex. We present approximation algorithms and parameterized algorithms for optimization problems and demonstrate that they are efficient for "almost-transitive" preferences.

ACKNOWLEDGEMENTS

Thank you to everyone who has helped me along the way to completing this dissertation.

I am especially grateful for the expert guidance and support that I received from my advisor, Prof. Ross McConnell, during my Ph.D. studies at Colorado State University (CSU). He has always been encouraging and has offered insights to help me develop my research ideas. Without his patience and persistence it would have been impossible for me to complete this dissertation.

I would also like to thank my other committee members, Prof. Francisco Ortega, Prof. Harvey Cutler, and Prof. Alexander Hulpke. Their constructive suggestions and comments are valuable for improving this dissertation.

Finally, I would like to thank all my friends who have helped to make the last few years in Fort Collins enjoyable and memorable, my parents, grandparents, and my son Neal, who has been a great source of strength throughout this period of time.

DEDICATION

This disseration is dedicated to my son Neal, and to my parents.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

Given that the majority of optimization problems in graph theory and its applications to computer science and operations research are NP-complete, much attention has been paid to graph classes that may still have polynomial algorithms for optimization problems. Comparability graphs are an example of such a special class of graphs in which maximum clique, minimum coloring, and many other optimization problems have polynomial algorithms because of its relationship to partial order [1] [2]. As a fundamental building block in science and information processing, comparability graphs and partial orders play an integral role in fields such as planning [3] [4], scheduling theory [5] [6], data mining [7], and chemistry [8] [9] [10]. Comparability graphs and partial orders, as well as related problems such as modular decomposition and transitive orientation problems, have been extensively studied [11] [12].

An important application of orders is to explain people's behavior by assuming that they can rank their choices according to their preferences. In economics, the use of orders can help understand satisficing behavior in a more natural way and thus, build economic models and predict customer demands. This is achieved by intuitively assigning each item a value, i.e., if a person prefers x to y, the value assigned to x should be greater than that of y. By observing the choices individuals make, we can infer their preferences as well as build an understanding of their values through experiments. On the basis of such transitive preferences, it is possible to solve useful optimization problems with efficient algorithms.

However, this approach has a number of disadvantages. For an application to use algorithms on comparability graphs, the input data must be strictly transitive. Many economists and decision theorists believe transitivity is a key element of rational choice in decision theory. However, non-transitive preferences exist, and non-transitive preferences have also been discussed for a long time [13]. It is not uncommon for non-transitive preferences to occur in a variety of decision contexts as a result of poor judgment, random and sloppy choices, or unexamined biases. It is

possible, for example, that someone prefers x to y and y to z, but he is indifferent between x and z.

Ideally, we would like to develop mathematical models to provide a reasonable representation of non-transitive preferences. Our goal is to provide a measure of transitivity so that when the graph is transitive, we may use the results from previous studies of comparability graphs and partial order to develop a polynomial algorithm. In case the graph is non-transitive, we have algorithms based on the transitivity of the graph, and we hope to develop algorithms based on the transitivity of the graph. Additionally, the more transitive the graph is, the more efficient the algorithms will become.

Semiorders were introduced in [14] as a possible mathematical for preference in social science. *Semiorder* is defined as a type of ordering in which each item x has a numerical score $\alpha(x)$, and where there is a given margin of error t, such that y is comparable to x if $\alpha(y) - \alpha(x) > t$. Semiorders are special cases of partial orders, and are therefore transitive.

In chapter 6, we propose a generalization of a semiorder in which it reflects the fact that there could be a range of difference between the minimum of difference that can sometimes be perceived and the minimum difference that can be reliably perceived. In the new model of double-threshold semiorders, there are two thresholds, t_1 and t_2; if the difference $\alpha(y) - \alpha(x)$ is less than t_1, then y is not preferred to x; if the difference is greater than t_2 then y is preferred to x; if it is between t_1 and t_2, then y may or may not be preferred to x. We call such a relation a (t_1, t_2) *double-threshold semiorder*, and the corresponding directed graph $G = (V, E)$ a (t_1, t_2) *double-threshold digraph*. With correct assignment of t_1 and t_2 values, every directed acyclic graph is a double-threshold digraph. Obviously, a (t_1, t_2) double-threshold semiorder is a semiorder if $t_1 = t_2$. We introduce a parameter λ that describes the minimum ratio of t_2/t_1 satisfiable by a given DAG G λ shows that DAGs can be subclassed into increasingly nested hierarchies. There is an O(nm/λ) algorithm for finding λ. Furthermore, we show that the parameter λ provides a useful measure of transitivity and we present approximation algorithms and parameterized algorithms for optimization problems on arbitrary DAGs.

In chapter 7, we also propose another measure of transitivity, β, which corresponds to the longest directed path that does not have an edge connecting the first vertex to the last vertex. Similar to λ, we present approximation algorithms and parameterized algorithms for optimization problems. Also, we compare lambda with beta in terms of the values. Then, we show the hardness of finding an undistributed graph with a β-minimizing acyclic orientation.

Chapter 2

Preliminaries

A graph $G = (V, E)$ consists of a finite set V of vertices, and a finite set $E \subset V \times V$ of edges. In this paper, we will consider only graphs that have no loops or multiple edges. Every ordered pair $(u, v) \in E$ is a *directed edge*. When there is an edge (u, v), we say that v is adjacent to u. The directed edge (u, v) is an *outgoing edge* on u and an *incoming edge* on v. The *set of neighbors* of a vertex x is noted $N(x)$. When (u, v) and (v, u) are both edges, we let $uv = \{(u, v), (v, u)\}$ denote an *undirected edge*. If for any edge (u, v), (v, u) is an edge, the graph is an undirected graph. We let n denote $|V|$, and m denote $|E|$.

Given an undirected graph $G = (V, E)$, we define the *complement* of G as $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{(u, v) : u, v \in V, u \neq v, (u, v) \notin E\}$. A $clique$ C is a subset of V such that $xy \in E$ for any $x, y \in C$. A clique C is a *maximal clique* if there does not exist a vertex x such that $C \cup \{x\}$ is a clique. The term clique comes from Luce and Perry [15]. They use complete subgraphs in social networks to model the cliques of people. A $path$ is a sequence of vertices $(v_0, v_1, \ldots, v_k)$ $(k \geq 1)$ that consecutive vertices $v_i v_{i+1}$ has an edge for $0 \leq i \leq k-1$. If a path contains no repeat vertices, it is a *simple path*. If two paths do not share a vertex, they are *vertex disjoint*. A *directed acyclic graph* (DAG) is a directed graph that has no cycles. An *induced graph* $G|X$ can be generated using the set X as the set of vertices and edge $(u, v) \in E$ such that $u \in X$ and $v \in X$ as the set of edges. The *vertex separator* S for nonadjacent vertices a and b is a subset of V such that if S is removed from the graph, a and b are separated into distinct connected components.

The *set union* $A \cup B$, is defined by $A \cup B = \{x | x \in A \text{ or } x \in B\}$. The *set difference* $A - B$ is defined by $A - B = \{x | x \in A \text{ and } x \notin B\}$. Two sets A, B *overlap* if they intersect and neither of them contains the other, i.e., $A \cap B \neq \emptyset$, $A - B \neq \emptyset$, and $A - B \neq \emptyset$. The *symmetric set difference* $A \Delta B$, is defined to be $(A - B) \cup (B - A)$. The *trivial subsets* of a set A are $\emptyset$ and A itself.

Chapter 3

Background

3.1 Optimization Problems in Graph Theory

Graph theory is considered a fundamental tool for many arrangement, networking, optimization, matching, and operational problems. Optimal solutions or approximation solutions to graph theory problems such as maximum clique, minimum coloring, maximum independent set, and minimum clique cover offers useful insights and efficient algorithms for real-world applications.

The *maximum clique problem* on a graph $G(V, E)$ is to find a clique C in G with maximum cardinality $|C|$. The cardinality of the maximum clique on G is usually written as $\omega(G)$. A subset $S \in V$ in $G = (V, E)$ is an *independent set* if there is no edge between any pair of vertices of V. The *maximum independent set problem* is to find an independent set S such that its cardinality, $|S|$, is maximum. In general, the cardinality of the maximum independent set on G is denoted as $\alpha(G)$. The *complement graph* of G is a graph $\overline{G}$ such that it has the same vertex set as G, and vertices u, v are adjacent in G if and only if when they are nonadjacent in G. It is easy to see $\omega(G) = \alpha(\overline{G})$. *Graph coloring* is the problem that assigns each vertex with a color such that no two adjacent vertices have the same color. The *chromatic number* of a graph G, $\chi(G)$, is the smallest number of colors needed to color the vertices of G. A *clique cover* is a partition of the vertices of the graph into cliques. The *clique covering number* $\theta(G)$ of a graph G is the minimum number of cliques in G needed to cover the vertex set of G. Similarly, $\chi(G) = \theta(\overline{G})$. A *vertex cover* of a graph is a set of vertices that includes at least one endpoint of every edge of the graph. Let *vertex cover number* $\kappa(G)$ denote the size of a minimum vertex cover. The optimization problem on a graph has many applications ranging from code theory, reliability, genetic probability, planning, frequency allocation, and many other areas.

3.2 NP-completeness

3.2.1 Overview

In 1971, Steve Cook published his paper "The complexity of theorem-proving procedures" [16], which provided the first published NP-completeness results. Cook formalized the notions of polynomial-time reduction and NP-completeness in this paper. Furthermore, he proved the existence of an NP-complete problem by showing that the Boolean Satisfiability Problem (usually known as SAT) is NP-complete. This result is also known as "Cook's Theorem." At the same time, however, Leonid Levin identified the same results, although his results weren't published until 1973. "Cook's Theorem" is now commonly referred to as the "Cook-Levin Theorem" because of the fact that Levin's independent results were accomplished at the same time. Besides the Boolean satisfiability problem, in 1972, Richard Karp proved that several other problems (Karp's 21 NP-complete problems) were NP-complete [17] (see Figure 3.1). Thus, there exists a class of NP-complete problems. Since each one of the 21 problems is NP-complete, even though they present a tree structure in Figure 3.1, they can all reduce to each other. As an example, we show the reduction from Vertex Cover to Clique in 3.2.2.

All the optimization problems we mentioned in section 3.1 have straightforward brute-force exponential solutions. Many people are interested in finding a polynomial solution for them. Generally, we consider problems that are solvable in polynomial time as tractable and the problems that require an exponential time to solve as intractable. We can naturally think of tractable problems to be easier problems and intractable problems to be harder problems. The NP-completeness theory offers insight into deciding whether certain types of problems are more likely to be solvable in polynomial time. NP-completeness applies directly to decision problems, namely, problems with an answer that is either "yes" or "no." Problems such as Maximum Independent Set, Maximum Clique, Minimum Coloring, and Minimum Clique Cover are optimization problems that have an associated value as a solution. Although NP-complete problems are confined to decision problems, we can still connect optimization problems to decision problems by imposing a bound on the value we try to optimize. For example, Graph Coloring is an optimization problem, but determining

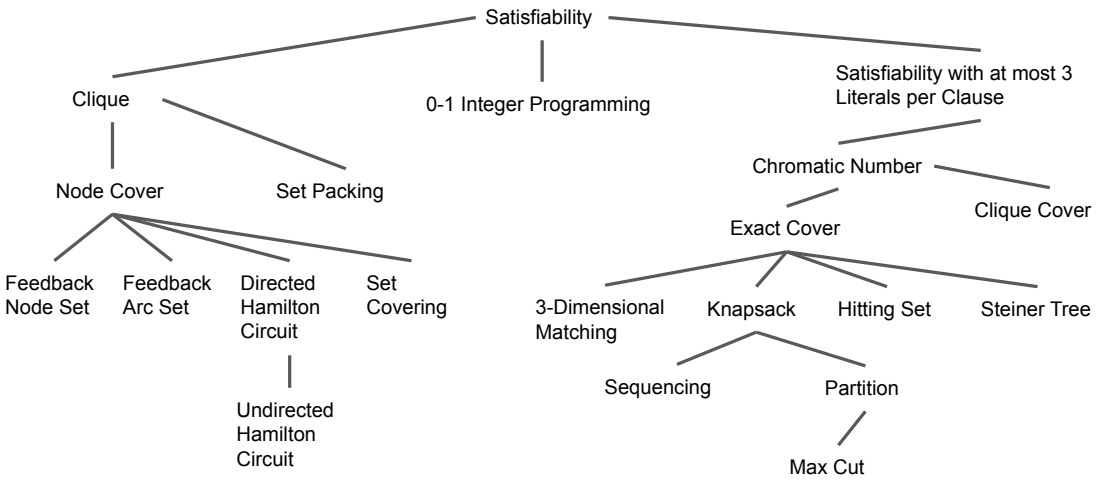

Figure 3.1: Karp's 21 NP-complete problems. Reduction goes from the upper level to the lower one. Note that Chromatic Number is also called Graph Coloring Problem; Directed/Undirected Hamilton Circuit is now called Directed/Undirected Hamilton Cycle; some definitions of the problems such as the Knapsack problem are different from today.

whether a graph is k-colorable is a decision problem. Therefore, the theory of NP-completeness is relevant to optimization problems.

Before using any terms in NP-completeness, we will briefly explain the concept of reduction and several classes of problems related to NP-completeness: P, NP, NP-complete, and NP-hard. A *reduction* from an optimization problem A to an optimization problem B is a procedure that can transform any instance of A into some instance of B with the requirements: first, the transformation takes polynomial time; second, the answer for A is "yes" if and only if the answer for B is also "yes." We only consider decision problems in P, NP, and NP-complete. A problem is in class *P* if it is solvable in polynomial time. A problem is in class *NP* if a certificate that shows a possible result of the input problem is verifiable in polynomial time in the size of the input. A problem is in class *NP-complete* if it is in NP, and every problem in NP can reduce to it. In other words, the NP-complete problems are the "hardest" problems in class NP. The class of NP-complete problems is a very important and interesting class of problems, and we do not know whether it is solvable in polynomial time. A problem is in class *NP-hard* if it is at least as hard as the NP-complete problems. Notice that NP-hard problems do not have to be in NP, and they do not have to be

decision problems. If an optimization has its decision variants in NP-complete, we will commonly refer to it to be NP-hard.

It has already been proved that the problems mentioned above were amongst the first few problems that were shown to be NP-hard. For instance, in the minimum graph coloring problem, determining if a graph is 2-colorable is equivalent to determining whether or not the graph is bipartite, then it is computable in linear time using the breadth-first search or depth-first search. However, determining if a graph is 3-colorable is one of the already known NP-complete problems. This makes the minimum coloring problem NP-hard. It is widely believed, but so far, no proof is available to prove or disprove the famous problem "P = NP," that NP-complete problems do not have polynomial algorithms. Without awareness of NP-completeness could result in attacking the same problem over and over again with little hope of progress. Therefore, if a problem is shown to be NP-complete or NP-hard, we should stop looking for polynomial algorithms as such a solution would contradict commonly accepted assumptions.

3.2.2 Reduction example from Vertex Cover to Clique

Let VERTEX-COVER(G, k) denote the decision problem of whether G has a vertex cover of size k. Also, let CLIQUE(G, k) be the problem of whether G has a clique of size k. We notice that if there exists a clique $C \in \overline{G}$, then $V - C$ is a vertex cover in G. An algorithm for reducing VERTEX-COVER(G, k) to CLIQUE goes as follows: For a given a graph $G = (V, E)$ and an integer k, we generate its complement graph $\overline{G}$ = (V, $\overline{E}$); then we solve the problem CLIQUE($\overline{G}$, $|V| - k$); If there is a solution for CLIQUE($\overline{G}$, $|V| - k$), return "Yes" for VERTEX-COVER(G, k), else return "No".

To prove that this reduction is correct, we need to show the following two statements are true:

- If there is a solution to VERTEX-COVER(G, k), then there must be a solution to CLIQUE(G, $|V| - k$)

  Proof. Suppose $G = (V, E)$ has a vertex cover $S \subseteq V$, where $|S| = k$. For any edge $(u, v) \in E$, either $u \in S$ or $v \in S$. Thus, for any pair of vertices u' and v', if $u' \notin S$ and

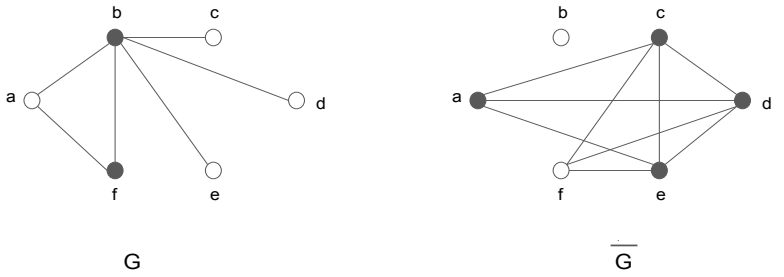

Figure 3.2: The left is a graph $G = (V, E)$ with vertex cover $\{b, f\}$. The right is the complement $\overline{G}$ with clique {a, c, d, e}.

$v' \notin S$, then $(u', v') \notin E$. This is equivalent to say that for any pair of vertices that are both not in the vertex cover V of G, there is an edge between them in $\overline{G}$. Therefore $G - S$ is a clique in $\overline{G}$ of size $|V| - k$. □

- If there is a solution to CLIQUE(G, $|V| - k$), then there must be a solution to VERTEX-COVER(G, k)

Proof. Conversely, suppose that $\overline{G}$ has a clique $C \subseteq V$, where $|C| = |V| - k$. Let (u, v) be an edge in E, so $(u, v) \notin \overline{E}$. Since C is a clique in $\overline{G}$, C can not include both u and v. This implies that at least one of u or v does not belong to C, since every pair of vertices is connected in C. Thus, at least one of u or v is in $V - C$, which renders $V - C$ a size k set cover of G. □

3.2.3 Other Techniques

Although it is difficult to find polynomial algorithms, we can nevertheless apply other techniques to get a solution for the optimization problem in practice, and the following techniques can lead to considerably quicker algorithms:

- Restriction: If we restrict the input graph to a particular structure, such as planer graphs, interval graphs, or comparability graphs, polynomial algorithms are possible. The restriction may narrow

the application of the algorithm; however, the input of various problems is bounded to a specific structure. For instance, design problems in circuits or transportation have an input of an interval graph. In bioinformatics, sequence assembly is the problem of aligning and merging fragments from a longer DNA sequence to reconstruct the original sequence. The input to the sequence assembly problem usually forms an interval graph or a circular-arc graph.

- Approximation: Rather than searching for an optimal solution, approximation is the process of trying to find a solution that is within a factor of an optimal one. Using approximation algorithms, we can prove a bound on the ratio between the optimal solution and the result of the approximation algorithm, and we can arrive at this solution in less than exponential time. Additional background information about approximation algorithms is included in chapter 3.3.

- Parameterization: Parameterized complexity analysis measures the time complexity of an algorithm not just in terms of the input length but also one or multiple side parameters. The complexity of a problem is then measured as a function of those parameters. Parameterized complexity allows us to study how different parameters influence the complexity of the problem. In parameterized algorithms, a parameter k is simply a relevant secondary measurement that encapsulates some aspect of the input instance. For problems involving the satisfiability of Boolean formulas, k can be the number of variables, or clauses, or the number of clauses that need to be satisfied. For problems involving a set of strings, k can be the maximum length of the strings, or the size of the alphabet, or the maximum number of distinct symbols appearing in each string. A discussion of parameterized algorithms is presented in the following chapter 3.4.

- Randomization: An algorithm that uses random numbers to decide what to do next anywhere in its logic is called a Randomized Algorithm. For example, in Randomized Quick Sort, we use a random number to pick the next pivot. In some cases, probabilistic algorithms allow us to get a faster average running time and are the only practical means of solving a problem.

- Heuristics: Heuristic algorithms are meant to solve problems faster and more efficiently than conventional algorithms by sacrificing optimality, accuracy, precision, and completeness in ex-

change for speed. Heuristic algorithms are often used to solve NP-complete problems. Some of the most well-known examples of heuristic algorithms are Simulated Annealing, Swarm Intelligence, Genetic Algorithms, and Artificial Neural Networks. These algorithms simplify and apply clever simplifications or methods to solve computationally complex problems. Optimizing algorithms can be supplemented with heuristics to provide a good baseline or together with heuristics to produce an individual solution. When approximate solutions suffice and exact solutions would be computationally expensive, heuristic algorithms are employed most often. Our objective in this paper is to find exact solutions or use approximation algorithms or parameterized algorithms with mathematically rigorous proofs, so we will not present any heuristic algorithm in the paper.

3.3 Approximation algorithms

3.3.1 Introduction

Approximation algorithms allow us to relax the requirement of finding an optimal solution and instead settle for a "good enough" solution. An approximation algorithm aims to relax the requirement as little as possible. It tries to find a solution that closely approximates the optimal solution in terms of its value. In general, the relative difference between the result of an approximation algorithm and its optimal solution is determined by α, see definition 3.3.1.

Definition 3.3.1. An *α-approximation algorithm* for an optimization problem is a polynomial-time algorithm for all instances of the problem that produce a solution whose value is within a factor of α of the value of an optimal solution.

In an α-approximation algorithm, we will refer to α as the *approximation ratio* or *approximation factor* of the algorithm. In this paper, we will follow the convention that $\alpha > 1$ for minimization problems, while $\alpha < 1$ for maximization problem. We can use α as a mathematically rigorous method for proving how well a heuristic performs on all instances and be able to determine the

kinds of instances on which it does not perform well. The optimal solution to a problem will be denoted OPT when there is no ambiguity.

3.3.2 Example of an Approximation Algorithm

A sample approximate algorithm is presented for the problem of minimum vertex cover. Let us start with some definitions.

Definition 3.3.2. Given a graph $G = (V, E)$, a subset of the edge $M \subseteq E$ is said to be matching if no two edges of M share an endpoint.

A maximal matching is a matching that is not a subset of any other matching, i.e., we cannot add more edges to it. A maximum matching M is matching with the maximum possible cardinality. A maximal matching can easily be computed in polynomial time by simply greedily picking edges and removing the endpoints of picked edges. A maximum matching can also be solved in polynomial time [18]. The maximum matching problem in general graphs can be solved in $O(\sqrt{n}m)$, and there are various versions of this algorithm [19] [20] [21]. We can observe that maximal matching provides a lower bound for the Minimum Vertex Cover problem for G since a vertex cover must contain at least one endpoint of a matched edge. Based on this lower bounding scheme, the following algorithm is readily apparent.

Algorithm 1: Approximation Algorithm for Minimum Vertex Cover

```
1 begin
2 |   Find a maximal matching in G and return the set of matched vertices.
3 end
```

Theorem 3.3.1. *Algorithm 1 is a factor 2 approximation algorithm for the Minimum Vertex Cover problem.*

Proof. Let M be the maximal matching picked. Since M is maximal, no edge can be left uncovered by the set of vertices picked. This implies that M is a vertex cover of G. As argued above,

$|M| \leq$ OPT. This approximation factor follows from the observation that the algorithm picks the vertex cover with cardinality $2|M|$, which is at most 2*OPT. □

In the context of improving the approximation guarantee for Minimum Vertex Cover, the following question arises: can we improve the approximation guarantee of Algorithm 1 by a better analysis? Consider a complete bipartite graph, $K_{n,n}$. Algorithm 1 will choose all $2n$ vertices, whereas the minimal vertex cover is n. Therefore, the approximation factor 2 cannot be improved by better analysis. There are more connections between the minimum vertex cover problem and the matching problem, especially in bipartite graphs. In any bipartite graph, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover. This is known as *Kőnig's theorem* [22], which will be discussed in section 4.5.

3.3.3 Other Related Concepts

Definition 3.3.3. The class APX (an abbreviation of "approximable") is the set of NP optimization problems that allow polynomial-time approximation algorithms with approximation ratios bounded by a constant (or constant-factor approximation algorithms for short).

Definition 3.3.4. A *polynomial-time approximation scheme* (PTAS) is a family of algorithm $\{A_\epsilon\}$, where there is an algorithm for each $\epsilon > 0$, such that A_ϵ is a $(1+\epsilon)$-approximation algorithm (for minimization problems) or a $(1-\epsilon)$-approximation algorithms (for maximization problems).

Many problems have polynomial-time approximation schemes. However, there are some problems that are extremely difficult to approximate, for example, the max clique problem.

Theorem 3.3.2. *[23] Let n denote the number of vertices in an input graph, and consider any constant $\epsilon > 0$. Then there does not exist an O($n^{1-\epsilon}$)-approximation algorithm for the maximum clique problem unless P = NP.*

3.4 Parameterized algorithms

In parameterized algorithms, a parameter k refers to a secondary measurement that encapsulates certain aspects of the input instance. There is no standard way of defining k. It can be the size of the solution or a number describing how "structured" the input instance is.

Algorithm with running time of $O(f(k) * n^c)$ for parameter k and a constant c independent of n and k are called *fixed parameter algorithms*, or *FPT algorithm*. For example, an $O(1.2738^k + kn)$ time polynomial space parameterized algorithm is shown for Vertex Cover [24], where n is the number of vertices and k is the size of the vertex cover. This means that Vertex Cover is fixed-parameter tractable with the size of the solution as the parameter. Typically the goal of parameterized analysis is to design FPT algorithms. We want to make both the $f(k)$ factor and the constant c in the bound on the running time as small as possible. For some problems, we can come up with less efficient *XP algorithms*, where the running time is in the format of $f(k) * n^{g(k)}$.

3.4.1 Introduction

A simple FPT algorithm for the Vertex Cover problem is shown in the following example. We will use the volume of the optimal solution as the parameter k. For a graph $G = (V, E)$, the naive algorithm searches for every possible subset of k vertices and runs in time O(n^k), which is not FPT. However, FPT algorithms can be derived from a simple observation. The crucial point is that each edge must be covered. Thus, as long as there is at least one uncovered edge (u, v), we proceed as follows. Try to add u to the solution set and run the algorithm recursively to check whether the remaining graph can be resolved by at most $k - 1$ vertices. If this succeeds, you already have a solution. If it fails, then move u out of the solution set and run the same recursive algorithm with vertex v. If both recursive calls fail to find a solution, then we can be sure that there is no way to find a vertex cover of size k. Hence there is a total of 2^k recursive calls, and each recursive call is easy to implement in O($n + m$) time. Additionally, note that if there is a vertex x with $|n(v)| \leq k + 1$, we must include x in the solution; otherwise, we have to include all its neighbors, thereby going over our solution size limit k. We know from this observation that $m \leq nk/2$. Therefore, the

algorithm runs in O($2^k * k * n$), which satisfies the format to be FPT. This preprocessing step for achieving an equivalent "smaller sized" version of a given problem in polynomial time is called kernelization and is used universally across almost all practical implementations that attempt to solve an NP-hard problem. In section 3.4.2, we will discuss kernelization in more detail.

Different problems may not yield FPT algorithms when k is chosen to be the size of the optimal solution. Graph Coloring, for example, is NP-complete, so we do not hope for a polynomial-time algorithm in all cases. We will look at the Graph Coloring problem parameterized by the number of colors k. If it is FPT, we will have an O($f(k) * n^c$) algorithm for every k and some constant c. Since we also know that whether a graph is 3-colorable is NP-complete and O($f(k) * n^c$) is polynomial if $k = 3$. Thus, if Graph Coloring parameterized by the number of colors were in FPT, then P = NP. Note that even an XP algorithm with running time $f(k) * n^{g(k)}$ for it would also imply P = NP.

Examples such as Graph Coloring demonstrate that not all problems have FPT algorithms. Let us look at the Max Clique problem as another example problem. There is a simple O(n^k) time algorithm to check whether a clique of size at least k exists. This is an XP algorithm, but we do not know if there exists an FPT algorithm. In this case, NP-hardness is insufficient to differentiate between XP algorithms and FPT algorithms. In the W-hierarchy, deciding if a given graph contains a clique of size k is a W[1]-complete problem. Being a W[1]-complete problem allows us to prove (under certain complexity assumptions) that even though a problem is polynomial-time solvable for every fixed k, the parameter k has to append in the exponent of n in the running time. Thus, this problem is not FPT. This theory has proved to be a valuable tool in predicting which parameterized problems are FPT and which are not.

Despite the fact that, most likely, there is no FPT algorithm for cliques parameterized by size k, we can still choose other attributes of the problem as a parameter. Let us denote Δ as the maximum degree in a graph. Then we have a new algorithm: guess one vertex v in the clique, then the remaining vertices must be among the Δ neighbors of v. By this, we can try all of the 2^Δ subsets of the neighbors of v and return the largest clique we found. The total running time is

O($2^{\Delta} * \Delta^2 * n$), which is FPT. In this sense, the classification of the problem into "solvable" or "unsolvable" crucially depends on the choice of parameter.

We can always quickly find a parameter for most optimization problems: the solution's size. In some cases, we find a parameter that is a measure of some property of the input instance but is not explicitly given in the input. The reason we investigated this particular measure is that we thought that it was typically small in the input instances we cared about. These parameters may express some structural characteristic of a typical instance, and we can use any number of these parameters. When a graph has genus g, it can be drawn without crossing edges on a sphere with g holes in it. We may explore parameters for the genus if we believe the problem is easy on planar graphs and the instances are nearly planar. In addition, Treewidth [25] is a good parameter to use if the input instance is a tree-like graph.

3.4.2 Kernelization

We begin by presenting a few Vertex Cover kernelization algorithms based on a few natural reduction rules. Let a given instance VERTEX-COVER(G, k) (or (G, k) if there is no ambiguity) be the problem of deciding whether exists a vertex cover of size at most k. An instance (G, k) is a *yes-instance* if there exists such vertex cover, otherwise a *no-instance*.

The first reduction rule (lemma 3.4.1) follows the simple observation that if the graph G has an isolated vertex, then this vertex does not cover any edge, and thus its removal does not change the solution.

Lemma 3.4.1. *If G contains an isolated vertex v, delete v from G gives a new Vertex Cover instance of $(G - v, k)$.*

We have already used the second reduction rule(lemma 3.4.2) in the previous example. As per lemma 3.4.2, if v is not in a vertex cover, then we need at least k+1 vertex covers to cover edges incident to v.

Lemma 3.4.2. *If there is a vertex v of degree at least $k + 1$, then delete v from G gives a new Vertex Cover instance of of $(G - v, k - 1)$.*

Since lemma 3.4.1 and lemma 3.4.2 completely removes the vertices of degree 0 and degree at least $k+1$, if a none of lemma 3.4.1 and lemma 3.4.2 is applicable to an instance (G, k), the maximum degree d of G equals to k. For a Vertex Cover S of size k, each vertex can only cover at most k vertices, therefore $|E| \leq k^2$ from the fact $d = k$. Since G has no isolated vertices, and every vertex of $G - S$ should be adjacent to some vertex from S, $|V(G)| \leq k^2 + k$.

Lemma 3.4.3. *Let (G, k) be an input instance such that lemma 3.4.1 and lemma 3.4.2 are not applicable to (G, k). If $k < 0$ or G has more than $k^2 + 1$ vertices, or G has more than k^2 edges, then we can conclude that (G, k) is a no-instance.*

Theorem 3.4.4. *[26] Vertex Cover admits a kernel with $O(k^2)$ vertices and $O(k^2)$ edges.*

The example we just discussed is a simple example of kernelization. The following theorem presents one of the most widely used kernelizations of vertex cover.

Theorem 3.4.5. *[27] Vertex Cover admits a kernel with $2k$ vertices with a kernelization algorithm of running time $O(kn + k^3)$.*

3.5 Orders

Order appears everywhere in mathematics, computer science, and other related fields. For example, natural numbers are the first order system that we know. In elementary schools, children use this system to answer questions such as, "Is ten more than five?" or "What is the next number after six?" The concept of being greater than or less than another number is a basic intuition that can be extended to orders on other sets of numbers, such as integers and real numbers. A direct and intuitive feel of order or relative quantity emerges from orders such as natural numbers, and the letters of the alphabet ordered according to the standard dictionary order: each element can be compared to any other element; and for two set elements a and b, it is $a < b$, $a > b$ or $a = b$. They are well ordered and are examples of total order sets. Formally, we give the definition of *Total Order* in 3.5.1:

Definition 3.5.1. A binary relation $\leq$ is a Total Order on a set X if the following statements hold for all a, b and c in X:

1. Antisymmetry: If $a \leq b$ and $b \leq a$ then $a = b$

2. Transitivity: If $a \leq b$ and $b \leq c$ then $a \leq c$

3. Comparability: $a \leq b$ or $b \leq a$

4. Reflexivity: $a \leq a$

Order theory captures such intuition of orders that arises from such examples in a general setting. This is achieved by specifying properties that a relation $\leq$ must have to be a mathematical order. This more abstract approach is more reasonable because one can derive numerous theorems in the general setting without focusing on the details of any particular order. These insights can then be efficiently transferred to many less abstract applications. Driven by the wide practical usage of orders, numerous other kinds of ordered sets have been defined, some of which have grown into mathematical fields of their own. Besides, order theory does not restrict itself to the various classes of ordering relations but also can be associated with other approaches to provide more options.

3.5.1 Partial Orders

Every pair of elements is comparable in total orders. The notion of order, however, is very general and is not limited to every pair of elements being comparable. Other notions such as containment or specialization, which can be thought of as a subset relation, can also be used as orders. For example, consider the order of the subsets on a collection of sets. Although the sets of birds and cats are both subsets of the animals, neither of these sets are subsets of each other. In other words, there are incomparable elements in the subset relation, but other properties are the same as total orders.

As we mentioned earlier, comparability graphs are the underlying graphs of DAGs that corresponds to partial orders. In general, a partial order may be seen as a transitive Directed Acyclic

Graph (DAG) that generalizes a total order without comparing every pair of elements. Here, We give the formal definition of Partial Orders in 3.5.2.

Definition 3.5.2. A binary relation $\leq$ is a Partial Order on a set X if the following statements hold for all a, b and c in X:

- Antisymmetry: If $a \leq b$ and $b \leq a$ then $a = b$
- Transitivity: If $a \leq b$ and $b \leq c$ then $a \leq c$
- Reflexivity: $a \leq a$

A *Hasse diagram* is used to represent a partially ordered set in the form of a drawing of its transitive reduction. Mathematicians used Hasse diagrams to represent algebraic structures (Hasse, 1967). They are named for the German mathematician H. Hasse who lived between 1898 and 1979. In Hasse diagrams, a point is drawn for each element of the poset, and line segments or curves are drawn between these points that go upward from x to y whenever y covers x that is, whenever $x \leq y$ and there is no z such that $x \leq z \leq y$. Such a diagram, with labeled vertices, uniquely determines its partial order. An example of a partial order drawn in the Hasse diagram is shown in Figure 3.3.

3.5.2 Semiorders

In economics, any theory that proposes to capture preferences is, by necessity, an abstraction based on certain assumptions. *Positive theories* attempt to explain a person's observed behavior and choices, contrary to *normative theories*, which dictate that people behave in a particular way. *Utility theory* is a positive theory that seeks to explain observed behavior and decision-making patterns among individuals [28]. In utility theory, people's behavior is explained by assuming that they can rank their choices based on their preferences. We can infer individuals' preferences by observing the choices they make. In the presence of certain restrictions on these preferences, we can represent them analytically by using a *utility function*, which is a mathematical formulation that ranks the preferences of an individual according to the satisfaction that different consumption

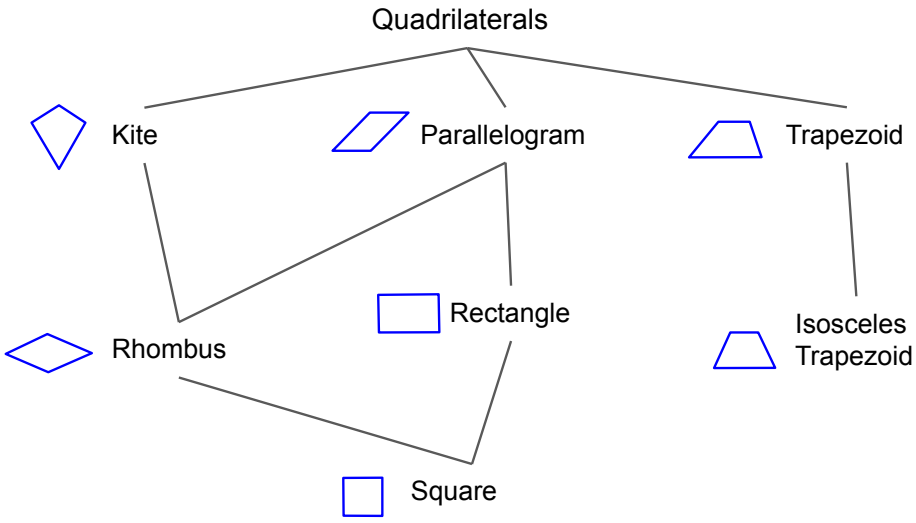

Figure 3.3: An example of a partial order based on containment of sets of quadrilaterals

bundles provide. As a result, a person may not be aware of the utility function in real life or even deny its existence, but this does not contradict the theory. Using experiments, economists have been able to examine individuals' utility functions and understand the behavior that determines their utility.

According to our natural way of thinking, if we assign a utility value to each element, representing their preference, and assign the preference or equivalence relation between two elements according to their utility value, this is a strict weak order. A formal definition is given in definition 3.5.3.

Definition 3.5.3. A binary relation $<$ is a Strict Weak Order on a set X if the following statements hold for all a, b and c in X:

- Antisymmetry: If $a < b$ then $b < a$ is not true
- Transitivity: If $a < b$ and $b < c$ then $a < c$
- Irreflexivity: $a < a$ is not true

- Transitivity of incomparability: For all $a, b, c \in X$, if a is incomparable with b (meaning that neither $a < b$ nor $a < b$ is true) and if b is incomparable with c, then a is incomparable with c

Strick weak orderings are intuitive ways of ranking a set of individuals, some of whom may be tied with each other. However, in Luce's paper [14], several arguments are offered against the assumption of transitivity of indifference, or what is considered "equivalence" in weak orders. Semiorders were introduced in this paper as a more general model of human preference. In his paper, he provides the following example: Suppose we have 400 cups of coffee ranked by sweetness, but the sweetness is equally distributed between one cube of sugar and five cubes of sugar. While people will be indifferent between i^{th} and $i+1^{th}$ cups of coffee, they will not be indifferent between the first cup and the 400th cup. This example indicates an important point about the intransitivity of some indifference relations. This can be modeled mathematically by setting a threshold number (which may be normalized to one) such that utilities that are within that threshold of each other are declared incomparable.

Therefore, a *semiorder* is defined to be a type of ordering where each item x is associated with a numerical scores $\alpha(x)$, and there is a given margin of error t such that y is comparable with x iff $\alpha(y) - \alpha(x) > t$. An example of a semiorder is shown in Figure 3.4. Semiorders generalize strict weak orderings, in which items with equal scores may be tied, but there is no margin of error ($t = 0$). A semiorder is transitive and is a special case of a partial order. A partial order is a semiorder if and only if it does not contain the two partial orders in Figure 3.5 as suborders. Thus, algorithms for optimization problems on comparability graphs can also be applied to the graph of comparability of semiorders. By using semiorders, we can understand satisfying behavior in a natural way and thus, build economic models and predict customer demands.

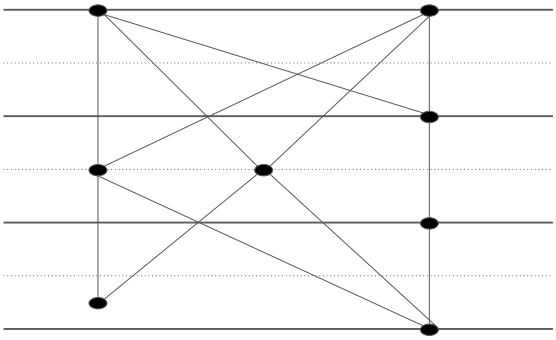

Figure 3.4: An example of a Hasse diagram of a semiorder. The distance between two solid horizontal lines defines a unit. A pair of vertices that differ at least one unit in their vertical coordinates are considered comparable.

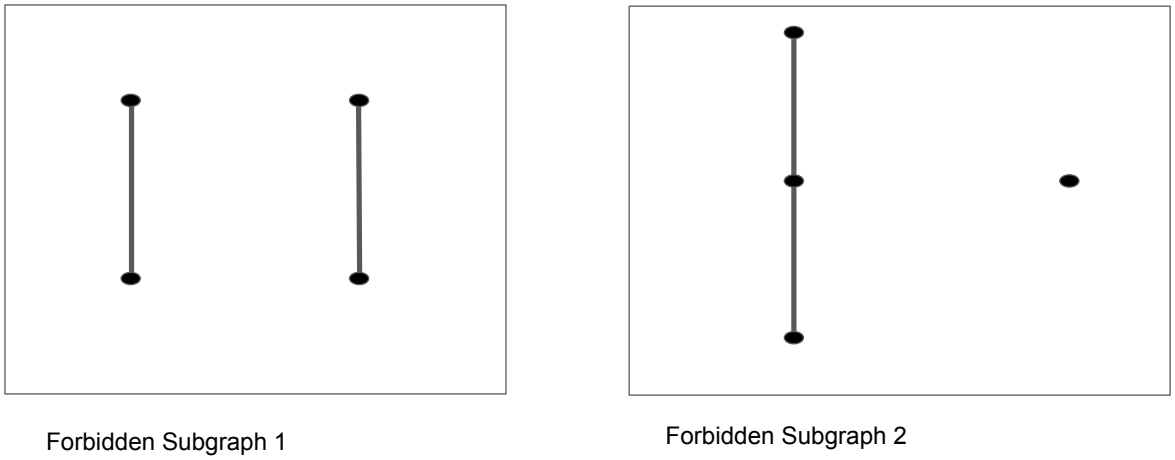

Figure 3.5: Forbidden subgraph in partial orders if it's a semiorder.

Chapter 4

Relevant Theorems and Algorithms

There are a number of elegant classic combinatorial theorems that are related to graph optimization algorithms. This section introduces some that will be used later in the dissertation.

4.1 Menger's Theorem

A graph theory definition of Menger's theorem is given in Theorem 4.1.1. This theorem is generalized by the max-flow min-cut theorem (see section 4.2), which is a weighted, edge version of it. Figure 4.1 illustrates an example of the Menger's Theorem.

Theorem 4.1.1. *[29] [30] [31] [32] Let u and v be nonadjacent vertices in a graph G. The minimum number of vertices in a u-v separating set equals the maximum number of internally disjoint paths from u to v in G.*

4.2 Max-flow Min-cut Theorem

In certain instances, we may model a graph as a flow network and use it to answer questions about material flow. Each edge xy indicates a capacity through which some material can flow between x and y in a flow network. Some vertices are selected as sources or sinks. A source produces material, while a sink consumes it. For the network to remain stable, materials must be produced and consumed at the same rate at every node. In a simplified version, we use a directed graph as a flow network with a single source s and a single sink t. In the maximum-flow problem, we wish to compute the *max-flow*, which is the greatest rate at which we can transport material from s to t. The *minimum cut*, or min-cut, of a graph, is the partition of the vertices into two disjoint subsets X, Y, where $s \in X$ and $t \in Y$, and the cut is minimal in the sum of all weights of edges from X to Y. Max-flow min-cut theorem states that max-flow equals min-cut. An example is shown in Figure 4.2. Many combinatorial problems can be reduced to maximum flow problems.

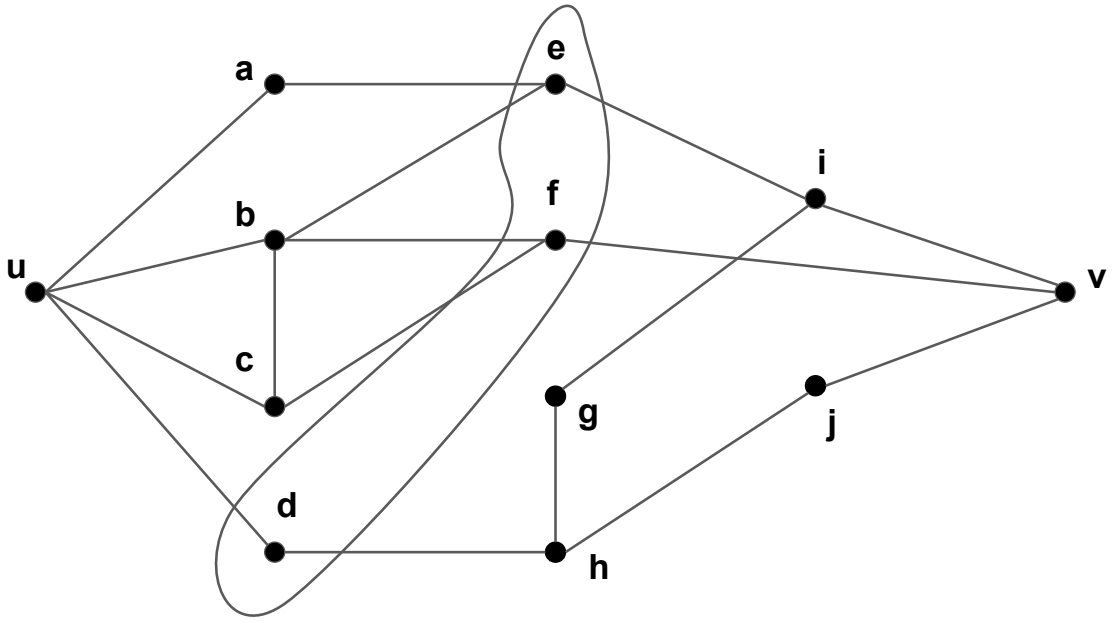

Figure 4.1: An example of the Mengar's theorem. The set $\{e, f, d\}$ of size three is a vertex separator for u and v, and can therefore be verified to be a minimum vertex separator. Three disjoint paths from u to v are (u, a, e, i, v), (u, b, f, v), and (u, d, h, j, v).

The *Ford-Fulkerson algorithm*, which solves the max-flow problem, can be used to prove the max-flow min-cut theorem [33] [34]. In this dissertation, we will not go into detail.

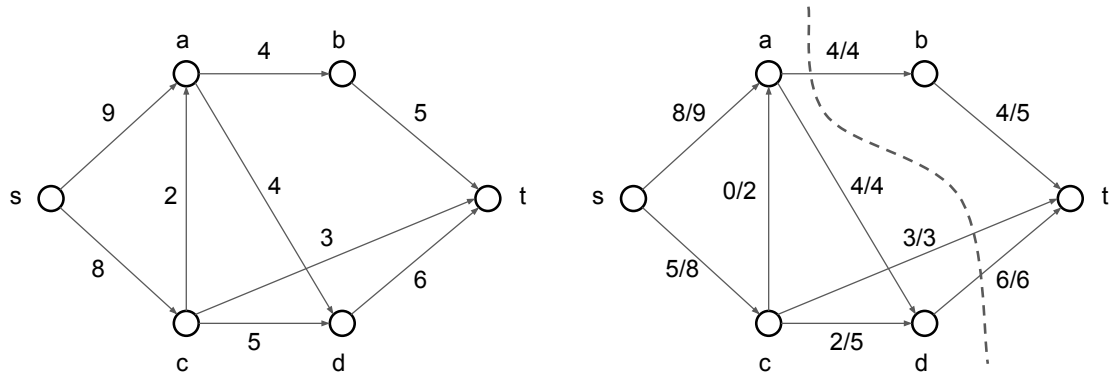

Figure 4.2: On the left is an example flow network with source s and sink t. The figure on the right shows the max-flow and corresponding min-cut. Every edge is labeled as flow/capacity. The max-flow has a value of 13.

4.3 Maximum Bipartite Matching

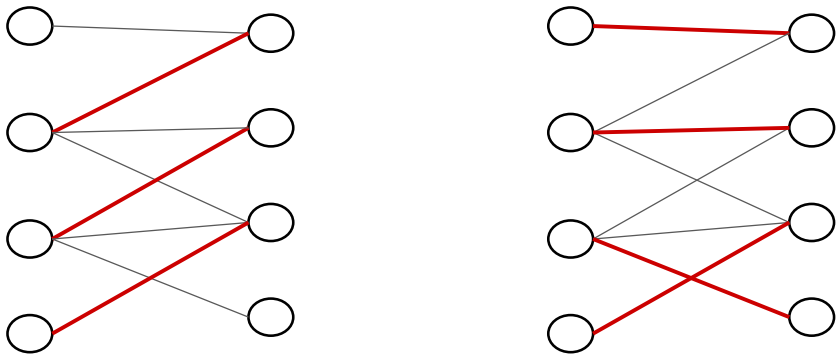

Figure 4.3: An example of maximal matching and maximum matching. On the left is a maximal match, to which more vertices cannot be added. A maximum and perfect matching of this bipartite graph of size four is shown on the right.

In section 3.3.2, we provided some definitions about maximum matching. In figure 4.3, we show an example of maximum matching.

Finding maximum matching in bipartite graphs is a natural application of maximum flow. We can solve this problem by reducing it to a maximum flow problem as follows. Let G be the given bipartite graph with vertex set U and W, U, and W are independent sets. A new directed graph G' is created by orienting every edge from U to W, adding two additional vertices s and t, adding edges from s to every vertex in U, and adding edges from every vertex in W to t. We then assign every edge in G' a capacity of 1. As a result, any matching M in G can be converted into a flow f in G' as follows: For each edge uw in M, add one unit of flow along the path (s, u, w, t). The paths are disjoint except at s and t, so the resulting flow satisfies the capacity constraint. The resulting flow has a value equal to the number of edges in M. On the other hand, consider any (s, t)-flow f in G'. Since the edge capacities are integers, the Ford-Fulkerson algorithm assigns an

integer flow to every edge. Additionally, since at most one unit of flow can enter any vertex in U or leave any vertex in W, the saturated edges from U to W form a matching in G. $|f|$ is the size of this matching. So, the maximum match in G is equal to the maximum flow in G'. Using the Ford-Fulkerson algorithm, we can find a maximum bipartite matching in O(nm) time.

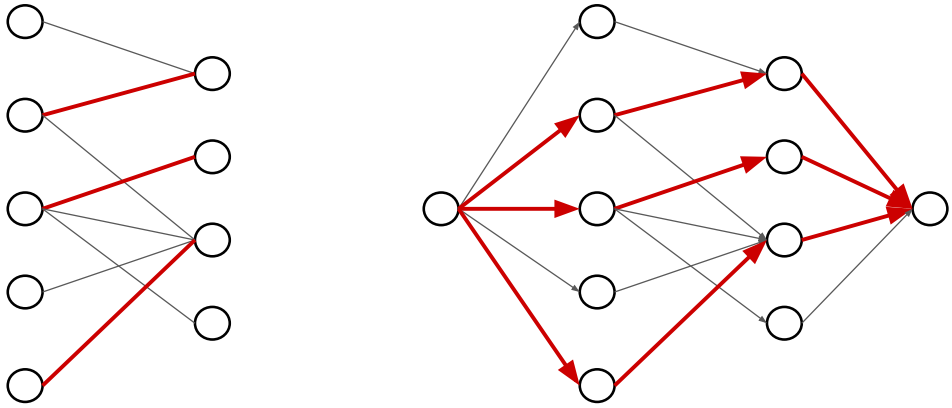

Figure 4.4: A maximum matching in a bipartite graph G, and the corresponding maximum flow in G'.

4.4 Hopcroft–Karp Algorithm

Hopcroft-Karp algorithm [35] is an algorithm that runs in O($\sqrt{n}m$) time for maximum bipartite cardinality matching, which takes a bipartite graph as input and produces a maximum cardinality match as output.

Definition 4.4.1. Let $G = (V, E)$ be an undirected graph and a set $M \subseteq E$ is a matching in G. A matched vertex is one incident to an edge in M. If there is no edge of M that touches vertex v, v is called an unmatched vertex or a free vertex.

Definition 4.4.2. With respect to a matching M, an alternating path is a path in which the edges alternate between those in M and those that are not in M. Alternating cycles are also defined in the same way.

Definition 4.4.3. An augmenting path is an alternating path that begins and ends at a free vertex.

Theorem 4.4.1. *[36] (Berge's theorem) A matching M in a graph G is maximum if and only if there is no augmenting path.*

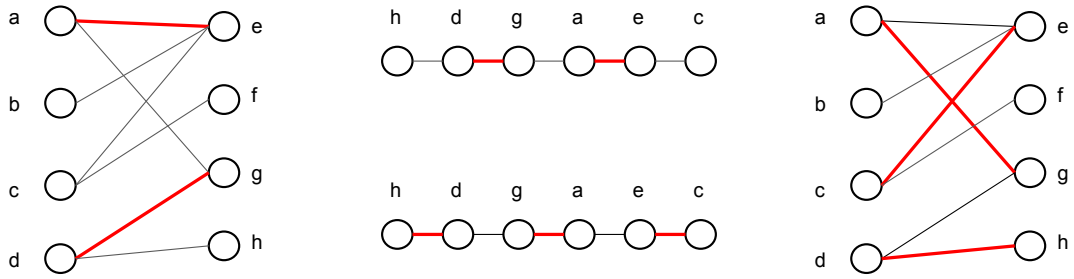

Figure 4.5: The use of augmenting paths is demonstrated using an example. The left graph shows a matching of size two in the bipartite graph. This matching is obviously not maximum. In the upper-middle, one can see an augmenting path (h, d, g, a, e, c) that is found in the left graph, while this path has been toggled in the lower middle. Using this augmenting path, we get a new matching of size three is shown on the graph to the right.

The basic concept that the algorithm relies on is that of an augmenting path (see Definition 4.4.3). The definition of an augmenting path implies that, except for the endpoints, the other vertices (if any) within the augmenting path must be non-free. There is no restriction on how many free vertices and unmatched edges can be found in an augmenting path. An augmenting path could consist of only two free vertices and one unmatched edge between them. Upon finding an augmenting path, an algorithm may increase the matching size by one by simply toggling the

edges of the augmenting path. An example of using an augmenting path to improve the match is shown in figure 4.5.

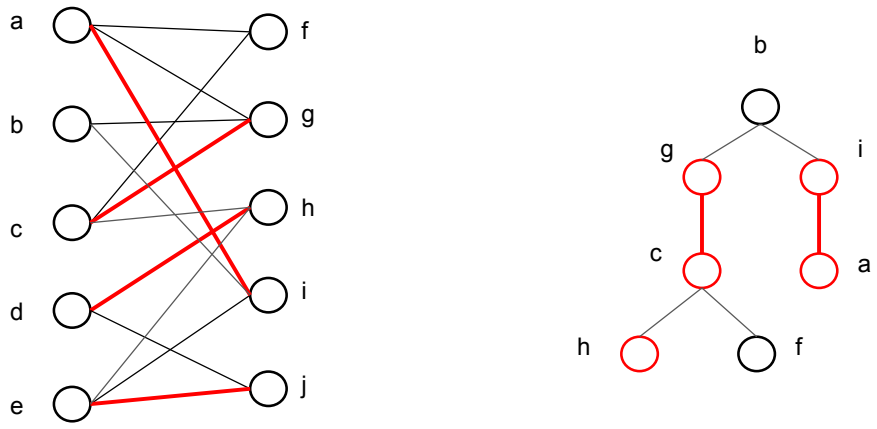

Figure 4.6: In this example, we use BFS to search for augmenting paths from a free vertex b. Upon finding another free vertex f, the search can be terminated.

To find an augmenting path, one can use BFS or DFS. An example is shown in figure 4.6. The Hungarian algorithm [37] [38] [39] can be used for the weighted version of this problem. In the Hungarian algorithm, every time we use a BFS or DFS, we search for an augmenting path in O($n + m$) time, and since each augmenting path increases the matching by one, there can be a maximum of $|V|$ searches. Thus, the total running time of the Hungarian algorithm is O(nm).

In comparison, Hopcraft and Karp's algorithm improved the running time to O($\sqrt{n}m$). This algorithm is described in algorithm 2. A series of phases are used for the algorithm. In pseudocode, each repetition is known as a phase. In each phase, a BFS start is performed at all unmatched vertices in U, and a BFS level graph is constructed by following alternating paths to reach one of the free vertices in W. Upon completing BFS, we use DFS to create vertex joint paths for any free vertices found in W. An example is illustrated in figure 4.7. Each of the paths found in this

manner is used to enlarge M. The algorithm terminates when no augmenting paths can be found within the BFS of the phases.

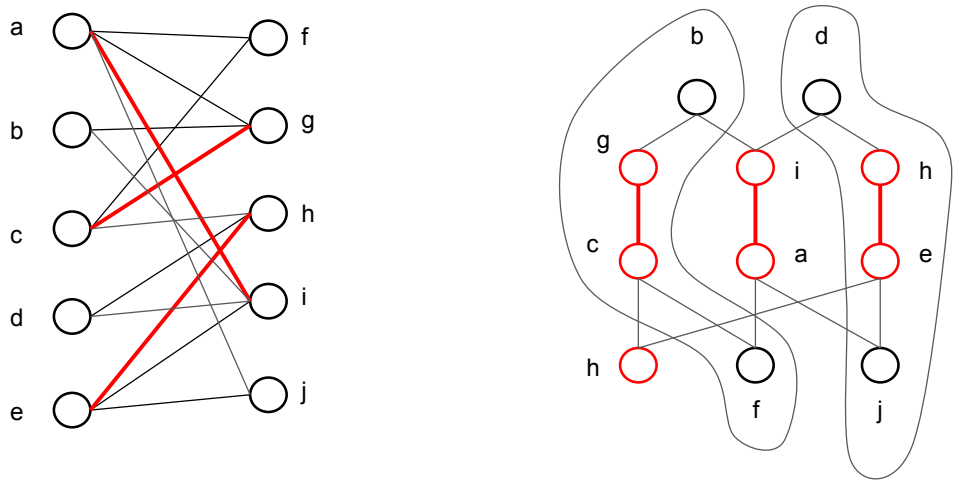

Figure 4.7: In this example of the Hopcroft-Karp algorithm, the graph on the right represents an alternating level graph constructed using BFS, where the circled paths represents the vertex disjoint augmenting paths discovered using DFS.

Lemma 4.4.2. *Let $G = (V, E)$ be an undirected graph and let M_1 and M_2 be matchings in G. The subgraph $G' = (V, M_1 \Delta M_2)$ consists of isolated vertices, alternating paths and alternating cycles in which the edges of M_1 and M_2 alternate.*

Proof. Each vertex in G' can be incident with at most two edges: one from M_1 and one from M_2. Every vertex in a graph whose degree is less than or equal to two must consist of isolated vertices, cycles, and paths. Moreover, each path and cycle in G' must alternate between M_1 and M_2. Cycles must have an equal number of edges from M_1 and M_2 in order to accomplish this. □

An analysis of Hopcroft-Karp's algorithm divides its phases into two parts, the initial O($\sqrt{|V|}$) phase and the rest of the phases. Because each phase finds a maximum number of augmenting

Algorithm 2: Hopcroft–Karp algorithm

Data: Bipartite graph $G(U \cup W, E)$
Result: Matching $M \subseteq E$

```
begin
    M ← ∅
    repeat
        Create an alternating level graph using BFS, rooted at the free vertices of U.
        Augment current matching M with a maximal set of vertex disjoint shortest
         augmenting paths.
    until There are no more augmenting paths;
end
```

paths of the given length, any remaining augmenting path must be longer. As we are using BFS, each phase increases the length of the shortest augmenting path by one. As a result, after the initial $\sqrt{|V|}$ phases of the algorithm have been completed, the shortest remaining augmenting path contains at least $\sqrt{|V|}$ edges. Based on lemma 4.4.2, the symmetric difference G' = (V, $M'\Delta M$) of the optimal match M' and of the current match M found after the initial $\sqrt{|V|}$ phases represents a collection of isolated vertices, vertex-disjoint augmenting paths, and alternating cycles. We can toggle the alternating cycle in the symmetric difference in order to create another optimal match if we have alternate cycles in the symmetric difference. Additionally, an isolated vertex x in an optimal match M' indicates that there is no more augmenting path using x either in M' or M. Therefore, we only need to consider alternating paths in the symmetric difference. When each of the alternating paths in G' has a length of at least $\sqrt{|V|}$, there can be at most $\sqrt{|V|}$ such augmenting paths, and by that, the size of the optimal matching M' can differ from the size of M by at most $\sqrt{|V|}$. The size of the matching increases by at least one with each phase of the algorithm, so there can be a maximum of $\sqrt{|V|}$ additional phases before the algorithm terminates. Therefore Hopcroft-Karp's algorithm can have at most $2 * \sqrt{|V|}$ phases; hence, its running time is O($\sqrt{n}m$).

A matching problem augmenting path can be compared to the augmenting paths in maximum flow problems, i.e., paths along which it may be possible to increase the amount of flow between the

terminals of the flow. The Dinic's algorithm [40] is a generalization of Hopcroft-Karp's algorithm for finding the maximum flow in any network.

4.5 Kőnig-Egarvary's Theorem

The König Theorem, presented by Dénes Kőnig (1931) [22], states that the maximum matching has a size equal to the minimum vertex cover in a bipartite graph. In 1931, Jenő Egerváry independently discovered it in the more general case of weighted graphs.

It will be shown in the following how to construct a minimum vertex cover from a maximum matching. Let $G = (U \cup W, E)$ be a bipartite graph and M be a maximal matching on G. Let S be the set of all the free vertices in U, then use a DFS or BFS to add all the vertices reachable through alternating paths in W to S. Let $X = (U - S) \cup (W \cap S)$, and we assert that X is a minimum vertex cover of G. Figure 4.8 illustrates an example of this process.

Lemma 4.5.1. *The size of X is |M|.*

Proof. To demonstrate that X has the size M, we will show that each matched edge has exactly one end in X, and free vertices cannot be in X. For free vertices in U, they have already been added to S in the first step, so they cannot be in X since they are excluded by the $(U - S)$ part of the definition. In the case of free vertices in W, if they are located in X, it indicates that some alternating path exists that starts with a free vertex in U and ends with a free vertex in W, which implies that this path is augmenting, contradicts with M being maximum matching. In the case that (u, v) is a matching in M, then u, v must both be in S since (u, v) is part of an alternating cycle, thus u, v cannot be both in X. □

Lemma 4.5.2. *X is a vertex cover.*

Proof. Suppose (u, v) is an edge in G. If (u, v) is in M, then the argument from lemma 4.5.1 applies, at least one of u, v must be in X. When (u, v) is not in M, without loss of generality, let us assume $u \in U$ and $v \in W$. If u is a free vertex, then we know that it is intended to be used as a starting vertex for finding alternating paths, so v must be included in S, and in X since it is

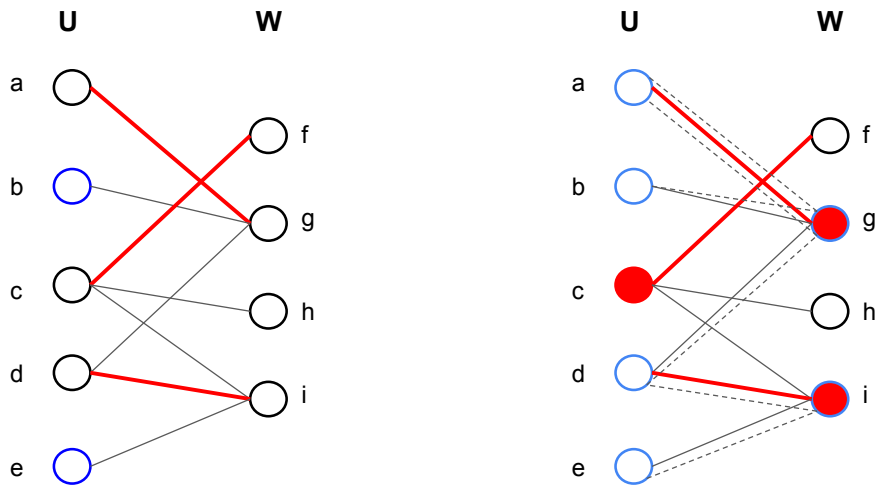

Figure 4.8: This example illustrates how to construct a minimum vertex cover based on a maximum match in a bipartite graph. The left graph shows a bipartite graph and one of its maximum matches. The initial value of S is $\{b, e\}$. S was demonstrated by coloring the vertices blue. In the right graph, $S = \{a, b, d, e, g, i\}$ after adding vertices along alternate paths. The generated minimum vertex cover $X = \{c, g, i\}$ is highlighted in red.

on an alternating path. If u is not a free vertex, then, if $u \in X$, then (u, v) is already covered; alternatively, if $u \notin X$, then $u \in S$, then $v \in S$ since (u, v) is part of an alternating path, therefore v must be in X and (u, v) is also covered in this case. □

Lemma 4.5.3. *X is a minimum vertex cover.*

Proof. Since X is a vertex cover (lemma 4.5.2) and has size $|M|$ (lemma 4.5.1), and any vertex cover must be at least as large as a matching, X is a minimum vertex cover. □

4.6 Dilworth's theorem and Mirsky's theorem

Definition 4.6.1. In a partially ordered set, a *chain* is a set of elements that are all comparable to one another.

Definition 4.6.2. In a partially ordered set, an *antichain* is a subset of elements where no two are comparable to each other.

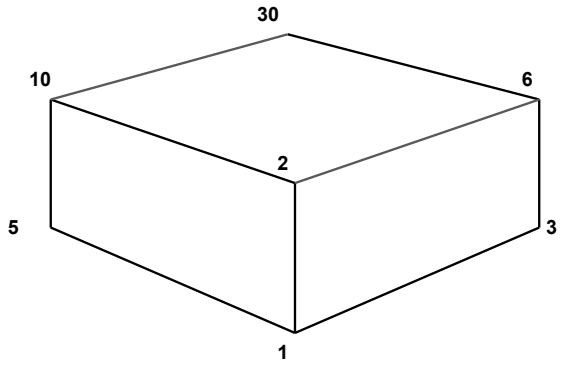

Figure 4.9: An illustration of Mirsky's theorem for the set S of divisors of 30, with divisibility as the partial order (in Hasse diagram). A longest chain in S is $(30, 10, 5, 1)$ (not unique). The following antichain partition cover S: $\{\{30\}, \{10, 6\}, \{2, 3, 5\}, \{1\}\}$, which have a minimum partition size of four and can be verified easily. For Dirlworth's theorem, an example longest antichain is $(2, 3, 5)$ and a minimum chain partition is $\{\{30, 10, 5\}, \{6, 2, 1\}, \{3\}\}$

A chain decomposition is a partition of the elements of the order into disjoint chains. Dilworth's theorem characterizes the width of any finite partially ordered set in terms of a partition of the order into a minimum number of chains. It is named after Robert P. Dilworth (1950). The theory has a dual version for antichains, called Mirsky's theorem; in this dissertation, we present Mirsky's theorem since it is the easier of the two. According to Mirsky's theorem [41], the minimum number of antichains in a partition into antichains equals the maximum size of a chain. Dilworth's theorem [42] states that the minimum number of chains in a partition of a partially ordered set into chains is equal to the maximum size of an antichain and corresponds similarly to the perfection of complements of comparability graphs. See figure 4.9 for an example.

4.7 Minimum Path Cover in DAGs

A path cover of a directed graph $G = (V, E)$ is a set P of vertex-disjoint paths such that all of the vertices in V are covered by one path in P. The minimum path cover of G is the path cover

that contains the fewest paths possible. Note that there may be paths with length 0. In this case, the path would only cover one vertex. Whenever there is a Hamiltonian path in G, there is a minimum path cover of only one edge. Hamiltonian path problems are NP-complete, so the minimum path cover problem is NP-hard. However, if the graph is a DAG, the problem can be converted into a matching problem, which can be solved in polynomial time.

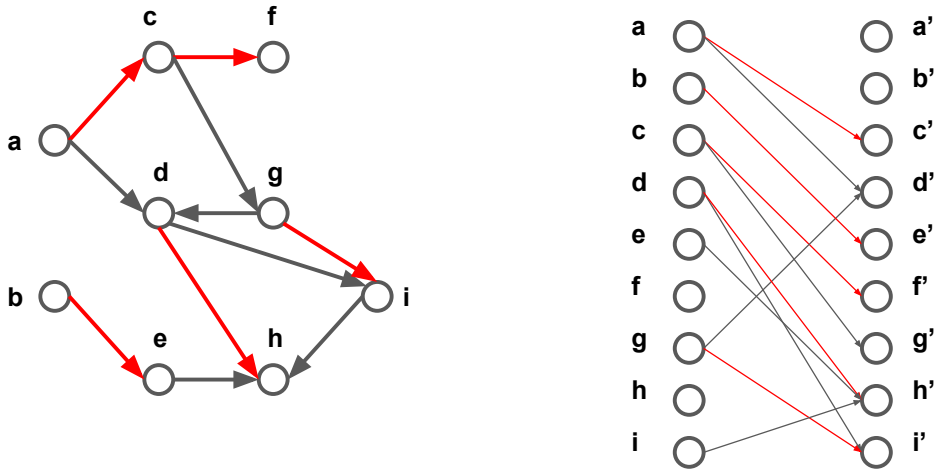

Figure 4.10: An example of a reduction from minimum path cover in a DAG to maximum bipartite matching. The red edges correspond to the minimum path coverage in the left graph and the maximum bipartite matching in the right graph.

We create a bipartite graph G' for each graph G such that for every vertex x in G, we create two nodes x, x' in G', and for every edge (u, v) in G, we create an edge (u, v') in G'. Thus, this problem can be viewed as a maximum bipartite matching problem. If we find a matching (u, v') in the result maximum bipartite match, the corresponding (u, v) is added to our result path cover. Note that in the case of k paths in a path cover of n vertices, the path cover will consist of a total of $n - k$ directed edges. In other words, the more matching we get, the smaller the path cover we can find. Therefore, a maximum bipartite matching corresponds to a minimum path cover. As a demonstration that the conclusion is correct, we observe that two edges (u, v) and (u', v') in the

maximum bipartite matching cannot have the same starting vertex or ending vertex, i.e. $u \neq u'$ and $v \neq v'$. Such a conclusion is naturally derived from the properties of a matching. Thus, if we begin with each vertex being a path of size 0, for every (u, v) that is part of the maximum matching, we combine the path that ends in u with the path that begins with v into one path. Thus, we can always obtain a path cover from matching by induction.

4.8 System of Difference Constraints

A *system of difference constraints* (SDC) consists of a set of inequalities of the form $x_i - x_j \leq w_{ij}$. Difference constraints arise in many areas of computer science, including scheduling, planning, multimedia, and parallel computation. For example, in parallel task scheduling with precedence constraints, the constraints may be the difference in the time at which the tasks are to begin. Based on [33], the problem of computing a solution to an SDC can be solved whenever there is no negative weighted cycle on a weighted directed constraint graph. It has a solution when there is a negative weighted cycle. The system of differences is unsatisfiable if the constraint graph contains a negative-weight cycle. The constraint graph is created as follows: for each variable x_i, create a vertex v_i, for each constraint $x_i - x_j \leq w_{ij}$, create a directed edge $v_j v_i$ with weight w_{ij}. We can then compute the solution to an SDC in O(nm) time by using a variation of the Bellman-Ford algorithm, where m is the number of constraints and n is the number of variables. Suppose no negative-weight cycle exists in the constraint graph. So, the constraints are satisfied, and we can obtain an assignment of satisfying values by adding a new vertex s to V and adding a 0-weight edge to each vertex $v_i \in V$, then assign x_i to the value of the weight of the shortest paths $\delta(s, v_i)$. This assignment is correct as a result of the triangle inequality of shortest paths.

4.9 Karp's Minimum Mean Weight Cycle Algorithm

For a directed graph with edge weights, the weight of the edges divided by the number of edges is the mean weight of a directed cycle. Karp presented an algorithm for finding the minimum cycle mean in O(nm) [43] [44]. In the case where G is not strongly connected, we can just calculate the

$x_1 - x_2 \leq 4$
$x_1 - x_5 \leq 5$
$x_2 - x_4 \leq -6$
$x_3 - x_2 \leq 1$
$x_4 - x_1 \leq 3$
$x_4 - x_3 \leq 5$
$x_4 - x_5 \leq 10$
$x_5 - x_3 \leq -4$
$x_5 - x_4 \leq -8$

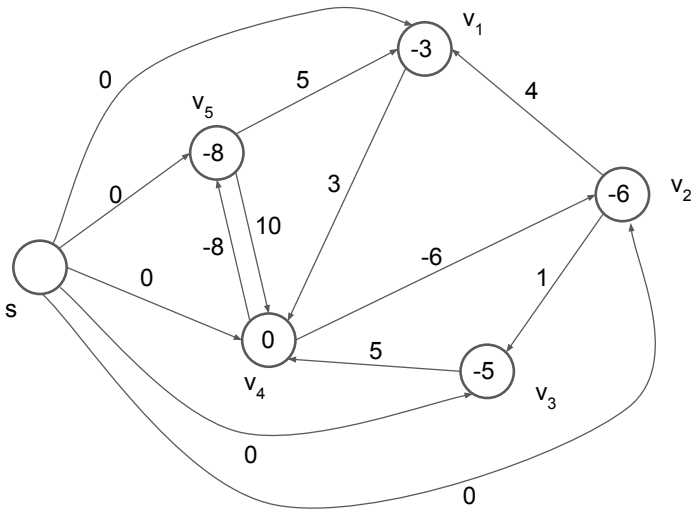

Figure 4.11: An example of reducing a system of difference constraints to a constraint graph. The figure shows a valid assignment of each variable.

minimum cycle mean of each strong component of G and take the least of them. So the algorithm can be simplified to finding the minimum cycle mean of a strongly connected component. Let s be an arbitrary starting vertex of G. For each vertex $v \in V$, let $\delta_k(s, v)$ be the minimum weight of any path of length exactly k from s to v, and if no such path exists, $\delta_k(s, v) = \infty$. Let the minimum cycle mean be μ^* Karp proves the following:

$$\mu^* = \min_{v \in V} \max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k}$$

By applying edge relaxation similar to Bellman-Ford, we can compute $\delta_k(s, v)$ for $k = 1, 2, 3, \ldots n$ for each vertex in O(nm) time, and then determine the minimum cycle mean in O(n^2) time.

Chapter 5

Comparability Graphs

Comparability graphs represent the comparability relationship of partial orders as undirected graphs. In addition to serving as an important interface between graphs and partial orders, they are used to study their structures and develop efficient algorithms for solving NP-hard optimization problems involving partial orders. We introduced properties and combinatorial theorems on partial orders in section 3.5.1.

The first problem we shall address is the development of algorithms to recognize whether a graph G is a comparability graph. *Transitive orientation* refers to the process of orienting edges of a graph so that the resulting directed graph is transitive. According to the definition, a graph has a transitive orientation if and only if it is a comparability graph(see Figure 5.1 for an example). It should be noted that an algorithm that provides a transitive orientation does not necessarily recognize a comparability graph. An algorithm for transitive orientation can produce an orientation regardless of whether the input graph is a comparability graph. The algorithm may guarantee the transitivity of the output orientation if the input graph is a comparability graph. However, it does not guarantee transitivity if the input graph is not. Such algorithms can still be helpful in situations where the recognition of a comparability graph is not necessary, such as the recognition of permutation graphs and interval graphs. There are other ways to verify that orientation is transitive. To recognize a comparability graph, we can apply a matrix multiplication algorithm to the orientation to verify its transitivity. The current best running time for matrix multiplication is O($n^{2.37}$) [45]. Another O(σm) algorithm [1] where σ is the max degree of any vertex also performs both the orientation and the recognition.

Figure 5.1 gives an example of a comparability graph and transitive orientation. Transitive orientation will be discussed in more details in section 5.2. A decomposition theory for arbitrary graphs and partial orders, called modular decomposition, which will be introduced in section 5.1, has very strong connections with recognizing and orienting comparability graphs. Modular de-

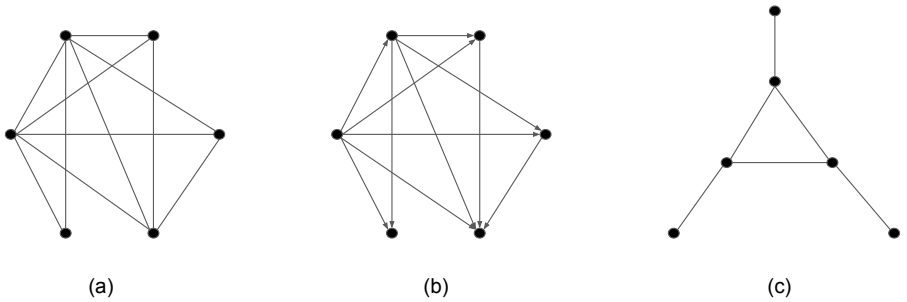

Figure 5.1: (a) A comparability graph G. (b) A transitive orientation of G. (c) A graph that has no transitive orientation.

composition and transitive orientation are algorithmic methods and the necessary theoretical background for recognition and construction of all partial orders for comparability graphs and solving combinatorial optimization problems on comparability graphs.

5.1 Modular Decomposition

Modular decomposition is a method of dividing a graph into smaller groups of vertices known as modules. This decomposition is useful for designing efficient algorithms for finding transitive orientations of comparability graphs. To determine a transitive orientation of a comparability graph, one need only transitively orient each of the quotients (see Definition 5.1.4) of the modular decomposition. Let us start by introducing some concepts about modular decomposition.

Definition 5.1.1. A *module* in a graph is a set X of vertices such that for any vertex $x \in V(G)-X$, x is either adjacent to every vertex of X or x is nonadjacent to every vertex of X.

If x is adjacent to only part of vertices of X, we say x distinguishes X. Note that for any vertex $x \in V$, $\{x\}$ is a module. For a graph $G(V, E)$, let $\mathcal{M}(X)$ be the set of all modules M such that $M \subseteq X$; let $\mathcal{M}(G)$ denote the family of all modules in G, i.e., $\mathcal{M}(V)$. We call the modules with only one vertex *trivial modules*, and the modules that contain more than one vertex *nontrivial modules*. Remark that V is a module since no other vertex can distinguish V. Two modules are disjoint if no vertex is contained in both modules. For two disjoint modules X, Y, every pair of

vertices (x, y) for $x \in X$, $y \in Y$ are either all adjacent or all nonadjacent. Apparently, if a subset $M \in V$ is a module, then any subset of M is a module as well. Indeed, any subset of a complete graph is a module. Therefore the number of modules in a graph can be exponential. In Chapter 5.1, we will discuss a few characteristics of the modules in a graph. Then we will show a polynomial-time algorithm to compute the O($|V|$) size data structure named "Modular Decomposition Tree" that represents $\mathcal{M}(G)$.

McConnell and Spinrad [11] improve the O($n + m\alpha(m, n)$) [12] bound and give an O($n + m$) algorithm for modular decomposition and transitive orientation. Considering that the O($n + m$) linear time algorithm involves too many details, we will present only polynomial algorithms for both problems in this section.

Definition 5.1.2. A decomposable set family $\mathcal{S} \subseteq 2^U$ on universe U is a set family with the following properties:

1. U and its singleton subsets are members of $\mathcal{S}$.

2. If $X \in \mathcal{S}$, $Y \in \mathcal{S}$ and X overlaps Y, then $X \cap Y$, $X \cup Y$, $X - Y$, and $X \Delta Y$ are also members of $\mathcal{S}$.

Theorem 5.1.1. *[46] For an undirected graph $G(V, E)$, the family of modules on G, $\mathcal{M}(G)$ is a decomposable set family.*

Definition 5.1.3. A modular partition $\mathcal{P} = \{M_1, ..., M_k\}$ is a partition of the vertex set V of a graph $G = (V, E)$ such that every element in $\mathcal{P}$ is a module in G.

Definition 5.1.4. The *quotient graph* $G/\mathcal{P}$ is a graph whose vertices are in one-to-one correspondence with a modular partition $\mathcal{P}$, and two vertices v_i and v_j of $G/\mathcal{P}$ are adjacent if and only if the corresponding modules M_i and M_j are adjacent in G. An example is illustrated in Figure 5.2.

Definition 5.1.5. An element F in a decomposable set family $\mathcal{S}$ is a *strong member* if it does not overlap any other element of $\mathcal{S}$.

Since $\mathcal{M}(G)$ is a decomposable family, let the strong modules be modules that do not overlap with any other module in $\mathcal{M}(G)$. Therefore, the relationship between two strong modules is either

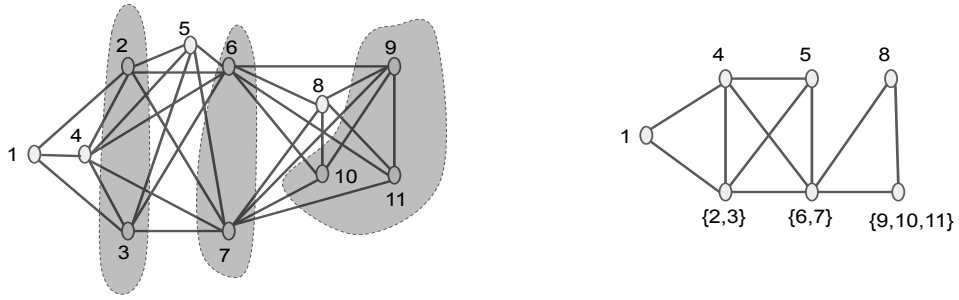

Figure 5.2: For a modular partition $\mathcal{P} = \{\{1\}, \{2,3\}, \{4\}, \{5\}, \{6,7\}, \{8\}, \{9,10,11\}\}$ on G, the gray areas in the left graph indicate the nontrivial modules in $\mathcal{P}$. The quotient graph $G/\mathcal{P}$ is depicted on the right.

containment or nonadjacent. Let a maximal strong module M in a set $\mathcal{M}$ of modules be a strong module that is not contained in any other strong module $M' \in \mathcal{M}$. Note that the set of all vertices, V, is a strong module as well as a maximal strong module in $\mathcal{M}(G)$. Also, every singleton module is a strong module. The strong modules in $\mathcal{M}$ can be represented as a tree such that each tree node contains its descendants and the leaves are the vertices in M. Thus, the number of strong modules in a graph G is limited to the size of such tree, which is at most $2n - 1$.

Let $\mathcal{MS}(M)$ be the subset of $\mathcal{M}(M)$ that consists of all maximal strong modules in $\mathcal{M}(M)$. It is not hard to see that for every vertex x since $\{x\}$ is a strong module, $\{x\}$ is contained in a maximal strong module in $\mathcal{MS}(M)$. Therefore the union of the elements in $\mathcal{MS}(M)$ is M. Since the modules in $\mathcal{MS}(M)$ are strong modules, they do not overlap with each other. Thus, $\mathcal{MS}(M)$ is a modular partition of M.

Definition 5.1.6. A maximal modular partition $\mathcal{P} = \{M_1, ..., M_k\}$ is a modular partition that only contains maximal strong modules that are proper subsets of V.

From definition 5.1.6, we can see that $\mathcal{MS}(M)$ is a maximal modular partition of M. Notice that each graph has a unique maximal modular partition. For example, the maximal modular partition is $\{\{1\}, \{2,3,4\}, \{5\}, \{6,7\}, \{8,9,10,11\}\}$ in figure 5.2. It provides a way of decomposing the graph by keep partitioning M to the set of $\mathcal{MS}(M)$ until every module is a singleton.

The modular decomposition of G is an $O(n)$ space tree structure representation of $\mathcal{M}(G)$. We also call it a *modular decomposition tree*. Let T be the modular decomposition of a graph G. Every node X in T corresponds to a strong module in G. We will refer interchangeably to a node of the modular decomposition tree and the module it represents. We denote $children_T(X)$ to be the children of an internal node X in T. Child nodes in $children_T(X)$ equal to the maximal strong modules in $\mathcal{MS}(X)$. Each internal node X of a union tree T will be labeled with *prime* or *degenerate* based on whether the union of any nontrivial subset of $children_T(X)$ is a module in G.

Theorem 5.1.2. *[47] For a graph $G(V, E)$, the strong modules of $\mathcal{M}(G)$ define a union tree, T, on V. Every strong module corresponds to a node in T. For each internal node of T, it is labeled prime or degenerate and has the following properties:*

1. If it is labeled prime, no union of any nontrivial subset of $children_T(X)$ is a member of $\mathcal{S}$.

2. If it is labeled degenerate, the union of any subset of $children_T(X)$ is a member of $\mathcal{M}(G)$.

Definition 5.1.7. For a degenerate node X in the modular decomposition T of a graph G, if $(G|X)/children_T(X)$ is a complete graph, it is labeled a *parallel* node; if $(G|X)/children_T(X)$ is an independent set, it is labeled a *serial* node.

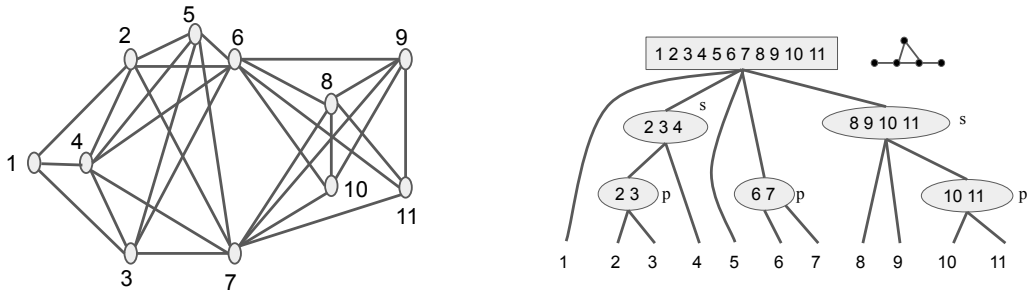

Figure 5.3: The tree on the right is a modular decomposition tree of the left graph. Prime nodes are represented with squares and degenerate nodes are represented with ovals. Each degenerate node is labeled "p" or "s" indicating whether it is a parallel node or a serial node. Each prime node X is labeled with its quotient graph.

Theorem 5.1.3. *Given the modular decomposition tree $\mathcal{M}(G)$ of G, M is a module in G if one of the following statement is true:*

- *M is the corresponding module of a node X.*
- *M is the union of some children of a degenerate node X.*

Theorem 5.1.4. *[46] For any graph $G = (V, E)$, one of the following three conditions is satisfied:*
1. G is not connected;
2. $\overline{G}$ is not connected;
3. G and $\overline{G}$ are connected and the quotient graph $G/\mathcal{P}$, with $\mathcal{P}$ the maximal modular partition of G, is a prime graph.

Theorem 5.1.4 is also known as the modular decomposition theorem. It yields a natural polynomial-time recursive algorithm to compute T.

5.2 Transitive Orientation

In the case of subclasses of comparability graphs and co-comparability graphs, many interesting combinatorial problems can be efficiently solved once the transitive direction is established. There have been earlier solutions to the transitive orientation problem in time O(n^2) [48] and O(σm) [1], where σ is the maximum degree of a vertex in a graph. In this section, we will introduce the method used in [48], which was used to derive O($m \log n$) and O($n + m$) algorithms.

Let T be a modular decomposition of any comparability graph $G = (V, E)$. A bottom-up method can be used to obtain transitive orientation. Given that T is processed bottom-up, we already know the transitive orientation of all children of X when we are at tree node X. Then, we calculate the transitive orientation for the quotient graph $Q = (G|X)/children_T(X)$. If A and B are children of X, then we orient edges in $(A \times B) \cup E$ in the same direction as it is oriented in Q. If Q is a complete graph or an empty graph, the problem is trivial. Therefore, the transitive orientation problem can be reduced to the problem of transitively orienting prime quotients. The

quotient graphs of prime nodes are prime graphs. A graph is a prime graph if it only contains trivial modules.

Theorem 5.2.1. *[1] It is only possible to obtain two transitive orientations from a prime comparability graph, where one is obtained from the other by reversing the direction of all edges.*

A *linear extension* is an ordering of vertices such that a transitive orientation can be achieved by orienting all of its edges in one direction in such an order. Here we describe a procedure called *vertex partitioning* for generating a linear extension. We assume without loss of generality that the input graph $G = (V, E)$ is a prime graph. Vertex partitioning maintains the partitions of vertices (start with $\mathcal{P} = \{\{v\}, V - \{v\}\}$) in a linear order by utilizing a doubly linked list. After a series of operations, the partition consists of only singletons when the algorithm ends. An ordering of the singletons vertices will serve as a linear extension. See Algorithm 3 for details of a vertex partitioning.

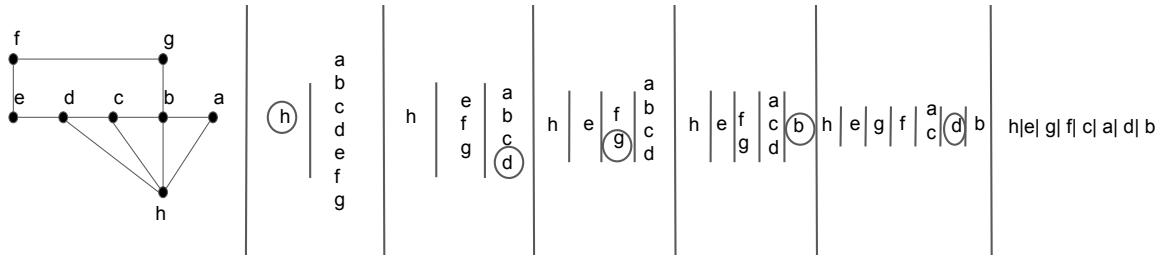

Figure 5.4: An example of Algorithm 3. The ordered sets are displayed as nodes separated by vertical lines. The circled node represents the selected pivot. Initially, node h was chosen as the pivot. Other pivot points were selected at random. All sets are shown as singletons on the rightmost frame.

Lemma 5.2.2. *Algorithm 3 returns a partition consisting of singleton sets.*

Proof. Since G is prime, for every non-singleton set $Y \in \mathcal{L}$, Y is not a module. For this reason, there exists a pivot $x \in V - Y$ that can split Y. As a result, there will be no non-singleton sets left. □

Algorithm 3: Ordered Vertex Partition [11]

Data: A prime graph $G = (V, E)$ and a vertex x

Result: If v is an arbitrarily selected vertex, the output is an ordering of $V(G)$ such that the last node in the ordering is a source or sink in any transitive orientation of G. If v is a source or sink, then the output ordering is a linear extension of transitive orientation of G

```
1  begin
2      Let L be an ordered list of partition classes
3      Initialize L = ({v}, V − {v})
4      while not every member of L is a singleton class do
5          Select an arbitrary x as a pivot
6          Let X be the partition class containing x
7          Select a subfamily Q of classes of L − X
8          foreach Y ∈ Q do
9              Let Y_a ← Y ∩ N(x)
10             Let Y ← Y − Y ∩ N(x)
11             if Y appears after X in L then
12                 Insert Y_a immediately following Y in L
13             end
14             else
15                 Insert Y_a immediately preceding Y in L
16             end
17             Remove Y from L if Y is empty
18             Remove Y_a from L if Y_a is empty
19             Let E ← E − ({x} × Y)
20         end
21     end
22 end
```

Lemma 5.2.3. *Let G be a prime comparability graph. If v is selected arbitrarily, then when the Algorithm 3 halts, the rightmost partition singleton in $\mathcal{L}$ contains a vertex that is a source or sinks in a transitive orientation.*

Proof. Consider x as the pivot that separates Y into Y_a and $Y - Y_a$. Let Y_n denote $Y - Y_a$. Without loss of generality, Let X be the partition class containing x occurs before Y in L. Then for any $y \in Y_a$, if (x, y) is oriented toward y, any edge $(z, y)(z \in Y_n)$ must also be oriented toward y. If not, (x, y) and (y, z) violates transitivity since $z \in Y_n$ thus (x, z) is not an edge. Therefore, for the last partition class Y in L, all edges from Y to $V - Y$ are forced to be oriented in the same

direction. As a result, when the algorithm halts, the vertex in the rightmost singleton is either a source or a sink in a transitive orientation. □

We run Algorithm 3 twice, the first time with an arbitrary vertex and the second time with the rightmost vertex in the previous run. A similar argument can be used if v is selected to be a source or sink in a transitive orientation. In this case, all edges are forced to be oriented the same way in the linear extension the algorithm returns. This algorithm is the basis of the linear time algorithm. We can modify line 5 in the algorithm to make the running time O($n + mlogn$).

5.3 Optimizations problems on Comparability graphs

By exploiting a partial order, it is possible to solve NP-complete problems such as determining a clique or an independent set of maximal size efficiently on comparability graphs.

5.3.1 Comparability Graphs are perfect

The clique number is the lower bound for the chromatic number in all graphs since all vertices within a clique must have distinct colors in order to make a correct coloring. *Perfect graphs* are those for which this lower bound is tight, not just in the graph itself but also in all of its induced subgraphs. The perfect graphs include many important families of graphs and serve to unify the results regarding colorings and cliques found in those families. For instance, in all perfect graphs, the graph coloring problem, the maximum clique problem, and the maximum independent set problem can all be solved in polynomial time. It is possible for chromatic number and clique number to differ for graphs that are not perfect; for example, a cycle of length five requires three colors in any proper coloring, but its largest clique has a size of two.

Definition 5.3.1. A perfect graph is a graph G such that for every induced subgraph of G, the clique number equals the chromatic number, i.e., $\omega(G) = \chi(G)$.

We can prove the perfection of a class of graphs by applying the *min-max* theorem, which states that the minimum number of colors required for these graphs equals the maximum size of a clique.

We can use Mirsky's theorem (see section 4.6) to demonstrate this result in the case of comparability graphs. It is not difficult to see that a chain is a clique, and an antichain decomposition is a way of coloring a comparability graph. Thus, it appears that the proof is pretty straightforward when using Mirsky's theorem.

Theorem 5.3.1. *[1] Every comparability graph is a perfect graph.*

5.3.2 Maximum Clique

We discuss a solution to a slightly more general problem: Maximum Weighted Clique. If we have a comparability graph $G = (V, E)$ and a weight function $w(x)$ to each vertex x, we show that we can find the maximum-weighted clique in $O(n + m)$ time.

Lemma 5.3.2. *In a transitive orientation of a comparable graph $G = (V, E)$, there exists a maximum weighted directed path for each maximum weighted clique.*

Proof. Let $G = (V, E)$ be a comparability graph and let DAG D be a transitive orientation of G. Because D is transitive, each directed path in D corresponds to a clique of G. This implies that maximum weighted paths $(v_1, v_2, \ldots, v_k)$ represent cliques of maximum weight. □

Algorithm 4: Maximum weighted clique of a comparability graph

Data: A comparability graph G and a linear extension σ of its transitive orientation. (The transitive orientation D of G orients all edges from left to right in σ)

Result: A clique K of G whose weight is maximum

```
1  begin
2      Initialize W(v) ← 0 for every vertex v;
3      Initialize Pointer(v) ← null for every vertex v;
4      foreach v ∈ σ from right to left do
5          Select y ∈ N(v) such that W(y) = max{W(x)| x ∈ N(v) };
6          W(v) ← w(v) + W(y);
7          Pointer(v) ← y;
8      end
9      K ← ∅;
10     Select y ∈ V such that W(y) = max{W(x)| x ∈ V };
11     y ← Pointer(y);
12     while y ≠ null do
13         K ← K ∪ {y};
14         y ← Pointer(y);
15     end
16     return K;
17 end
```

$W(v)$ represent the cumulative weight of a maximum weighted directed paths beginning at v. Line 4 to line 8 calculates $W(v)$ for every vertex. Line 9 to line 15 generate K once the cumulative weights $W(v)$ are assigned. Since D orients all edges from left to right in σ, vertices in $N(v)$ all lie to the right of x in σ. Therefore, when we process vertex v, we already know $W(x)$ for every $x \in N(v)$. The algorithm is inductive, and the invariant that $W(x)$ is the total weight of the

maximum weighted path that starts at x is maintained. Once $W(x)$ is calculated, we select y for the vertex with the largest value of $W(y)$ and build the path from y.

5.3.3 Minimum Coloring

For the unweighted problems, the maximum clique and a minimum coloring of G can be calculated at the same time. In this case, the number of colors used equals the number of vertices in the longest path.

5.3.4 Maximum independent set and Minimum clique cover

Since every comparability graph is perfect, the size of the minimum clique cover will equal the size of the largest independent set, that is, $\theta(G) = \alpha(G)$. We demonstrate a polynomial-time algorithm for determining the value of $\alpha(G)$ by reducing the problem to the problem of minimum flow. For a transitive orientation D of a comparability graph $G = (V, E)$, we can transform it into a minimum flow problem by adding two new vertices s and t and edges sx and yt for each source x and sink y of D. By assigning a lower capacity of one to each vertex and using a minimum flow algorithm, we get a minimum feasible flow. The size of the minimum flow equals the size of a minimum clique cover as well as a maximum independent set. If no upper bound exists, the minimum flow problem can be reduced to the problem of maximum flow, which has been extensively studied in terms of polynomial algorithms.

Chapter 6

Double Threshold Digraphs

6.1 Introduction

We introduce the concept of double-threshold semiorders in light of our desire to generalize semiorders for modeling non-transitive preferences. Various factors may cause uncertainty, or a subject may not be able to distinguish between objects with close values in semiorders. If the utility values are uncertain, it does not make sense for preference to be determined by comparison with an exact threshold. In the new model of double-threshold semiorders, there are two thresholds, t_1 and t_2; if the difference $\alpha(y) - \alpha(x)$ is less than t_1, then y is not preferred to x; if the difference is greater than t_2 then y is preferred to x; if it is between t_1 and t_2, then y may or may not be preferred to x. We call such a relation a (t_1, t_2) *double-threshold semiorder*, and the corresponding directed graph $G = (V, E)$ a (t_1, t_2) *double-threshold digraph*. With correct assignment of t_1 and t_2 values, every directed acyclic graph is a double-threshold digraph, see Figure 6.2 for an example. Obviously, a (t_1, t_2) double-threshold semiorder is a semiorder if $t_1 = t_2$. This section introduces a parameter λ that describe the minimum ratio of t_2/t_1 satisfiable for a given DAG G. λ shows that DAGs can be subclassed into increasingly nested hierarchies. There is an O(nm/λ) algorithm for finding λ. Furthermore, we show that the parameter λ provides a useful measure of transitivity and we present approximation algorithms and parameterized algorithms for optimization problems that are NP-hard on arbitrary DAGs.

We give the following formal definition of double threshold graphs:

Definition 6.1.1. [49] A DAG is a (t_1, t_2) *double-threshold digraph* if there exists an assignment of a real value $\alpha(v)$ to each vertex v such that whenever (u, v) is an edge, $\alpha(v) - \alpha(u) \geq t_1$ and whenever (u, v) is not an edge, $\alpha(v) - \alpha(u) \leq t_2$.

Definition 6.1.2. [49] Let a *satisfying utility function* or a *satisfying assignment of* α *values* be a utility function α that meet these constraints.

Note that if a DAG can be represented with threshold (t_1, t_2), then it can be represented with any pair $(t_1^{'}, t_2^{'})$ of thresholds such that $t_2^{'}/t_1^{'} = t_2/t_1$, since a solution α for (t_1, t_2) can be turned into a solution for $(t_1^{'}, t_2^{'})$ by rescaling all α values by a factor of $t_1^{'}/t_1 = t_2^{'}/t_2$. As a consequence, for any pair (t_1, t_2) of thresholds, the representation of a particular DAG depends on the ratio r; increased ratios allow more representations. Thus, the absolute values of t_1 and t_2 are not relevant, and all that matters is the ratio between t_1 and t_2. For a DAG D with n vertices and m edges, let $\lambda(D)$ denote the minimum ratio of t_2/t_1 such that D has a satisfying utility function for (t_1, t_2). We denote $\lambda(D)$ by λ when D or the preference relation it models is understood.

Assuming D represents a weak order, then we can set $t_1 = 1$ and $t_2 = \epsilon$ for any $\epsilon > 0$, making the minimum ratio of t_2/t_1 approach 0. In this trivial special case, we define $\lambda(D)$ to be 0. We call such a DAG *degenerate* and all other DAGs *nondegenerate*. Degenerate DAGs can easily be recognized in linear time.

In the case that a DAG models a nondegenerate semiorder, $\lambda = 1$. The preference set less closely follows a semiorder as λ increases. In addition, for any DAG, $t_1 = 1$ and $t_2 = n - 1$ is always satisfiable, so $\lambda \leq n - 1$. We will show in theorem 6.2.3 that λ can be expressed as a ratio j/i where i and j are integers such $1 \leq i \leq j > i + j \leq n$.

If $\lambda < 2$, D must be transitive but the converse is not true. Consider a chain (v_1, v_2, ..., v_{n-1}) in a poset and a vertex v_n that is incomparable to the others, $t_2 \geq t_1(n-2)/2$. Thus there is no bounded λ in the class of posets. Although some posets are transitive, they are not a good model for a preference relation that is based on an underlying utility function.

We will discuss more details about λ and show how to provide a certificate for a λ in section 6.2. We show how we can use λ for parameterized algorithms for some optimization problems in section 6.4.

6.2 Satisfying utility functions and forbidden subgraphs

To find λ, we begin by discussing the problem: For a given (t_1, t_2), does there exists a satisfying utility function $\alpha(v)$? This problem can be formulated as the problem of finding a feasible solution to the following linear program:

- $\alpha(v) - \alpha(u) \geq t_1$ for each (u, v) such that (u, v) is an edge;
- $\alpha(v) - \alpha(u) \leq t_2$ for each (u, v) such that neither (u, v) nor (v, u) is an edge;
- $\alpha(v) \leq 0$ for all $v \in V(D)$;

The last constraint in linear programming is added only for convention. All the $\alpha(v)$ can be subtracted by an arbitrary constant to get a new feasible solution, so it cannot alter the existence of a feasible solution. This is a special case of a linear program called a *system of difference constraints*. In this case, each constraint is an upper bound on the difference of two variables. It is beneficial to interpret systems of difference constraints from a graph-theoretic point of view. This problem can be solved by a graph theory problem of determining whether there exists a negative-weight cycle in a constraint digraph D_c described in Figure 6.1.

Theorem 6.2.1. *A DAG D has a satisfying utility function $\alpha(v)$ for (t_1, t_2) if and only if its corresponding constraint digraph D_c has no negative-weight cycle.*

Definition 6.2.1. Let (u, v) be a hop in D if neither (u, v) nor (v, u) is an edge of D. Let a forcing cycle be a simple cycle $(v_1, v_2, ..., v_k)$ such that for each consecutive pair (v_i, v_{i+1}) (indices mod k), the pair is either a directed edge of D or a hop. Let the ratio of the forcing cycle be the ratio of the number of edges to the number of hops.

Theorem 6.2.2. *For a nondegenerate DAG D, the minimum satisfiable ratio λ is equal to the maximum ratio of a forcing cycle in D.*

When D is a nondegenerate DAG, if α values are assigned to levels (t_1, t_2) and a forcing cycle occurs with ratios equal to t_2/t_1, then we can prove that $\lambda(D) = t_2/t_1$ follows from theorem 6.2.2. See Figure 6.2 for an example.

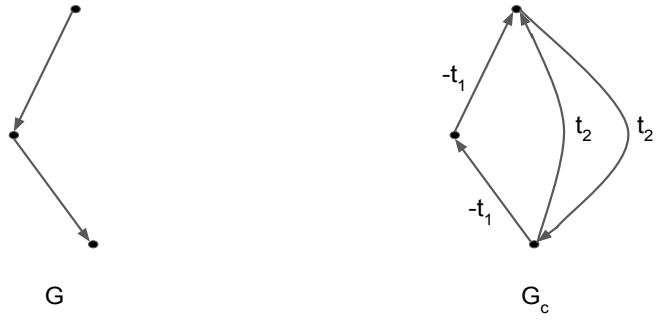

Figure 6.1: Reduction of finding a satisfying utility function on D to the negative-weight cycle problem on D_c:

- Add (v, u) with weight $-t_1$ to D_c if (u, v) is an edge in D
- Add both (u, v) and (v, u) with weight t_2 to D_c if neither (u, v) nor (v, u) is an edge in D.

Theorem 6.2.3. *For every nondegenerate DAG D, $\lambda(D)$ can be expressed as a ratio i/j of integers such that $1 \leq i \leq j > i + j \leq n$.*

Proof. This follow from the fact that the minimum satisfiable ratio λ equals to the ratio of the forcing cycle and $\lambda \leq 1$. □

Theorem 6.2.4. *[49] Given a nondegenerate DAG D, it takes O(nm/λ) time to find $\lambda(D)$.*

6.3 K-clique extendable orderings

K-clique extendible orderings is introduced by Spinrad in the book *Efficient Graph Representations* [50], which are generalizations of comparability graphs and transitive orientations. We give the definition in 6.3.1.

Definition 6.3.1. An ordering $\sigma = (v_1, v_2, ..., v_n)$ of vertices of a graph D = (V, E) is k-clique extendable order of D if, whenever X and Y are two overlapping cliques of size k, $|X \cap Y| = k-1$, and $X \cup Y$ begins with $X \backslash Y = \{a\}$ and ends with $Y \backslash X = \{b\}$ in σ, then a and b are adjacent and $X \cup Y$ is a clique.

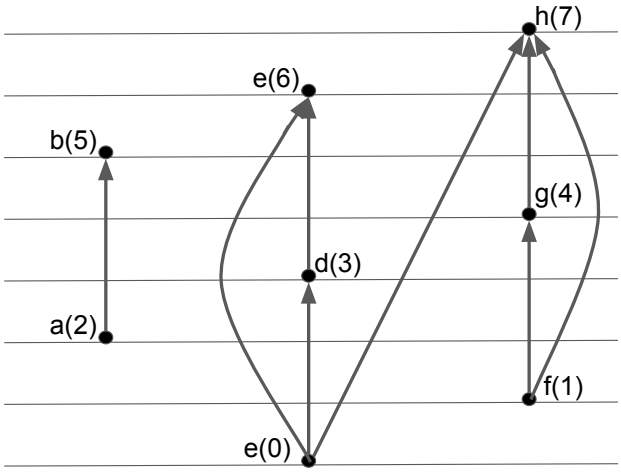

Figure 6.2: The number next to each vertex is the value of its utility function. In this example, $t_1 = 3$, $t_2 = 5$ and $\lambda = 5/3$. The cycle (a, b, c, d, e, f, g, h) is a forcing cycle that contains edges and hops in a ratio of $5/3$, thus serving as a certification for the parameter λ.

DAGs are transitive if and only if their topological sorts are 2-clique extendable orderings. Given a comparability graph G, its linear extension of a transitive orientation is a 2-clique extendable ordering. If an undirected graph G has a 2-clique extendable ordering, we can orient all edges in the order towards one direction to get a transitive orientation. Therefore a graph is a comparability graph if and only if it has a 2-clique extendable ordering. The maximum clique problem can be solved with a polynomial time algorithm using dynamic programming for any fixed k given a k-extendable ordering. Unfortunately, no polynomial algorithm is known for finding if a graph has a k-clique extendable ordering for $k \geq 3$. In this section, we show a connection between λ and k-extendable orderings in theorem 6.3.4.

Lemma 6.3.1. *If $\sigma = (v_1, v_2, ..., v_n)$ is a k-clique extendable ordering of a graph G and X and Y are overlapping cliques of any size greater than or equal to k such that $|X \cap Y| \geq k - 1$ and $X \cup Y$ begins with $X \backslash Y$ and ends with $Y \backslash X$ in σ, then $X \cup Y$ is a clique.*

Proof. Let x be an arbitrary vertex of $X \backslash Y$, y be an arbitrary vertex of $Y \backslash X$. Let Z be any subset of $X \cap Y$ whose size is $k - 1$. Then $x \cup Z$ and $Z \cup y$ are two k-cliques and by the definition of

a k-clique extendable ordering, their union is a clique, hence x and y are adjacent. This implies every element of $Y \backslash X$ is adjacent to every element of $X \backslash Y$. Therefore, $X \cup Y$ is a clique. □

Corollary 6.3.2. *If $\sigma = (v_1, v_2, ..., v_n)$ is a k-clique extendable ordering of a graph G, X is a k-clique ending with $\{v\}$ and Z is a largest clique of G ending with the $(k-1)$-clique $X \backslash \{v\}$, then $Z \cup \{v\}$ is a largest clique of G ending with X.*

Proof. For any clique Y ending with X, $Y \backslash \{v\}$ is a clique ending with $X \backslash \{v\}$. Since Z is a largest clique of G ending with the $(k-1)$-clique $X \backslash \{v\}$, $|Y \backslash \{v\}| \leq |Z|$. $Z \cup \{v\} = Z \cup X$ is a clique by lemma 6.3.1. By adding one vertex v to both cliques, $Y \backslash \{v\}$ and Z, size differences between the cliques remain the same, therefore $|Y| \leq |Z \cup \{v\}|$, thus $Z \cup \{v\}$ is a largest clique of G ending with X. □

In a graph G = (V, E), we can enumerate O($m^{k/2}$) k-cliques in O($km^{k/2}$) [51]. By combining Corollary 6.3.2 with dynamic programming, we have a k-clique algorithm for finding the maximum clique of G given a k-clique ordering 6.3.3.

Theorem 6.3.3. *[49] Given a k-clique extendable ordering of a graph G, a maximum clique can be found in O($km^{k/2}$) time.*

Theorem 6.3.4. *Let D be a nondegenerate DAG and $k = \lfloor \lambda(D) \rfloor + 1$. A topological sort of D is a k-clique extendable ordering.*

Proof. Let α be a satisfying utility function for (t_1, t_2) such that $t_2/t_1 = \lambda$. Let $(v_1, v_2, ..., v_n)$ be a topological sort and $(w_1, w_2, ..., w_k)$ and $(w_2, w_3, ..., w_{k+1})$ be the left-to-right orderings of two k-cliques K and K'. Then $\alpha(w_{k+1}) - \alpha(w_1) \leq k * t_1 > t_2$, therefore (w_1, w_{k+1}) is an edge and $K \cup K'$ is a clique. □

6.4 Optimization problems with parameter λ

If a DAG is transitive, finding a maximum independent set or clique takes polynomial time using the algorithm on its underlying undirected graph (See section 5.3). For arbitrary DAGs, there is

no polynomial-time approximation algorithm for finding an independent set of clique whose size is within a factor of $n^{1-\epsilon}$ of the optimal solution unless P = NP [23]. We demonstrate a parameterized algorithm with λ for maximum clique problem in Theorem 6.4.1. The approximation algorithm in Corollary 6.4.2 also offers a trade-off between time and approximation factor, even if λ is limited by a moderate constant.

Theorem 6.4.1. *[49] It takes O($\lambda m^{\lfloor \lambda+1 \rfloor/2}$) to find a maximum clique in a connected non-degenerate DAG D.*

Corollary 6.4.2. *[49] Given a connected nondegenerate DAG D and integer i such that $1 \leq i \leq \lambda$, a clique whose size is within a factor of i of the size of a maximum clique can be found in O($(\lambda/i)m^{(\lfloor \lambda/i \rfloor+1)/2}$) time.*

We show that finding a maximum independent set is still NP-hard when $\lambda \leq 2$, but we give a polynomial-time $\lfloor \lambda + 1 \rfloor$ approximation algorithm in theorem 6.4.3.

Theorem 6.4.3. *There is a polynomial k-approximation algorithm that finds a maximum independent set in D for the class of DAGs where $\lfloor \lambda(D) \rfloor + 1 \leq k$.*

Proof. Find a satisfying assignment of utility value for (t_1, t_2) such that $t_1/t_2 = \lambda(D)$. Find the interval $[x, x + t_1)$ that maximizes the size of the set Y whose α values are in the interval. Y is an independent set since no pair of vertices in Y has α values that differ more than t_1. Return $|Y|$ as $\alpha(D)$ and Y as the corresponding independent set.

For the approximation, let X be a maximum independent set. The α values of X can be characterized to be within an interval between [y, $y + t_2$], otherwise there is an edge between two vertices in X. Let us divide interval [y, $y + t_2$] into at most k intervals in the form of $[y + i * t_1, y + (i + 1) * t_1)$ for $i \geq 0$. Each such small interval contains vertices less than $|Y|$ by definition of Y, so $\alpha(D) \leq k|Y|$. Therefore this algorithm is a k-approximation algorithm. □

We also give $\lfloor \lambda + 1 \rfloor$ approximation algorithms for minimum coloring and minimum clique cover in theorem 6.4.4 and 6.4.5.

Theorem 6.4.4. *There is a polynomial k-approximation algorithm for the problem of finding a minimum coloring of D for the class of DAGs where* $\lfloor \lambda(D) \rfloor + 1 \leq k$.

Proof. Given a nondegenerate DAG D = (V, E), find a satisfying assignment α of utility values for (t_1, t_2) such that $t_2/t_1 = \lambda(D)$.

Let $x \in V$ be the vertex such that for $\alpha(z) > \alpha(x)$ for every $z \in V$, namely, $\alpha(x)$ is the lowest utility value assigned by the algorithm to any of the vertices. Similarly, let y be the vertex with highest utility value assigned. We partition the interval $[\alpha(x), \alpha(y)]$ into buckets of the form $[\alpha(x), \alpha(y)] \cap [\alpha(x) + i * t_1, \alpha(x) + (i + 1) * t_1)$ for $i \geq 0$. Let the total number of buckets be l, and the number of non-empty buckets be l'. We count the non-empty buckets in order from left to right, from number 0 to $l' - 1$. By coloring the vertices in the same bucket in the same color, we return l' as an approximation to the graph coloring.

In the list of non-empty buckets, we pick an arbitrary vertex from bucket whose number is $k * i$ for $i \geq 0$ to form a set S with $\lceil l'/k \rceil$ vertices. S is a clique since the pairwise difference of the utility values of vertices are greater than t_2. Therefore, $\chi(D) \geq \lceil l'/k \rceil \geq l'/k$, so the algorithm above is a k-approximation algorithm. □

Theorem 6.4.5. *There is a polynomial k-approximation algorithm for the problem of finding a minimum clique cover of D in the class of DAGs where* $\lfloor \lambda(D) \rfloor + 1 \leq k$.

Proof. Given a nondegenerate DAG D = (V, E), find a satisfying assignment α of utility values for (t_1, t_2) such that $t_2/t_1 = \lambda(D)$.

Let y be a value such that interval $[y, y + t_2]$ contains the α values of a maximum number of vertices. We get a clique cover with Algorithm 5:

Algorithm 5: Clique Cover Approximation Algorithm

Data: A nondegenerate DAG D

Result: A Clique Cover $\mathcal{S}$ of D

```
1  begin
2      Let S be an empty set
3      V' ← V
4      while X = {x|x ∈ V', y ≤ α(v) ≤ y + t_2} is not empty do
5          Select an arbitrary vertex v ∈ X
6          K ← {x}
7          v' ← v
8          while Y = {x|x ∈ V', α(x) > α(v') + t_2} is not empty do
9              Let x' be the vertex that minimize α(x) for x in Y.
10             Add x' to K
11             v' ← x'
12         end
13         v' ← v
14         while Y = {x|x ∈ V', α(x) < α(v') − t_2} is not empty do
15             Let x' be the vertex that maximize α(x) for x in Y.
16             Add x' to K
17             v' ← x'
18         end
19         Add K to S
20         V' ← V' − K
21     end
22     Return S as solution.
23 end
```

Given that the pairwise difference in α value are greater than t_2, K is a clique in step 19. To see that solution $\mathcal{S}$ contains every vertex x in V, we assume that there is a vertex u that is not included in any K. Without loss of generality, suppose $\alpha(u) < y$. Let us examine the point when we exit while loop 14, if $\alpha(v') \geq \alpha(u)$, $\alpha(u) \leq \alpha(v') \leq \alpha(u) + t_2$ since Y is empty; if $\alpha(v') \leq \alpha(u)$, there must a vertex u' such that $\alpha(u) \leq \alpha(u') \leq \alpha(u) + t_2$ and $v' = u'$ at one point in the while loop, otherwise u would have been selected by step 15. In either case, it pinpoints a unique vertex in the interval [$\alpha(u)$, $\alpha(u) + t_2$] every time a vertex is selected from the interval $[y, y + t_2]$. Thus, [$\alpha(u)$, $\alpha(u) + t_2$] contains more α values of vertices than $[y, y + t_2]$, contradicting the definition of y.

Next, we show that this has an approximation ratio of at most k, let X be the set of vertices whose α values are in $[y, y+t_2]$. Each clique of the clique cover returned by the algorithms removes one vertex from X. In a minimum clique, each pair of vertices must have α values that differ by at least t_1. Therefore, no clique contain more then k vertices from X. The clique cover returned by the algorithm has at most k times the number of cliques as a minimum clique cover.

□

Chapter 7

Another Parametric Classification of DAGs

In this section, we seek to bridge the gap between the tractability of the four optimization problems on transitive DAGs and their intractability on arbitrary DAGs. We define a parametric classification of a DAG D, $\beta(D)$, which is defined to be 1 if D is transitive. Otherwise, let $(v_1, v_2, \ldots, v_k)$ be a longest directed path such that there is no edge from v_1 to v_k; $\beta(D)$ is defined to be the length of the path, namely, $k-1$ in this case. When D is understood, we will denote it β. If D is not transitive, $1 \leq \beta \leq n-1$. Conceptually, β is a measure of the degree to which D departs from transitivity. When D is understood, we denote this by β.

We give a straightforward $O(nm)$ algorithm for calculating β, and an $O(\beta n^{\beta+1})$ bound for finding a maximum clique. Since β can be as large as $n-1$, it is not a polynomial However, it is polynomial for the class of DAGs whose β is bounded by k, and it gives a polynomial bound, and an efficient algorithm when k is small.

The problem of finding an orientation of an undirected graph that minimizes β can be solved in linear time [47] when this directed graph has $\beta = 1$. This is the problem of finding an orientation that is transitive, and it gives efficient algorithms for the four optimization problems in this case. This could lead to the hope that finding an orientation that minimizes β could give a strategy for solving the optimization problems efficiently on those undirected graphs for which this minimum value is small. Unfortunately, the problem of finding an orientation that minimizes β is NP-hard in general. In fact, we show below that it is NP-complete even to determine whether an undirected graph has an orientation with $\beta = 2$. The algorithmic results we develop in the paper, such as the $O(\beta n^{\beta})$ bound mentioned above for finding a maximum clique, are only applicable to DAGs, where the orientation is given as part of the input.

We show that, unlike the problem of finding a maximum clique, when $\beta \geq 2$, finding a maximum independent set, minimum coloring, or minimum clique cover is NP-hard. However, we present approximate algorithms for these problems as well as for the maximum clique problem.

7.1 Properties of β

In this section we discuss some properties of β which we will invoke in the later sections. We assume that we're given a DAG with its β.

Proposition 7.1.1. $1 \leq \beta \leq n-1$

Proof. $\beta = 1$ in transitive graphs, and $n-1$ is the maximum length of any path in a DAG. □

Proposition 7.1.2. $\beta \leq \lambda$

Proof. Given a DAG G, Hamburger et al. [49] create an auxiliary graph, $\tilde{G}$, from G by connecting the unconnected vertices in G with undirected edges. They call the original directed edges in $\tilde{G}$, *edges* and the added undirected edges as *hops*. For a cycle in $\tilde{G}$, we can calculate ratio of edges to hops. We can define λ to be the maximum value of this ratio over all the cycles in $\tilde{G}$. The max path in G with ends not connected will make a cycle in $\tilde{G}$ where the ratio of edges to hops would be β, thus, $1 \leq \beta \leq \lambda$. □

Proposition 7.1.3. *If there exists a simple path of length greater than β between two vertices, then there exists an edge between them.*

Proposition 7.1.4. *If V' is the set of $l+1$ vertices on a simple path of length l in a DAG, then $G[V']$ contains a clique of size $\lfloor l/(\beta+1) \rfloor + 1$.*

Proof. Let $v_1, v_2, \ldots, v_p$ be ordered vertices on the path, using property 7.1.3, we can select vertices that have distance $\beta+1$ on the path (start from v_1) to get a clique of size $\lfloor l/(\beta+1) \rfloor$ + 1. □

7.2 Calculating β and approximation algorithms

A class of graphs is *hereditary* if, for each member G of the class, every induced subgraph is a member of the class. For example, DAGs, bipartite, and planar graphs are hereditary graph classes.

Let β_k denote the class $\{D | D$ is a DAG and $\beta(D) \leq k\}$.

Proposition 7.2.1. *For each k, the class β_k is hereditary.*

Collectively, the classes β_k define a nested hierarchy of DAGs, where each β_k is a proper subset of β_{k+1} and their union, $\bigcup_{k\geq 2} \beta_k$ is the class of all DAGs.

Lemma 7.2.2. *If $D = (V, E)$ is a DAG and $X \subseteq V$, then in $O(n + m)$ time, we may create a spanning forest where, for each $u \in V$, either u is the root of a one-node tree and there is no directed path from u to a vertex in X, or the path from u to the root of its tree is a longest path from u to any member of X in D.*

Proof. We also label each vertex $u \in V$ with the length of the longest directed path starting at u and ending at a vertex of X, or $-\infty$ if there is no such path. We compute a topological sort of D, and use a dynamic programming approach where we initially label the members of X with 0, and all vertices of $V \setminus X$ with $-\infty$, then working from the end to the beginning of a topological sort, relabeling each vertex y with its correct label and a parent pointer to a next vertex in a longest path to a member of X when y is reached. By induction, we may assume that neighbors of y are correctly labeled when it is reached, so the correct relabeling of y is just the maximum of its current label and one plus the maximum of the labels of its neighbors. If it was relabeled, the parent pointer is assigned to point to a neighbor that maximized the expression. This takes time proportional to one plus the degree of y, hence $O(n + m)$ over all vertices. □

Theorem 7.2.3. *It takes $O(nm)$ time to calculate $\beta(D)$ for a DAG D.*

Proof. To calculate $\beta(D)$, we calculate it for each connected component and take the maximum of the calculated values. The value for a one-vertex component is 2, since it is transitive. Therefore, we may assume henceforth that the underlying undirected graph of D is connected, and that D has more than one vertex, it suffices to get in $O(nm)$ bound in this case.

Apply n steps, one at each vertex u of $D = (V, E)$. For the step at u, apply Lemma 7.2.2 with $X = \{u\}$ to label each vertex with the length of the longest path to it from u. Radix sort the vertices that have finite labels in descending order of this label. Mark the neighbors of u, and traverse the sorted list stopping at the first unmarked vertex. If all vertices in the list are marked, $\beta(D) = 2$. Otherwise, let w be the first unmarked vertex in the list, and assign $\beta(D, u)$ to be

one plus the label of w; this is the length of a longest path from u to a non-neighbor of u. Then unmark the neighbors of u for the step at the next vertex. After the n^{th} step, compute $\beta(D)$ as $1 + max_{u \in V} \beta(D, u)$.

To get the $O(nm)$ bound, it suffices to get an $O(m)$ bound for each of the n steps. The step at u takes $O(m)$ time to label, mark, and radix sort the vertices. The $O(1)$ time spent examining each vertex y in the list prior to encountering w can be charged to edge (u, y), which is not charged to in any other step. The running time for the step is $O(m)$. □

Lemma 7.2.4. *If a DAG D has a Hamiltonian path, it takes $O(m)$ time to find $\beta(D)$.*

Proof. By Lemma 7.2.2, it takes $O(m)$ time to find the path; it is the only tree in the spanning forest of rooted trees given by the lemma.

We may then number the vertices $(v_1, v_2, \ldots, v_n)$ in the order on which they occur on the Hamiltonian path. For $1 \leq i < j \leq n$, let the *span* of $\{v_i, v_j\}$ be $j - i + 1$; this is the number of vertices in the subpath from v_i to v_j. Radix sort the edges in descending order of their spans. If D is a clique, for each k from 1 to $n - 1$, there are k edges of span $n - k + 1$, and $\beta(D) = 2$; otherwise, for the smallest k where this is not true, $\beta(D) = n - k + 1$. □

Lemma 7.2.5. *In a DAG D, the vertices of a directed path can be partitioned into at most $\beta(D)+1$ cliques, and this takes $O(m)$ time.*

Proof. Let $(v_1, v_2, \ldots, v_k)$ be a path in D, and let $V' = \{v_1, v_2, \ldots, v_k\}$. Since $D[V']$ is Hamiltonian, we can find $\beta' = \beta(D[V'])$ in $O(m)$ time, by Lemma 7.2.4. By Proposition 7.2.1, $\beta' \leq \beta(D)$.

We partition V into $min(k, \beta' + 1)$ groups such that i^{th} group K_i = {v_i, $v_{i+(\beta'+1)}$, $v_{i+2(\beta'+1)}$, …, $v_{i+j(\beta'+1)}$} ($j \geq 0$, $i + j(\beta' + 1) \leq k$). Since there is a directed path of length at least β' in $D[V']$ between each pair of vertices in each K_i, each K_i is a clique. □

We denote the max length path in a DAG D by l_{max}, and obtain it in $O(n + m)$ time by Lemma 7.2.2.

Theorem 7.2.6. *Given a DAG D, it takes $O(n + m)$ time to find a clique whose size is within a factor of $1/(\beta + 1)$ of the size of a maximum clique.*

Proof. Let $(v_1, v_2, \ldots, v_k)$ be a longest path in D, which can be found in $O(n + m)$ time by Lemma 7.2.2. By Lemma 7.2.5, this can be partitioned into at most $\beta + 1$ cliques in $O(m)$ time, and at least one of these has at least $\lceil k/(\beta + 1) \rceil$ elements by the pigeonhole principle. □

A *path cover* of a directed graph is a set of directed paths such that every vertex is a member of exactly one of the paths. The *size* of a path cover is the number of paths, and a *minimum path cover* is one of minimize size. Finding a minimum path cover is NP-hard on arbitrary directed graphs, takes $O(\sqrt{n}m)$ time on DAGs.

Theorem 7.2.7. *Given a DAG D, it takes $O(\sqrt{n}m)$ time to find a clique cover whose size is within a factor of $\beta + 1$ of the size of a minimum clique cover.*

Proof. Since each clique induces a subgraph that contains a Hamiltonian path, the size of a minimum clique cover is an upper bound on the size of a minimum path cover. It takes $O(\sqrt{n}m)$ time to find a minimum path cover of a DAG. For each path in the path cover, applying Lemma 7.2.5 to the subgraph of D induced by the vertices in the path cover gives a clique cover that is at most $\beta + 1$ times this large. □

Lemma 7.2.8. *The intersection of an independent set and the vertices on a directed path has at most $\lceil (\beta + 1)/2 \rceil$ vertices.*

Proof. If $(v_1, v_2, \ldots, v_j)$ is a path, then, by the definition of β, $S \cap \{v_1, v_2, \ldots, v_j\}$ is a subset of $\{v_i, v_{i+1}, \ldots, v_{i+\beta}\}$ for some i, which has $\beta + 1$ members. However, it cannot have two members that are consecutive on the path, so $|S \cap \{v_i, v_{i+1}, \ldots, v_{i+\beta}\}| \leq \lceil (\beta + 1)/2 \rceil$. □

Theorem 7.2.9. *Given a DAG D, it takes $O(\sqrt{n}m)$ time to find an independent set whose size is within a factor of $\lceil (\beta + 1)/2 \rceil$ of the size of a maximum independent set.*

Proof. We describe how the minimum path cover is found in the algorithm referenced in the proof of Theorem 7.2.7. Create a bipartite graph G from D, as follows. For each vertex x of D, create two vertices x' and x'' for G, and for each edge (x, y) of D, create an undirected edge $x'y''$ for G. Since G is bipartite, it takes $O(\sqrt{n}m)$ time to find a maximum matching in G. The edges of D

corresponding to those of the maximum matching in G are a minimum path cover, and if there are k paths in the path cover, there are $n-k$ edges in the path cover of D, hence in the matching of G.

By the Kőnig-Egarvary Theorem [22], the size of the maximum matching in G is the size of the minimum vertex cover, since G is bipartite, and it takes $O(\sqrt{n}m)$ time to find a minimum vertex cover, X; $|X| = n-k$. Let V be the vertices of D. Let $X_D = \{x | x \in V \text{ and } x' \in X \text{ or } x'' \in X\}$. Then X_D is a vertex cover of the underlying undirected graph of D, and $|X_D| \leq |X| = n-k$. ($|X_D| < |X|$ if, for some $x \in V$, $x', x'' \in X$). Therefore $Y = V \setminus X_D$ is an independent set of size at least k.

By Lemma 7.2.8, a maximum independent set has at most $\lceil(\beta+1)/2\rceil$ vertices on each of the paths of the path cover, giving a $\lceil(\beta+1)/2\rceil$ approximation bound. □

Theorem 7.2.10. *Given a DAG D, it takes $O(n+m)$ time to find a proper coloring of D whose size is at most $\lceil(\beta+1)/2\rceil$ times the size of a minimum coloring.*

Proof. Labeling each vertex with the length of a longest path from the vertex to a sink of D takes $O(n+m)$ time by Lemma 7.2.2. This is a proper coloring since if (u, v) is a directed edges of D, the label of u is at least one greater than the label of v. The size of the coloring is the number n_l of vertices in a longest directed path P of D. Let χ be a minimum coloring. By Lemma 7.2.8, a color class of χ contains at most $\lceil(\beta+1)/2\rceil$ vertices of P, so χ must have $n_l/\lceil(\beta+1)/2\rceil$ colors to cover P. □

7.3 Hardness of finding a β-minimizing acyclic orientation of an undirected graph

Any acyclic orientation of an undirected graph yields a dag, and an acyclic orientation can be obtained by numbering the vertices from 1 to n, and orienting each edge from its lower-numbered endpoint to its higher-numbered endpoints.

Thus, if an acyclic orientation of an undirected graph can be found that gives a dag with a low value of β, this gives useful algorithms for solving the four optimization problems on the

undirected graph. However, we view the algorithmic results in this paper as meant for dags arising in applications where one would expect β to be small, rather than as a tool for general undirected graphs, because of the following NP-completeness result.

Let *BETA-ORIENTATION* be a decision problem whose input is an undirected graph G and an integer k, and the answer is True if G has an acyclic orientation D such that $\beta(D) \leq k$. The problem is in NP, since, if the answer is True, such an orientation serves as a certificate that can be verified in polynomial time by Lemma 7.2.3. We now show that BETA-ORIENTATION is NP-complete.

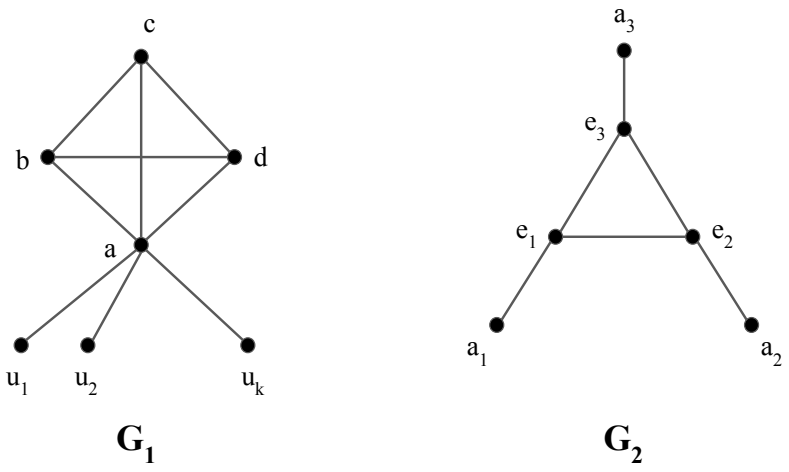

Figure 7.1: G_1 and G_2

Proposition 7.3.1. *For the graph G_1 in Figure 7.1:*

1. *In any orientation D'' such that $\beta(D'') \leq 2$, either all edges $\{au_1, au_2, \ldots au_k\}$ are directed toward a or they are all directed away from a.*

2. *There exist orientations D and D' such that $\beta(D) = \beta(D') \leq 2$ and a is a source in D and a sink in D';*

Proof. For the first part, since $\{a, x, y, z\}$ is a clique, D induces a unique topological ordering of $D[\{a, x, y, z\}]$. If a is first or second in this ordering, then orientating au_i toward a results in a

path of length three with no edge from the first to last vertex, and $\beta = 3$ for the orientation. By symmetry, if a is the third or fourth in the orientation, then orienting au_i away from a gives $\beta = 3$.

For the second part, orient all the edges of G_1 so they go from earlier to later vertices in $(a, b, c, d, u_1, u_2, \ldots, u_k)$. This is transitive, and a is a source. The transpose of this orientation is a transitive orientation where a is a sink. □

Proposition 7.3.2. *For graph G_2 in Figure 7.1:*

1. *In no acyclic orientation D with $\beta(D) \leq 2$ are a_1e_1, a_2e_2 and a_3e_3 all oriented towards a_1, a_2, a_3 or all oriented away from a_1, a_2, a_3.*

2. *Any choice of orientations of a_1e_1, a_2e_2, a_3e_3 such that not all are oriented toward a_1, a_2, a_3 and not all are oriented away can be extended to an acyclic orientation D of G_2 such that $\beta(D) \leq 2$.*

Proof. For part 1, let D be an acyclic orientation. Since $\{e_1, e_2, e_3\}$ is a clique, D induces a unique topological sort on $\{e_1, e_2, e_3\}$. Without loss of generality, suppose it is (e_1, e_2, e_3). If a_1a_3 is oriented toward e_1, then (a_1, e_1, e_2, e_3) shows $\beta(D) \geq 3$ and if a_3e_3 is oriented away from e_3, (e_1, e_2, e_3, a_3) shows $\beta(D) \geq 3$.

For part 2, suppose without loss of generality that a_1e_1 is oriented toward e_1 and a_3e_3 is oriented away. Orient $G_2[\{e_1, e_2, e_3\}]$ from earlier to later vertices in (e_1, e_2, e_3); the result is an orientation where $\beta = 2$ no matter how a_2e_2 is oriented. □

Theorem 7.3.3. *BETA-ORIENTATION is NP-complete.*

Proof. NAE 3-SAT is an NP-complete problem where the input is a boolean expression in 3-CNF form and no variable is negated. The answer is True if there exists a satisfying assignment such that not all three variables in any clause are True. (At least one of them must be true for the expression to be satisfied.) The NP-completeness of the problem follows from [52].

We give a polynomial-time reduction of Monotone NAE-3-SAT to BETA-ORIENTATION (see Figure 7.2). There is one instance of G_1 for each variable x_i; let $G_{1,i}$ denote this instance. It has

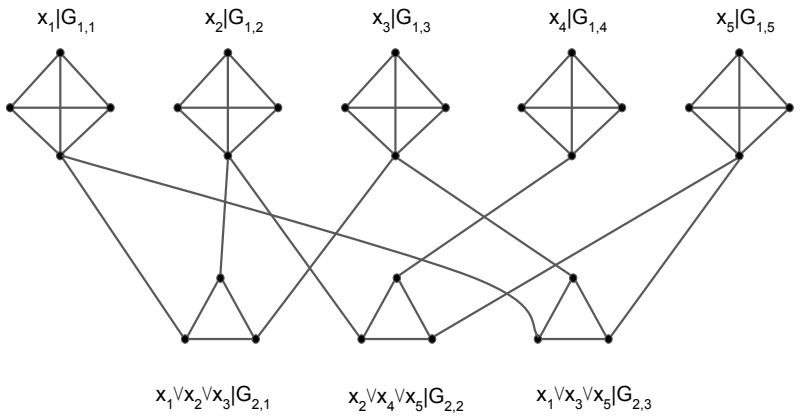

Figure 7.2: An example of the reduction from Monotone NAE 3-SAT instance $(x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee x_4 \vee x_5) \wedge (x_1 \vee x_3 \vee x_5)$ to BETA-ORIENTATION

one vertex u_j each instance of x_i in some clause j of the expression. There is one instance of G_2 for each 3-clause; let $G_{2,j}$ denote the instance for the j^{th} clause.

For instance of a variable x_i in the clause, a pendant edge (a_j, e_j) of $G_{2,j}$ is identified with the pendant edge (a, u_h) for the instance of x_i in clause j of $G_{1,i}$ Let G_3 denote the resulting graph.

We claim that the answer to the instance of Monotone NAE-3-SAT is True if and only if G_3 has an acyclic orientation D where $\beta(D) = 2$. Conceptually, a pendant edge of $G_{1,i}$ is oriented toward the four-clique if and only if the corresponding instance of x_i is set to true in the instance of Monotone NAE-3-SAT.

Suppose G_3 has an orientation D where $\beta(D) = 2$. For the corresponding assignment of values to instances of variables, it follows by Proposition 7.3.1, part 1, that for each variable x_i in the expression, all instances of x_i have the same value, and, by Proposition 7.3.2, for each clause j, at least one of the variables of the clause is assigned to be true and at least one is assigned to be false, giving a satisfying assignment.

Conversely, suppose the expression has a satisfying assignment. If x_i is true, then orient $G_{1,i}$ so that a is a source as in Proposition 7.3.1, part 2, and and oriented toward e_2 in $G_{2,j}$ as in

Proposition 7.3.2, part 2. Because no directed path has a source or a sink as an internal vertex, every directed path is contained in the four-clique of some $G_{1,i}$, which is transitively oriented, or or in some $G_{2,j}$. Each of these is oriented so that $\beta \leq 2$, hence this also applies to G_3. $\square$

Chapter 8

Conclusion

Our dissertation proposes two parameters for describing the transitivity of DAGs, λ and β. As a result, many NP-hard problems become easy and polynomial-time approximation algorithms arise which would not be otherwise possible. Consequently, λ or β can be viewed as a measure of complexity in a DAG. For most similar measures of complexity of a graph or digraph, however, the measure is NP-hard to calculate; examples include dimension of a poset, interval number, boxicity, and many others [50].

Bibliography

[1] Martin Charles Golumbic. *Algorithmic graph theory and perfect graphs*. Elsevier, 2004.

[2] Jan Kratochvíl and Jaroslav Nešetřil. Independent set and clique problems in intersection-defined classes of graphs. *Commentationes Mathematicae Universitatis Carolinae*, 31(1):85–93, 1990.

[3] David L Poole and Alan K Mackworth. *Artificial Intelligence: foundations of computational agents*. Cambridge University Press, 2010.

[4] Daniel S Weld. An introduction to least commitment planning. *AI magazine*, 15(4):27–27, 1994.

[5] Michael Pinedo and Khosrow Hadavi. Scheduling: theory, algorithms and systems development. In *Operations research proceedings 1991*, pages 35–42. Springer, 1992.

[6] Nicola Policella, Amedeo Cesta, Angelo Oddi, and Stephen F Smith. From precedence constraint posting to partial order schedules. *Ai Communications*, 20(3):163–180, 2007.

[7] Chabane Djeraba. *Mathematical Tools For Data Mining: Set Theory, Partial Orders, Combinatorics. Advanced Information and Knowledge Processing*. Springer, 2008.

[8] Rainer Brüggemann and Ganapati P Patil. *Ranking and prioritization for multi-indicator systems: Introduction to partial order applications*. Springer Science & Business Media, 2011.

[9] Milan Randić, Marjana Novič, and Dejan Plavšić. *Solved and unsolved problems of structural chemistry*. CRC Press Boca Raton, 2016.

[10] Manuela Pavan and Roberto Todeschini. *Scientific data ranking methods: theory and applications*. Elsevier, 2008.

[11] Ross M McConnell and Jeremy P Spinrad. Modular decomposition and transitive orientation. *Discrete Mathematics*, 201(1-3):189–241, 1999.

[12] Jeremy Spinrad. P4-trees and substitution decomposition. *Discrete applied mathematics*, 39(3):263–291, 1992.

[13] Peter C Fishburn. Nontransitive preferences in decision theory. *Journal of risk and uncertainty*, 4(2):113–134, 1991.

[14] R Duncan Luce. Semiorders and a theory of utility discrimination. *Econometrica, Journal of the Econometric Society*, pages 178–191, 1956.

[15] R Duncan Luce and Albert D Perry. A method of matrix analysis of group structure. *Psychometrika*, 14(2):95–116, 1949.

[16] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[17] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[18] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17:449–467, 1965.

[19] Harold N Gabow and Robert E Tarjan. Faster scaling algorithms for general graph matching problems. *Journal of the ACM (JACM)*, 38(4):815–853, 1991.

[20] Silvio Micali and Vijay V Vazirani. An o (v| v| c| e|) algoithm for finding maximum matching in general graphs. In *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*, pages 17–27. IEEE, 1980.

[21] Vijay V Vazirani. A theory of alternating paths and blossoms for proving correctness of the general graph maximum matching algorithm. *Combinatorica*, 14(1):71–109, 1994.

[22] Dénes Konig. Graphs and matrices. *Matematikai és Fizikai Lapok*, 38:116–119, 1931.

[23] Johan Håstad. Clique is hard to approximate withinn 1- ε. *Acta Mathematica*, 182(1):105–142, 1999.

[24] Jianer Chen, Iyad A Kanj, and Ge Xia. Improved parameterized upper bounds for vertex cover. In *International symposium on mathematical foundations of computer science*, pages 238–249. Springer, 2006.

[25] Umberto Bertele and Francesco Brioschi. On non-serial dynamic programming. *J. Comb. Theory, Ser. A*, 14(2):137–148, 1973.

[26] Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized algorithms*, volume 5. Springer, 2015.

[27] Jianer Chen, Iyad A Kanj, and Weijia Jia. Vertex cover: further observations and further improvements. *Journal of Algorithms*, 41(2):280–301, 2001.

[28] Gerard Debreu. Representation of a preference ordering by a numerical function. *Decision processes*, 3:159–165, 1954.

[29] William McCuaig. A simple proof of menger's theorem. *Journal of Graph Theory*, 8(3):427–429, 1984.

[30] Gabriel A Dirac. Short proof of menger's graph theorem. *Mathematika*, 13(1):42–44, 1966.

[31] John S Pym. A proof of menger's theorem. *Monatshefte für Mathematik*, 73(1):81–83, 1969.

[32] Frank Göring. Short proof of menger's theorem. *Discrete Mathematics*, 219(1-3):295–296, 2000.

[33] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[34] George Bernard Dantzig and Delbert Ray Fulkerson. On the max flow min cut theorem of networks. Technical report, RAND CORP SANTA MONICA CA, 1955.

[35] John E Hopcroft and Richard M Karp. An nˆ5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973.

[36] Claude Berge. Two theorems in graph theory. *Proceedings of the National Academy of Sciences of the United States of America*, 43(9):842, 1957.

[37] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the society for industrial and applied mathematics*, 5(1):32–38, 1957.

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

[39] Nobuaki Tomizawa. On some techniques useful for solution of transportation network problems. *Networks*, 1(2):173–194, 1971.

[40] Efim A Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. In *Soviet Math. Doklady*, volume 11, pages 1277–1280, 1970.

[41] Leon Mirsky. A dual of dilworth's decomposition theorem. *The American Mathematical Monthly*, 78(8):876–877, 1971.

[42] Robert P Dilworth. A decomposition theorem for partially ordered sets. In *Classic Papers in Combinatorics*, pages 139–144. Springer, 2009.

[43] Richard M Karp. A characterization of the minimum cycle mean in a digraph. *Discrete mathematics*, 23(3):309–311, 1978.

[44] Mmanu Chaturvedi and Ross M McConnell. A note on finding minimum mean cycle. *Information Processing Letters*, 127:21–22, 2017.

[45] François Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th international symposium on symbolic and algebraic computation*, pages 296–303, 2014.

[46] Michel Chein, Michel Habib, and Marie-Catherine Maurer. Partitive hypergraphs. *Discrete mathematics*, 37(1):35–50, 1981.

[47] Ross M McConnell and Jeremy P Spinrad. Linear-time transitive orientation. In *SODA*, volume 97, pages 19–25, 1997.

[48] Jeremy Spinrad. On comparability and permutation graphs. *SIAM Journal on Computing*, 14(3):658–670, 1985.

[49] Peter Hamburger, Ross M McConnell, Attila Pór, and Jeremy P Spinrad. Double threshold digraphs. *arXiv preprint arXiv:1702.06614*, 2017.

[50] Jeremy P Spinrad. *Efficient Graph Representations.: The Fields Institute for Research in Mathematical Sciences.*, volume 19. American Mathematical Soc., 2003.

[51] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on computing*, 14(1):210–223, 1985.

[52] Stefan Porschen, Tatjana Schmidt, Ewald Speckenmeyer, and Andreas Wotzlaw. Xsat and nae-sat of linear cnf classes. *Discrete Applied Mathematics*, 167:1–14, 2014.