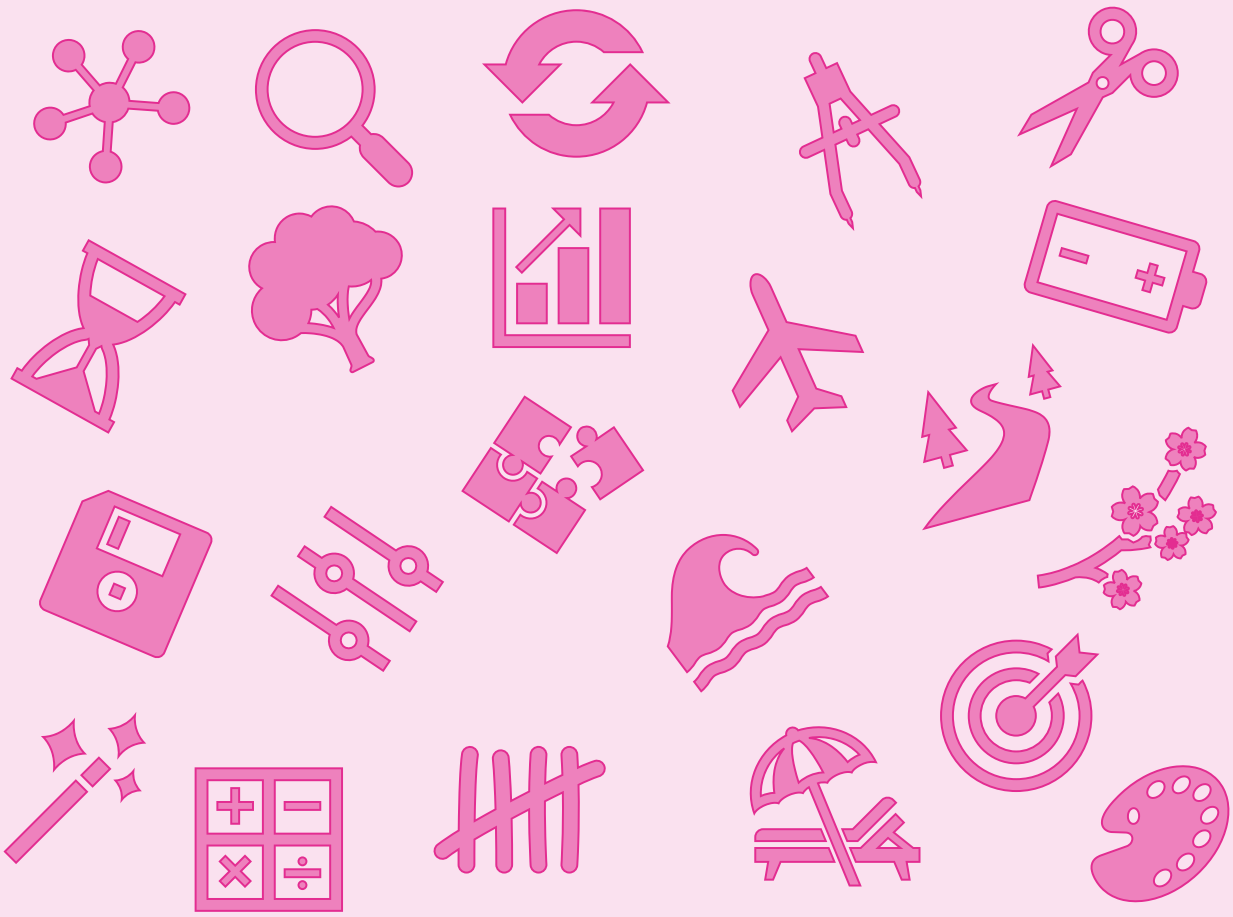
Parameterized Complexity of Restricted Variants of Some Classical Problems



Krisztina Szilágyi

Parameterized Complexity of Restricted Variants of Some Classical Problems

Geparametriseerde complexiteit van beperkte varianten van enkele klassieke problemen

(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor aan de
Universiteit Utrecht
op gezag van de
rector magnificus, prof. dr. H.R.B.M. Kummeling,
ingevolge het besluit van het College voor Promoties
in het openbaar te verdedigen op

woensdag 18 september 2024 des ochtends te 10.15 uur

Krisztina Szilágyi
geboren op 15 januari 1997
te Novi Sad, Servië

Promotor:
Prof. dr. H.L. Bodlaender

Copromotor:
dr. J. Nederlof

Beoordelingscommissie:
Prof. dr. D.N. Dadush
Prof. dr. A. Grigoriev
Dr. H.H. Liu
Prof. dr. D. Paulusma
Prof. dr. F.C.R. Spieksma

The author of this thesis was supported by the project CRACKNP that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 853234).

Acknowledgments

First and foremost, I would like to thank Jesper Nederlof for being such a great advisor. I am very grateful for his patience and guidance over these four years. I really enjoyed working with him and I learned a lot from him. I am thankful to my promotor Hans Bodlaender. What I particularly appreciate is that he always made me feel like an equal, and that he was very approachable. As head of the group, he made sure to create a healthy group dynamic, and encouraged us to maintain a good work-life balance.

I would like to thank my coauthors, Jesper, Hans, Isja, Céline, Carla and Ivan. I enjoyed working with all of them, and I learned a lot from them. Thanks to Alison, Erik Jan, Sukanya and all the Algorithmics TAs for making my teaching tasks enjoyable. I would also like to thank all the members of Algorithms and Complexity group, as well as members of the Geometric Computing group, for all the lunches we had together. Thank you for making me feel welcome in the Netherlands and for introducing me to Sinterklaas. In no particular order, I would like to thank Till for being my first friend in the Netherlands, Lucas for organizing social events, Thekla and Carla for all the academic and life advice, Jelle for the boardgames, Bob for the escape rooms, and Jeremy for all the dad jokes. A special thanks goes to Sukanya and Isja, for being such great office mates. Going through the whole thesis writing and job searching process together made it feel a lot easier and a lot more fun. Thanks for all the discussions, distractions and coffees.

After almost two years of corona, I really enjoyed traveling to conferences and meeting new and old colleagues. I would like to thank all the people responsible for organizing these wonderful events, and for the European Research Council (and Jesper) for funding all my travels. I am also grateful to Robert Ganian for hosting me in Vienna, Dušan Knop for hosting my visit to Prauge and offering me a postdoc position, and Marvin Künnemann for the upcoming visit to Karlsruhe. I learned a lot during these short visits, and they helped me make a step closer towards becoming an independent researcher.

I would like to take this opportunity to wholeheartedly thank my family. I am very grateful to my parents Rozalija and Čaba for their support throughout all these years, and for believing in me more than I believed in myself. Thanks for always being there for me, and thanks for all the advice whose value I started appreciating years later. I would also like to thank my siblings Daniel and Eva, for always cheering me up and for their optimistic outlook on problems. I would also like to thank Fontas, for all the love and support and for always making sure I have enough coffee.

Contents

Chapter 1

Introduction

1.1 Algorithms and Complexity

For thousands of years, humans have used step-by-step instructions to perform calculations, starting from Babylonian mathematicians (around 2500BC) and their calculations of movements of celestial bodies. The term *algorithm* originates from the name of Persian polymath Muhammad ibn Musa al-Khwarizmi, whose books present the first systematic procedures for solving linear and quadratic equations. His works were translated into Latin in the 12th century, and the Latinization of his name *Algorismi* gave rise to the term algorithm.

In the following centuries, procedures for calculating various quantities have been discovered. However, the process of formalizing the notion of algorithms was significantly slower: the invention of one of the first mechanical calculators by Leibniz in the 17th century raised the question of whether a similar machine could calculate truth values of logical formulas. In the 20th century, a lot of progress has been made in this regard, giving rise to several models of computation and thus a better understanding of the concept of computability and algorithms.

One of the most famous models of computation is the *Turing machine*, which is an abstract machine consisting of an infinite sequence of memory cells (memory tape), a "head" which can read and write onto the memory cells, a finite set of states, together with transition rules which determine the next step that the machine will make. This seemingly simple model is regarded as the model which captures the intuitive notion of computability: the *Church-Turing thesis* [117] states, roughly speaking, that any function on natural numbers that can be computed by a human can also be computed by a Turing machine.

Apart from the ability to prove that a problem cannot be solved by an algorithm, these advances in theory of computation allowed us to talk about whether a problem can be solved *efficiently*. We can measure how efficient an algorithm is by looking at the time (and memory) that it used as a function of its input size. At the first glance, the amount of time spent depends on the computer at hand. In many cases however, there is only so much we can achieve by upgrading our computer. For example, consider the Traveling Salesman Problem (TSP), in which we are given n cities

with their pairwise distances, and we are asked to find the shortest possible route visiting each city exactly once, and returning to the city of origin.

The naive algorithm for solving TSP, which tries all $n!$ permutations, becomes infeasible already for $n = 30$: on a today's desktop computer, this algorithm would take roughly 10^{14} years, which is more than the age of our universe. Using a supercomputer would reduce the computation time to about 100 million years.[1] If instead of upgrading our computer we improve our algorithm, we obtain much better results. Namely, the Bellman-Held-Karp algorithm [9, 72] takes about $2^n \cdot n^2$ steps, which means that for $n = 30$, it takes less than two minutes on a standard computer.

Even though it is significantly faster than the naive algorithm, the runtime of the Bellman-Held-Karp algorithm increases very quickly: already for $n = 40$, it takes about two *days*. In other words, algorithms with *exponential* running time are often not feasible for large input instances. *Polynomial* functions grow much slower than exponential ones, i.e. polynomial-time algorithms are significantly more efficient than exponential ones on large input instances. For example, for $n = 1000$, a 2^n time algorithm is 10^{31} times slower than a n^{90} time algorithm. This example illustrates why we can think of problems that can be solved in polynomial time as "much easier" than problems for which we do not have a polynomial-time algorithm.

To facilitate our understanding of difficulty of a problem, we classify problems into complexity classes. One of the most famous such classes is P, the class of problems solvable by a Turing machine (or a modern computer) using a polynomial number of steps. Even though polynomial-time algorithms are sometimes too slow for practical applications, we usually think of P as the class of problems that can be solved quickly. The class NP (somewhat counterintuitive, NP does not stand for non-polynomial, but rather for nondeterministic polynomial) is, roughly speaking, the class of problems whose solution can be verified in polynomial time.

Intuitively, verifying a solution sounds a lot easier than solving a problem. However, the existence of a problem whose solution can be verified in polynomial time, but cannot be solved in polynomial time (i.e. the existence of a problem that is in NP, but not in P) remains one of the fundamental open questions of computer science. This question was first asked by Cook [40] and independently by Levin [87] in the 1970s. In the same papers, the authors proved what is today known as the Cook-Levin theorem, stating that the classical problem of Boolean Satisfiability (also known as SAT) is *NP-complete*, i.e. that any problem in NP can be reduced to it in polynomial time.

Simultaneously to the development of computational complexity, the field of graph theory underwent a significant expansion in the last century, and these two areas became tightly intertwined. A *graph* is a structure consisting of *vertices* and *edges* between them. Although simple, graphs can be used to model many complex real world concepts, from social networks to route planning. Many of these applications require studying graph properties from an algorithmic point of view: it is often necessary to explicitly find a certain substructure in a given graph as opposed to merely showing its existence. Moreover, we are usually interested in an *efficient* algorithm

[1] A standard desktop computer performs about ten billion elementary operations per second, whereas a supercomputer can perform about 10^{17} operations per second, i.e. it is roughly a million times faster.

for finding the substructure.

For example, consider the EULERIAN CYCLE problem (incidentally, this problem is regarded as the origin of graph theory), which asks whether there is a cycle visiting each edge exactly once. This problem can be solved in time linear in the number of edges. In particular, it belongs to the class P. At the first glance, the HAMILTONIAN CYCLE problem, which asks whether there is a cycle visiting each *vertex* exactly once, appears to be similar to EULERIAN CYCLE. However, Karp [79] showed in 1972 that HAMILTONIAN CYCLE is NP-complete. In the same paper, 20 other problems were shown to be NP-complete.

In the last 50 years, many other problems have been shown to be either NP-complete or polynomial-time solvable, however, there are still a few problems whose complexity is not settled. The GRAPH ISOMORPHISM problem is one such example. This problem asks, informally speaking, whether two graphs are the same up to renaming their vertices. Recently, a breakthrough was made by Babai [4], who showed that GRAPH ISOMORPHISM can be solved in quasi-polynomial time. However, it is still not known whether it is in P or NP-complete.

Let us now consider a generalization of this problem, where we ask whether one graph is contained in the other (formally, whether one graph is isomorphic to a subgraph of the other one). This problem is known as SUBGRAPH ISOMORPHISM. In contrast to GRAPH ISOMORPHISM, the complexity of SUBGRAPH ISOMORPHISM has been settled. Namely, it is NP-complete, which can be shown by an easy reduction from CLIQUE, another one of Karp's 21 NP-complete problems.

This contrast between the complexity of SUBGRAPH ISOMORPHISM and GRAPH ISOMORPHISM illustrates one of the fundamental lines of research in complexity theory: namely, the change in complexity when we add or relax restrictions in a problem. In other words, if we have a hard problem, does it have some special cases that are easier? Does it have special cases that are equally hard as the general problem? Or, in another direction, if we have an easy problem, adding which (natural) restrictions makes it hard?

One approach to studying these questions is through the lens of *parameterized complexity*. Informally, we think of parameterized problems as problems where we restrict our input instances to only those where a certain parameter is small. In contrast to classical complexity, where the complexity of an algorithm is a function of the input size, in parameterized complexity we express the complexity as a function of multiple parameters. Let us briefly return to the SUBGRAPH ISOMORPHISM problem, where we ask whether a given k-vertex graph is isomorphic to a subgraph of a given n-vertex graph. In practical applications, k is much smaller than n, so an algorithm whose complexity is exponential in k is much faster than an algorithm whose running time is exponential in n.

Expressing complexity in terms of multiple parameters allows us to have a more fine-grained classification of NP-hard problems. Informally speaking, parameterized problems that belong to the class FPT (fixed-parameter tractable) can be thought of as the "easiest" NP-complete problems. The W hierarchy captures harder problems: problems occupying higher levels are harder (more precisely, at least as hard) than problems on the lower levels. The class XP (the class of problems that can be solved in polynomial time for each fixed value of the parameter) contains the W hierarchy,

and contains most natural problems.

1.2 Thesis overview

The title of this thesis, *Parameterized Complexity of Restricted Variants of Some Classical Problems*, leads us back to the question of how does the complexity of a problem change when adding restrictions to it. In order to describe the hardness of problems more precisely, we view them through the lens of parameterized complexity. In general, we can add restrictions to a problem in two ways.

Firstly, we can consider only input instances with some additional properties, for example by requiring the input graph to belong to a certain graph class. Note that the additional constraints on the input do not always make the problem easier: for example, the 3-Coloring problem (which asks whether the vertices of a graph can be colored using three colors such that there are no monochromatic edges), remains NP-complete even when restricted to 4-regular planar graphs [47]. However, if we require each vertex of the input graph to have degree at most three, the 3-Coloring problem becomes trivial: every such graph admits a 3-coloring [33], moreover, the 3-coloring can be constructed in linear time. Informally speaking, if we start from a hard problem, the challenge lies in balancing between adding too much structure to its input instances (and thus making the problem too easy) and adding too little structure (in which case the problem remains equally hard).

The second way of adding restrictions to a problem is by imposing more requirements on the solution. One such example is integer linear programming: namely, linear programming can be solved in polynomial time [78], but if we require the variables to have integral values, the problem becomes NP-complete [79]. Another common example is adding a connectivity constraint to a graph problem, e.g. Vertex Cover. This way, we obtain the Connected Vertex Cover problem, in which we require the vertex cover to be connected. In case of Vertex Cover, adding the connectivity constraint does not make the problem harder, i.e. Connected Vertex Cover remains FPT [44].

1.2.1 Adding structure to a hard problem

Firstly, we study two hard problems where adding structure to the input makes them easier. In the first case, we start from the Subgraph Isomorphism problem, and add (geometric) structure to it by restricting the problem to unit disk graphs. In the second case, we start from the Set Cover problem, and add (arithmetic) structure to it. Namely, we study a special case of the problem where the universe consists of integers, and the sets correspond to arithmetic progressions.

Chapter 3: Adding geometric structure

Let us return to the Subgraph Isomorphism problem. Recall that in this problem, we are given a k-vertex graph P and an n-vertex graph G (pattern and host graph respectively), and we are asked to determine whether there is a subgraph of G that is isomorphic to P. We will also consider the counting version of this problem,

Counting Subgraph Isomorphism. For graphs P and G, we define $\text{sub}(P,G)$ and $\text{ind}(P,G)$ as follows:

$$\text{ind}(P,G) = \{f : V(P) \to V(G) : f \text{ is injective}, uv \in E(P) \Leftrightarrow f(u)f(v) \in E(G)\},$$

$$\text{sub}(P,G) = \{f : V(P) \to V(G) : f \text{ is injective}, uv \in E(P) \Rightarrow f(u)f(v) \in E(G)\}.$$

Now we define Counting Subgraph Isomorphism as follows:[2]

Counting Subgraph Isomorphism
Input: Graph G, graph P
Task: Compute $\lvert\text{ind}(P,G)\rvert$ and $\lvert\text{sub}(P,G)\rvert$.

Note that Subgraph Isomorphism can be seen as an easier version of Counting Subgraph Isomorphism, where we are only interested in whether $|\text{ind}(P,G)|$ or $|\text{sub}(P,G)|$ are equal to zero. Due to their theoretical and practical importance, Subgraph Isomorphism and Counting Subgraph Isomorphism have been well-studied from many perspectives. The Counting Subgraph Isomorphism problem is an important problem in network analysis. Informally speaking, knowing how often certain patterns appear in a graph gives us a way of measuring the similarity between large graphs. For example, this allows us to determine which graph classes are more suitable for modeling real-world graphs such as protein-protein interaction networks [108].

One of the most recent (Counting) Subgraph Isomorphism solvers is `Glasgow` [96], which is based on constraint programming. As is often the case with hard counting problems, one approach is to resort to approximate counting. Several approximation techniques such as color coding [29], randomized enumeration [103] have been used. Another approach to reduce computational time is to use distributed algorithms [104,120]. A survey of Subgraph Isomorphism solvers can be found in [109]. Machine learning can also be used to speed up classical Subgraph Isomorphism algorithms [89].

From the theoretical perspective, Subgraph Isomorphism generalizes many classical graph problems, such as Hamiltonian Cycle, Longest Path and Clique. As mentioned earlier, it is an NP-complete problem. In practical applications, the pattern size k is often very small [109], thus it is natural to study the parameterization by k. Even very restricted cases of Subgraph Isomorphism (e.g. Clique) are known to be $W[1]$-hard [53] and thus unlikely to have an FPT algorithm.

It is known that Subgraph Isomorphism can be solved in polynomial time when both the pattern and host graphs are trees [95]. However, the problem becomes NP-hard already on graphs of treewidth at most two, which implies that it is not FPT parameterized by treewidth (under standard complexity theory assumptions). Parameterizations by multiple parameters have been studied in [93].

One way to make this problem easier is to add geometric structure to it, i.e. to restrict the input graphs to those with some geometric properties. On planar graphs, the problem can be solved in time $2^{O(k/\log k)}$ [26, 62, 100]. For this algorithm, there is also a matching (conditional) lower bound, namely $2^{o(n/\log n)}$ [26]. The Counting

[2] In this thesis, we consider the *labelled* version of this problem, i.e. we count isomorphisms mapping P to a subgraph of G rather than counting the number of subgraphs of G isomorphic to P.

Subgraph Isomorphism problem on planar graphs can be solved in subexponential time [100].

Apart from planar graphs, another large class of graphs with geometric structure is the class of geometric intersection graphs. These graphs can be represented as a collection of geometric objects whose intersections correspond to the edges of the graph. The simplest such graphs are interval graphs, i.e. intersection graphs of intervals on the real line. It has been shown that Subgraph Isomorphism is $W[1]$-hard (parameterized by k) even in this case [94].

Our results. We consider Counting Subgraph Isomorphism on unit disk graphs, which can be represented as a collection of unit disks in the plane such that edges correspond to intersecting disks. Moreover, we restrict ourselves to bounded ply unit disk graphs, where each point in the plane is contained in at most p disks. Bounded ply allows us to obtain small *separators*, i.e. small sets whose removal disconnects the graph. A *separation* of a graph $G = (V, E)$ is a pair (A, B) such that $A, B \subseteq V(G)$ and there are no edges between $A \setminus B$ and $B \setminus A$. Informally, a separation is a way of splitting a graph into two parts which interact only via a small set of vertices, i.e. the separator. Small separators are a key ingredient for solving Subgraph Isomorphism for several geometric graph classes, e.g. planar graphs [100] and intersection graphs of fat objects [36].

We use a dynamic programming approach, i.e. we break down the problem into simpler subproblems of the same type. Roughly speaking, our dynamic programming table is indexed by a vertical strip in embedding of the host graph, a separation of the pattern graph, and a function describing the "boundary behaviour" of the pattern. The small separators allow us to reduce the number of table entries, since they limit the number of options for boundary behaviour of the pattern. To further speed up our algorithm, we use an approach similar to [100]: we group isomorphic separations together and use a more efficient version of inclusion-exclusion. This allows us to obtain an FPT algorithm. We also prove a matching (conditional) lower bound.

Main result:
A $2^{O(pk/\log k)}n^{O(1)}$ time algorithm for Counting Subgraph Isomorphism on unit disk graphs of ply p.
Key ideas:

- Using **small separators** to reduce the number of possible boundary behaviours of the pattern
- **Grouping isomorphic separations** to reduce the dynamic programming table size
- Applying **efficient inclusion-exclusion** for faster table entry computation

Chapter 4: Adding arithmetic structure

The next problem we study is the Set Cover problem, where we are given a set U of size n (called the universe) together with a collection $\mathcal{S}$ of its subsets, and we are asked to find the smallest number of sets in $\mathcal{S}$ that cover U (i.e. whose union equals U). This problem is one of Karp's 21 NP-complete problems, and it is also known to be $W[2]$-complete parameterized by the solution size k [45].

A related problem (and another one of Karp's 21 NP-complete problems), Exact Cover, additionally requires the sets in the set cover to be disjoint. The Exact Cover problem can be used to model various tiling problems, as well as Sudoku puzzles. Knuth's Algorithm X [82] describes the natural algorithm for solving this problem. Namely, the algorithm deterministically chooses an element, then nondeterministically chooses a set that covers it (i.e. branches over all possible choices) and deletes the newly covered elements. Several versions of Exact Cover were studied in [15], where a $2^n n^{O(1)}$ time algorithm was described.

The greedy algorithm for Set Cover, which picks in each step the set that covers the most uncovered elements, can be shown to be a $\Theta(\log n)$-approximation. In fact, it is in some sense the best polynomial-time approximation: a $(1-\varepsilon)$-approximation of Set Cover is NP-hard for every $\varepsilon > 0$ [51]. Since Set Cover has many practical applications, including crew scheduling [98, 118], data mining [114] and information retrieval [111], many exact and approximation algorithms have been developed. The problem can be formulated as a linear program in a natural way, so linear programming techniques and heuristics can be applied to it. One of the most effective approaches for exact Set Cover algorithms is the branch-and-bound method [35]. This method consists of breaking the problem into several subproblems (branch) and uses certain rules to cut off some branches (bound).

From the graph perspective, an instance of Set Cover can be seen as a hypergraph, where vertices correspond to elements of the universe and hyperedges to the sets in $\mathcal{S}$. In this context, the Set Cover problem becomes the Edge Cover problem on the corresponding hypergraph. While Edge Cover can be solved in polynomial time on graphs (i.e. 2-regular hypergraphs) [65], on general hypergraphs it becomes $W[2]$-complete. Set Cover generalizes several classical graph problems such as Vertex Cover and Dominating Set. While Vertex Cover is FPT, the Dominating Set problem remains $W[2]$-complete on general graphs [45].

Several problems related to Set Cover have been studied from a number theoretic and extremal combinatorics perspective. Erdős introduced the notion of covering systems in the 1950s. Recall that an arithmetic progression (AP) is a sequence of numbers such that the difference between two successive elements is constant. A *covering system* is a set of (infinite) arithmetic progressions that cover the set $\mathbb{Z}$ (i.e. whose union is equal to $\mathbb{Z}$). An open problem, posed by Erdős [56], was whether the smallest difference of an arithmetic progression in a covering system can be arbitrarily large. This question was answered negatively about 50 years later by Hough [74].

Our results. In this thesis, we study a "finite version" of covering systems. Namely, we study the special case of Set Cover where the universe U is a set of integers, and the sets in $\mathcal{S}$ correspond to (finite) arithmetic progressions containing only elements of

U. Formally, we study the Cover by Arithmetic Progressions problem (CAP), defined as follows:

Cover by Arithmetic Progressions (CAP)
Input: Set X of n integers, integer k
Task: Is there a set of k APs that are fully contained in X and whose union equals X?

We also study the Exact Cover by Arithmetic Progressions problem (XCAP), a special case of Exact Cover:

Exact Cover by Arithmetic Progressions (XCAP)
Input: Set X of n integers, integer k
Task: Is there a set of k APs that are fully contained in X, and which partition X?

Heath [71] showed that both of these problems are weakly NP-complete. Restricting ourselves to this special case gives us more structure to work with. On a high level, our algorithms guess (i.e. branches over all possibilities) for each element of the universe which AP covers it. The crucial insight is the way in which we exploit the additional structure of the problem to reduce the number of guesses. Namely, we use the result of Crittenden and Vanden Eynden [42], stating that if k infinite arithmetic progressions cover the set $\{1, 2, \ldots, 2^k\}$, then they cover the whole set of positive integers.

Roughly speaking, our recursive algorithm for CAP branches over all possibilities of covering the first k^2+1 previously uncovered elements. By the pigeonhole principle, two of these elements will be covered by the same AP. However, these two elements might not be consecutive, i.e. their difference might be a multiple of the difference of the corresponding AP. We use the above structural result to help us reduce the number of possibilities for the difference of the AP, leading to an FPT algorithm.

The XCAP problem is intuitively harder than CAP: in contrast to CAP, in XCAP we cannot assume that all APs in the solution are inclusion-maximal. In other words, we might want to "interrupt" an AP to start another one. This is reflected in our algorithm by maintaining two sets for each AP, the set of "trully covered" elements (those that certainly belong to the AP) and the set of those that are "potentially covered" (those that belong to the AP unless we interrupted it earlier). Applying a more involved version of the tools used in the algorithm for CAP, we obtain an FPT algorithm for XCAP (albeit slower than the one for CAP).

Main result:
A $2^{O(k^2)}n^{O(1)}$ time algorithm for CAP and a $2^{O(k^3)}n^{O(1)}$ time algorithm for XCAP.
Key ideas:

- CAP, XCAP: **Branching algorithm** which guesses which AP covers an element
- CAP, XCAP: Using a **structural result by Crittenden and Vanden Eynden** to reduce the number of branches
- XCAP: Distinction between **trully and potentially covered elements**, reflecting usage of non-maximal APs

1.2.2 Adding constraints to an easy problem

In the second part of this thesis, our starting point is an "easy" problem. We add two natural requirements on the solution, making the problem harder. We consider the Flow problem, which can be informally described as follows. A *flow network* consists of vertices (two vertices, source and sink, play a special role) and edges, where each edge has a capacity. We are asked to maximize the amount of flow going from the source to the sink, subject to certain constraints (for a precise definition, see Section 2.2 or standard textbooks). One famous generalization of Flow is the Min Cost Flow. In this problem, in addition to a flow network, we are also given costs for each edge. The problem asks to find a maximum flow of lowest cost (for a precise problem statement, see Section 2.2).

Apart from the natural application to logistics, Flow and Min Cost Flow can be used to model various problems, from Shortest Path to Multiple Object Tracking [28]. As opposed to the previously discussed problems, Flow can be solved in polynomial time: the famous Edmonds-Karp algorithm solves it in time $O(|V||E|^2)$, where $|V|$ and $|E|$ are the numbers of vertices and edges respectively.

Since all their constraints are linear, both Flow and Min Cost Flow can be formulated as linear programs. Dantzig [49] used the simplex method to obtain the first pseudo-polynomial algorithm for Min Cost Flow. Over the years, faster algorithms have been developed using a combinatorial approach [66, 67]. Combining continuous optimization techniques with the existing graph theoretic approaches lead to further improvements [48,113]. Using the Interior Point Method together with efficient graph data structures, Chen et al. [39] gave an almost-linear time algorithm for Flow and Min Cost Flow: Namely, their algorithm solves these two problems in time $m^{1+o(1)}$ with high probability, where m is the number of edges and capacities and costs are polynomially bounded.

In this thesis, we study the Flow problem with an additional all-or-nothing constraint, i.e. we require the flow on each edge to either be zero or equal to the edge capacity. We also study the Min Cost Flow problem with an additional connectivity requirement.

In this case, we require the flow on each edge to either be zero or equal to the edge capacity. Formally, the All-or-Nothing Flow problem is defined as follows:

All-or-Nothing Flow
Input: A flow network (G, s, t, cap), and an integer r
Task: Is there an all-or-nothing flow of value exactly r?

The above problem is known to be NP-complete [3]. The parameterization of this problem by pathwidth and treewidth leads us to two recently introduced complexity classes, namely XNLP and XALP.

The class XNLP was introduced by Elberfeld [54] (where it was named $N[fpoly, flog]$) and Bodlaender et al. [23]. This class is defined as the class of parameterized problems which can be solved by a nondeterministic Turing machine in FPT time (i.e. in time $f(k)n^{O(1)}$ for some computable function f) and logarithmic space (i.e. in space $g(k)\log n$ for some computable function g). It captures problems that are "above" $W[t]$: Namely, XNLP-hardness implies $W[t]$-hardness for all t [23]. The class XALP can be seen as a tree analogue of XNLP, i.e. problems that are XNLP-hard parameterized by pathwidth are often XALP-hard parameterized by treewidth. Formally, the class XALP contains problems that can be solved in FPT time and logarithmic space (same as the class XNLP), using an additional stack.

Our results. The All-or-Nothing Flow problem was shown to be XNLP-complete with pathwidth as parameter [18] and XALP-complete with treewidth as parameter [22]. In this thesis, we show XNLP-hardness of this problem for planar graphs parameterized by outerplanarity (informally, outerplanarity is the number of "layers" of a planar graph). We reduce from the Binary CSP problem paramterized by pathwidth, which was shown to be XNLP-hard in [23]. The Binary CSP problem can be seen as a generalization of Graph Coloring and List Coloring, where each vertex has a list of allowed colors, and the edge constraints consist of pairs of allowed colors for their endpoints.

Given an instance of Binary CSP with bounded pathwidth, we construct an instance of All-or-Nothing Flow as follows. Firstly, we transform the path decomposition of the input graph to a nice path decomposition, i.e. a sequence of Introduce, Forget, Swap and Add-Edge operations which construct the graph step by step. For each of these operations, we describe a gadget corresponding to it. Each vertex in the All-or-Nothing Flow instance will have a label, i.e. it will correspond to one vertex of the Binary CSP graph. The flow through a vertex will correspond to choosing a color for its label.

The key insight is the way in which we verify whether an edge constraint is satisfied. Namely, in order to check whether the constraint of an edge uv is satisfied, we will use a gadget combining flows from a vertex with label u and a vertex with label v. In order to "decouple" these two flows afterwards, we exploit the properties of *Sidon sets*. A Sidon set is a set where each pair of its elements has a different sum. This allows us to ensure that, after the edge constraint check, we get exactly the same outgoing flows.

We use the All-or-Nothing Flow problem as a starting point for several other hardness results, such as Capacitated Vertex Cover, (Red-Blue) Dominating Set and Target Set Selection. Additionally, we improve two hardness results on Binary CSP. Namely, we show that Binary CSP is XNLP-complete on $k \times n$-grids and XALP-complete parameterized by outerplanarity. We use the hardness of Binary CSP to show hardness of Scattered Set, a problem that generalizes Independent Set.

Main result:
All-or-Nothing Flow is XNLP-hard parameterized by outerplanarity.
Key ideas:

- **Reduction from Binary CSP** parameterized by pathwidth (an XNLP-hard problem)
- Working with **nice path decompositions** to construct the graph via a sequence of simple operations
- Using **Sidon sets** to allow uncoupling of two flows entering the same edge

Chapter 6: Adding a connectivity constraint[3]

We study the Min Cost Flow problem with an additional connectivity constraint, namely we require the underlying graph of the flow (i.e. the graph induced by the edges with nonzero flow) to be connected. We call this problem Connected Flow, and we define it as follows (for a more detailed definition, see Section 2.2):

Connected Flow
Input: Directed graph $G = (V, E)$, $D \subseteq V$, $\mathsf{dem} : D \to \mathbb{N}$, $\mathsf{cost} : E \to \mathbb{N}$, $\mathsf{cap} : E \to \mathbb{N} \cup \{\infty\}$
Task: Find a flow $f : E \to \mathbb{N}$ such that for every $v \in D$ we have $\sum_{u \in V} f(u, v) = \mathsf{dem}(v)$ and the value $\mathsf{cost}(f) = \sum_{e \in E} \mathsf{cost}(e) f(e)$ is minimized.

The Connected Flow problem generalizes the famous Travelling Salesman Problem (TSP), in which we are given a graph with edge weights and we are required to find the cycle of minimum weight visiting each vertex of the graph exactly once. This well-studied problem is often used as a benchmark for optimization methods. One of the best studied heuristic approaches is the Ling-Kernighan (LK) method, introduced in [88]. Informally speaking, this approach starts from a (nonoptimal) cycle, and improves it by replacing parts of the cycle that are "out of place". Algorithms based on the LK method have recently been surpassed by those using another approach, namely stem-and-cycle, which was introduced in [105].

Moreover, Connected Flow generalizes an extension of TSP, called Many Visits TSP (MVTSP). In MVTSP, we are required to visit each vertex a number of

[3] The results in Chapter 6 also appeared in the thesis of Céline Swennenhuis [115].

times equal to its demand. The MVTSP problem was introduced by Cosmadakis and Papadimitriou [41], and in the last few years several variations have been studied. Kowalik et at. [85] improved the analysis of the MVTSP algorithm from Berger et al. [13], resulting in a $O^*(4^n)$ time exponential space algorithm. A $O(7.9^n)$ time polynomial space algorithm has also been developed [85]. A 3/2-approximation algorithm for the metric Many Visits Path TSP is given in [11]. In this case, the cost function is a metric, and we are required to find a path rather than a cycle visiting each vertex the required number of times. A generalization of MVTSP involving multiple salesman has also been studied [12].

Our results. We show that adding the connectivity constraint to Min Cost Flow makes the problem NP-hard, even in a very restricted case (namely, the case when we have two demand vertices). We also present parameterized algorithms for solving Connected Flow, which can be seen as an illustration of a "rounding" technique, i.e. a graph analogue of solving integer linear programs (ILP). Namely, a common approach to solving ILPs is to first solve their relaxed (i.e. fractional) version, and use it as a starting point for constructing an integral solution. In our setting, we first solve the relaxed version of Connected Flow, i.e. the Flow problem, and then modify the solution to fulfill the connectivity requirement.

We present an FPT algorithm for Connected Flow parameterized by the size of vertex cover, as well as a polynomial kernel for MVTSP for the same parameterization. We also study the parameterization by treewidth. We present an XP algorithm for Connected Flow, using standard techniques for dynamic programming on tree decompositions. We give a matching (conditional) lower bound for the special case of MVTSP, using a modification of the reduction from 3-CNF-SAT to Hamiltonian Cycle [46].

Main result:
An FPT algorithm for Connected Flow parameterized by vertex cover.
Key ideas:

- **Dynamic programming** table indexed by connected blocks of vertex cover and in/out degrees of some vertices
- **Approximating connected flow by flow** to reduce the table size

Organization

In Chapter 2, we introduce several important parameterized complexity notions and notation that will be used throughout the thesis. The next chapters correspond to the following papers:

- **Chapter 3:** J. Nederlof, K. Szilágyi. *Algorithms and Turing kernels for detecting and counting small patterns in unit disk graphs.* In International Confer-

ence on Current Trends in Theory and Practice of Computer Science (SOFSEM 2024), pp. 125-138. Springer Nature Switzerland, 2024.

- **Chapter 4:** I. Bliznets, J. Nederlof, K. Szilágyi. *Parameterized algorithms for covering by arithmetic progressions.* In International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2024), pp. 413-426. Springer Nature Switzerland, 2024.

- **Chapter 5:** H. L. Bodlaender, K. Szilágyi. *XNLP-hardness of Parameterized Problems on Planar Graphs.* In International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2024). Springer International Publishing, 2024. (to appear)

- **Chapter 6:** I. Mannens, J. Nederlof, C. Swennenhuis, K. Szilágyi. *On the parameterized complexity of the connected flow and many visits TSP problem.* In International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2021), pp. 52-79. Springer International Publishing, 2021.

We finish with some concluding remarks in Chapter 7.

Chapter 2

Preliminaries

In this chapter, we briefly introduce the parameterized complexity framework. We describe one of the central concepts in this area, namely *treewidth*. This graph parameter can be thought of as a measure of similarity between the given graph and a tree. We also discuss lower bounds, i.e. ways of proving that a given problem *cannot* be solved in certain running time. In the second section of this chapter, we introduce notation that will be used throughout the thesis.

2.1 Parameterized Complexity

In this section, we introduce standard parameterized complexity notions and results. This section is based on [45], where a more detailed introduction to parameterized complexity can be found.

Let us now formally define parameterized problems. For a finite set Σ, we define Σ^* as the collection of all (finite) strings that can be made using the elements of Σ.

Definition 2.1. *A* parameterized problem *is a language $L \subseteq \Sigma^* \times \mathbb{N}$, where Σ is a fixed finite alphabet.*

The parameterized analogue of P, i.e. the class of "easy" parameterized problems, is the class of fixed-parameter tractable problems (FPT).

Definition 2.2. *A parameterized problem is* fixed-parameter tractable (FPT) *if it can be solved in time $f(k)n^{O(1)}$, where f is a computable function of the parameter k and n is the size of the input.*

Informally, we think of parameter k as small, so we can "afford" an e.g. exponential dependency on k, but not an exponential dependency on n.

2.1.1 Treewidth

Maximum Weight Independent Set. Consider the following problem: we are given a map with labels (names of countries, regions, cities etc.). Showing all labels

at once would lead to a very cluttered and unusable map, so we have to decide which ones to show. In particular, the labels that we show should not overlap with each other, and we should prioritize more important labels (e.g. names of big cities as opposed to names of small villages).

This problem can be seen as a MAXIMUM WEIGHT INDEPENDENT SET (MWIS) problem. Namely, we can create a conflict graph whose vertices correspond to labels, and we connect two vertices by an edge if the corresponding labels overlap. We assign a weight to each vertex, which corresponds to the importance of its label (e.g. population size). The special case when all weights are equal is known as MAXIMUM INDEPENDENT SET (MIS). On general graphs, MIS is NP-hard [79], and can be solved in time $O(1.2^n)$ [119], where n is the number of vertices of the input graph.

However, when we look at the special case of MWIS where the input graph is a tree, the problem becomes significantly easier. Namely, it can be solved in linear time using the following dynamic programming approach. For a vertex v, let $w(v)$ be the weight of v and let T_v be the subtree rooted at v. We define $A[v,0]$ as the MWIS in T_v that does not contain v, and $A[v,1]$ as the MWIS in T_v. We can calculate $A[v,0]$ and $A[v,1]$ recursively (starting from the leaves) as follows. Let $v_1,\dots,v_q$ be the children of v. We have

$$A[v,0] = \sum A[v_i,1];$$

$$A[v,1] = \max\left\{A[v,0], w(v) + \sum A[v_i,0]\right\}.$$

To prove the correctness of the above recursive formulas, we need several observations. Firstly, the graphs T_v and $G-T_v$ "interact" only via v, i.e. there are no edges between them. Also, there are no edges between the subtrees $T_{v_1},\dots,T_{v_q}$, so if we take the maximum independent sets in each T_{v_i}, their union will still be an independent set.

Treewidth. The large discrepancy in the running time for trees and general graphs raises the question whether we can obtain a faster algorithm for a larger class of "tree-like" graphs. The parameter *treewidth* captures this notion of "tree-likeness". Informally, a tree decomposition of a graph G consists of a tree whose nodes have associated *bags* (subsets of $V(G)$). The bags are chosen in a way that allows us to mimic the above properties of trees. Formally, a tree decomposition is defined as follows.

Definition 2.3. *A* tree decomposition *of a graph $G=(V,E)$ is a pair $(\{X_i \mid i\in I\}, T=(I,F))$, with T a tree and $\{X_i \mid i\in I\}$ a collection of subsets of V, such that*

1. $\bigcup_{i\in I} X_i = V$;
2. *For each edge $\{v,w\}\in E$, there is an $i\in I$ with $v,w\in X_i$; and*
3. *For all $v\in V$, the set $I_v=\{i\in I \mid v\in X_i\}$ forms a connected subtree of T.*

The width *of a tree decomposition $(\{X_i \mid i\in I\}, T=(I,F))$ is $\max_{i\in I}|X_i|-1$, and the* treewidth *of a graph G is the minimum width of a tree decomposition of G.*

Note that the treewidth of a tree is equal to one. The notion of tree decompositions allows us to extend the above approach to a wider class of graphs. Given a graph G with a tree decomposition, for each node t we define X_t to be the corresponding bag, and $V_t = \cup_{t'} X_{t'}$, where t' goes over all nodes in the subtree rooted at t (including t). For each node t and each subset $S \subseteq X_t$, we define

$$A[t, S] = \text{max weight of } S' \text{ such that } S \subseteq S' \subseteq V_t,\ S' \cap X_t = S$$

$$\text{and } S' \text{ is an independent set.}$$

Informally, the set S describes the behaviour of the independent set in the bag X_t. The key observation is that the vertices in V_t interact with other vertices only via X_t.

Nice tree decompositions. In order to further facilitate computing the table entries, we introduce the notion of *nice tree decompositions*:

Definition 2.4. *A rooted tree decomposition* $(T, \{X_t\}_{t \in V(T)})$ *is* nice *if the following conditions are satisfied:*

- *For the root* r *of* T*, we have* $X_r = \emptyset$*. For each leaf* ℓ *of* T*, we have* $X_\ell = \emptyset$*.*
- *Each node* t *that is not a leaf is one of the following types:*
  - ***Introduce node****:* t *has exactly one child* t' *and* $X_t = X_{t'} \cup \{v\}$ *for some vertex* $v \notin X_{t'}$*;*
  - ***Forget node****:* t *has exactly one child* t' *and* $X_t = X_{t'} \setminus \{w\}$ *for some vertex* $w \in X_{t'}$*;*
  - ***Join node****:* t *has exactly two children,* t_1 *and* t_2 *such that* $X_{t_1} = X_{t_2} = X_t$*.*

This allows us to easily compute the dynamic program table entries bottom up (i.e. from leaves to the root). If the input graph has treewidth k, we obtain an algorithm for MWIS that runs in time $2^k \cdot k^{O(1)} \cdot n$ (for more details, we refer to Section 7.3 of [45]). Note that if the treewidth of G is small, this algorithm has a much better running time than the $O(1.2^n)$ algorithm for general graphs.

Computing treewidth (and the corresponding tree decompositions) is a large area of research, and many exact and approximation algorithms have been developed. Bodlaender [16] gives an exact algorithm for computing tree decompositions of width k in time $2^{O(k^3)}n$. Recently, Korhonen and Lokshtanov [84] gave an algorithm with running time $2^{O(k^2)}n^{O(1)}$. Often it is sufficient to obtain a constant factor approximation to treewidth: Korhonen [83] gives a $2^{O(k)}n$ time algorithm for computing a tree decomposition of width $2k+1$ or returns that the treewidth is larger than k.

Planar Graphs. Let us now focus on planar graphs. One of the key results concerns the relationship between grid minors and treewidth of planar graphs. Informally, a planar graph either has small treewidth or contains a large grid as a minor. Formally, we have the following result.

Theorem 2.5 ([110], [69])**.** *Let* $t \in \mathbb{Z}_{\geqslant 0}$*. Every planar graph* G *either has treewidth at most* $O(t)$*, or contains a* $t \times t$*-grid as a minor.*

Furthermore, the proof of the above theorem is constructive and yields a $O(n^2)$ algorithm that returns a $t \times t$-grid minor or a tree decomposition of width $O(t)$. This gives rise to *bidimensionality theory*. Roughly speaking, for many problems (so-called bidimensional problems), the following holds: if there is a solution of size k, the treewidth of the input graph is $O(\sqrt{k})$. This allows us to solve many problems on planar graphs (e.g. VERTEX COVER, INDEPENDENT SET, DOMINATING SET) in time $2^{O(\sqrt{k})}n^{O(1)}$, parameterized by solution size.

Pathwidth. An important special case of tree decompositions are *path decompositions*, defined as follows.

Definition 2.6. *A tree decomposition* $(\{X_i \mid i \in I\}, T = (I, F))$ *is a* path decomposition, *if* T *is a path, and the* pathwidth *of a graph* G *is the minimum width of a path decomposition of* G.

Pathwidth can also be related to the *interval thickness* of a graph:

Lemma 2.7 ([17]). *A graph* G *has pathwidth* k *if and only if it is a subgraph of an interval graph* H *such that* $\omega(H) = k + 1$.

Note that the treewidth of a graph is at most its pathwidth, but even graphs of treewidth one (i.e. forests) can have arbitrarily large pathwidth. Computing pathwidth is NP-complete, and the best known algorithm computes pathwidth in time $2^n n^{O(1)}$ on n-vertex graphs.

2.1.2 Lower bounds

In order to prove that a problem can be solved in certain running time, it suffices to provide an algorithm that solves it in the required time. However, proving that a problem *cannot* be solved within certain time is slightly more challenging. Working under the assumption that $P \neq NP$, we can show that a problem cannot be solved in polynomial time by showing that it is NP-hard. This is usually done by showing that there is a polynomial-time reduction from a problem that is known to be NP-hard.

In classical complexity theory, we consider two problems to be "equally hard" if there is a polynomial-time reduction from one to the other and vice versa. Assuming $P \neq NP$, Ladner [86] constructed an *NP-intermediate* problem: a problem that is in between P and NP-complete (i.e. it is neither in P nor NP-complete). However, this problem is somewhat artificial, and has no connection to other natural problems. We remark here that there are several natural problems which are believed to be NP-intermediate, e.g. the decision version of integer factorization. Until today, most natural problems can be considered equally hard with respect to polynomial-time reductions.

Parameterized Reductions. The parameterized complexity landscape however looks a bit different: there are some natural problems that are believed to be strictly harder than others, i.e. we know that one problem can be reduced to the other, but we do not know if the converse is true. In order to formalize this notion, we first

need to introduce the notion of *parameterized reduction*, which is the parameterized analogue of polynomial-time reduction. Given an instance (x, k) of a parameterized problem, we will use $|x|$ to denote the size of x.

Definition 2.8. *Given two parameterized problems $A, B \subseteq \Sigma^* \times \mathbb{N}$, a* parameterized reduction *from A to B is an algorithm that takes as input an instance (x, k) of A and outputs an instance (x', k') of B such that the following conditions are satisfied:*

- *(x, k) is a* YES*-instance of A if and only if (x', k') is a* YES*-instance of B;*
- *$k' \leqslant g(k)$ for some computable function g;*
- *the algorithm has running time $f(k) \cdot |x|^{O(1)}$ for some computable function f.*

It can be shown that if there is a parameterized reduction from A to B and B can be solved in FPT time, then A can also be solved in FPT time (see Theorem 13.2 [45]). We remark that the second condition in the above definition ensures that the new parameter is bounded by a function of only k: for example, reductions where $k' = n - k$ do not satisfy this condition.

W-hierarchy. The *W-hierarchy* captures the different levels of hardness of parameterized problems. The formal definition of classes $W[t]$ is somewhat involved: informally, for each nonnegative integer t, the class $W[t]$ contains problems which have a parameterized reduction to a version of SAT. It is known that $FPT = W[0]$ and that $W[i] \subseteq W[j]$ for all $i < j$. The notions of hardness and completeness are analogous to those in classical complexity theory.

An example of a $W[1]$-complete problem is INDEPENDENT SET. Problems that are $W[2]$-complete include DOMINATING SET and SET COVER. It is believed that $W[t] \subsetneq W[t+1]$ for all t. The class XP contains problems that can be solved in slice-wise polynomial time, i.e. in time $n^{f(k)}$. It is known that

$$FPT = W[0] \subseteq W[1] \subseteq W[2] \subseteq \cdots \subseteq XP,$$

and it is known that there are problems in XP which are not in FPT.

ETH lower bounds. Let us now focus on lower bounds for FPT problems. Recall that for FPT algorithms, the running time can have a large (e.g. double exponential) running time in k. Therefore, it would be useful to have a tool that allows us to distinguish better between problems in FPT. The EXPONENTIAL TIME HYPOTHESIS (ETH) states, roughly speaking, that 3-SAT cannot be solved in subexponential time in terms of number of variables.

Definition 2.9. *Let δ be the infimum of the set of constants c for which there exists an algorithm that solves 3-*SAT *in time $O^*(2^{cn})$,*[1] *where n is the number of variables. The* Exponential Time Hypothesis *states that $\delta > 0$.*

[1] We use O^* to suppress polynomial factors in input size.

Under the assumption of ETH, we can obtain lower bounds for other problems (i.e. show that there is no algorithm that solves our problem in certain running time): in order to exclude an algorithm with running time $O^*(2^{o(f(|x|))})$, we construct a reduction from 3-SAT to our problem that outputs instances of size $O(f^{-1}(n+m))$, where n and m are the number of variables and clauses respectively, and f^{-1} is the inverse function of f.

2.2 Notation

We let $O^*(\cdot)$ omit factors polynomial in the input size and $\tilde{O}(\cdot)$ omit polylogarithmic factors. Given a graph G and a subset A of its vertices, we define $G[A]$ as the subgraph of G induced by A. Given a graph $G = (V, E)$, we denote by $N(v)$ the open neighbourhood of a vertex v, i.e. $N(v) = \{u \in V : \{u, v\} \in E\}$. We denote by $\omega(G)$ the clique number of G, i.e. the size of the largest clique in G.

We denote all vectors by bold letters, the all ones vector by $\mathbf{1}$ and the all zeros vector by $\mathbf{0}$. We use the Iverson bracket notation: for a Boolean statement b, we define $[b] = 1$ if b is true and $[b] = 0$ if b is false. For integers $a \leqslant b$, we define $[a, b] = \{a, a+1, \ldots, b-1, b\}$ and $[a] = \{1, \ldots, a\}$.

Given a function $f : A \to B$, we define the image of f as $\mathbf{Im}(f) = \{b \in B : (\exists a \in A) f(a) = b\}$, and for $b \in B$ we define the preimage as $f^{-1}(b) = \{a \in A : f(a) = b\}$. Given $C \subseteq A$, we define the restriction of f to C as $f|_C : C \to B$, such that $f|_C(c) = f(c)$ for all $c \in C$. If $g = f|_C$ for some C, we say that f extends g.

We use the notion of multisets, which are sets in which the same element may appear multiple times. Formally, a multiset is an ordered pair (A, m_A) consisting of a set A and a multiplicity function $m_A : A \to \mathbb{Z}^+$.

Flow

Since we discuss several versions of the flow problem in this thesis, we give an overview of their definitions and relations between them.

Definition 2.10. *A* flow network *is a tuple (G, s, t, cap), with $G = (V, E)$ a directed graph, $s, t \in V$ two vertices, and $\mathsf{cap} : E \to \mathbb{N} \cup \{\infty\}$ a capacity function, assigning to each arc a positive integer capacity. A* flow *is a function $f : E \to \mathbb{N}$ that assigns to each arc a non-negative integer such that*

1. *for each arc $e \in E$, $0 \leqslant f(e) \leqslant \mathsf{cap}(e)$ (capacity constraint), and*
2. *for each vertex $v \in V \backslash \{s, t\}$, $\sum_{wv \in E} f(wv) = \sum_{vx \in E} f(vx)$ (flow conservation).*

The value *of a flow f is $\sum_{sx \in E} f(sx) - \sum_{ws \in E} f(ws)$.*

The Flow problem is defined as follows.

Flow
Input: A flow network (G, s, t, cap), and an integer r
Task: Is there a flow of value exactly r?

In Chapter 5, we discuss the All-or-Nothing Flow problem. We say that a flow f is an *all-or-nothing flow* if for each edge $e \in E$, we have $f(e) \in \{0, \mathsf{cap}(e)\}$.

All-or-Nothing Flow
Input: A flow network (G, s, t, cap), and an integer r
Task: Is there an all-or-nothing flow of value exactly r?

One standard version of Flow is the Min Cost Flow problem, where we are additionally asked to minimize the cost of the flow.

Min Cost Flow
Input: Directed graph $G = (V, E)$ with source node set $S \subseteq V$ and sink nodes $T \subseteq V$, $\mathsf{cost} : E \to \mathbb{N}$, $\mathsf{cap} : E \to \mathbb{N} \cup \{\infty\}$
Task: Find a function $f : E \to \mathbb{N}$ such that

- for every $v \in V \setminus (T \cup S)$ we have $\sum_{u \in V} f(u, v) = \sum_{u \in V} f(v, u)$,
- for every $e \in E : f(e) \leqslant \mathsf{cap}(e)$,
- the value of $\sum_{v \in S} \sum_{u \in V} f(v, u)$ is maximal,

and the value $\mathsf{cost}(f) = \sum_{e \in E} \mathsf{cost}(e) f(e)$ is minimized.

In Chapter 6, we will use a problem equivalent to Min Cost Flow, which we will call Demand Flow.

Demand Flow
Input: Directed graph $G = (V, E)$, $D \subseteq V$, $\mathsf{dem} : D \to \mathbb{N}$, $\mathsf{cost} : E \to \mathbb{N}$, $\mathsf{cap} : E \to \mathbb{N} \cup \{\infty\}$
Task: Find a function $f : E \to \mathbb{N}$ such that

- for every $v \in V$ we have $\sum_{u \in V} f(u, v) = \sum_{u \in V} f(v, u)$,
- for every $v \in D$ we have $\sum_{u \in V} f(u, v) = \mathsf{dem}(v)$,
- for every $e \in E : f(e) \leqslant \mathsf{cap}(e)$,

and the value $\mathsf{cost}(f) = \sum_{e \in E} \mathsf{cost}(e) f(e)$ is minimized.

Equivalence of Demand Flow and Min Cost Flow. We argue that Demand Flow is equivalent to Min Cost Flow by simple reductions. First we reduce in the forward way. For each $d \in D$, we create vertices $d_{\mathsf{out}}, d_{\mathsf{in}}$ where d_{out} is a source node with outgoing flow $\mathsf{dem}(d)$ and d_{in} is a sink node with ingoing flow $\mathsf{dem}(d)$. For all other vertices in $V \setminus D$, we create a node and connect to all its neighbors, where all outgoing edges to a vertex in D go to d_{in} and all ingoing edges from a vertex in D connect to d_{out}.

For the other way, let S be the set of source nodes and T be the set of sink nodes of the Min Cost Max Flow problem. We add one "big" node x to the graph, with demand equal to the total outgoing flow from all the source nodes. We add edges (t, x) for all $t \in T$ with $\mathsf{cost}(t, x) = 0$. Furthermore, we add edges (x, s) for all $s \in S$

with $\mathsf{cost}(x, s) = 0$. We set $\mathsf{cap}(x, s)$ and $\mathsf{cap}(t, x)$ to be equal to the corresponding outgoing and incoming flows.

Chapter 3

Detecting and Counting Small Patterns in Unit Disk Graphs

3.1 Introduction

A well-studied theme within the complexity of computational problems on graphs is how much structure within inputs allows faster algorithms. One of the most active research directions herein is to assume that input graphs are *geometrically* structured. The (arguably) two most natural and commonly studied variants of this are to assume that the graph can be drawn on $\mathbb{R}^2$ without crossings (i.e., it is planar) or it is the intersection graph of simple geometric objects. While this last assumption can amount to a variety of different models, a canonical and most simple model is that of *unit disk graphs*: Each vertex of the graph is represented by a disk with unit diameter and two vertices are adjacent if and only if the two associated disks intersect.

The computational complexity of problems on planar graphs has been a very fruitful subject of study: It led to the development of powerful tools such as *Bakers layering technique* and *bidimensionality* that gave rise to efficient approximation schemes and fast (parameterized) sub-exponential time algorithms for many NP-complete problems. One interesting example of such an NP-complete problem is *(induced) subgraph isomorphism*: Given a k-vertex pattern P and an n-vertex host graph G, detect or count the number of (induced) copies of P inside G, denoted with $\mathrm{sub}(P, G)$ (respectively, $\mathrm{ind}(P, G)$). Here we think of k as being much smaller than n, and therefore it is very interesting to obtain running times that are only exponential in k (i.e. fixed-parameter tractable time). This problem is especially appealing since it generalizes many natural NP-complete problems (such as Independent Set, Longest Path and Hamiltonian Cycle) in a natural way, but its generality poses significant challenges for the bidimensionality theory: It does not give sub-exponential time algorithms for this problem.

Only recently, it was shown in a combination of papers [26,62,100] that, on planar graphs, subgraph isomorphism can be solved in $2^{\tilde{O}(\sqrt{k})}n^{O(1)}$ time for many natural pattern classes, and in $2^{O(k/\log k)}$ time for general patterns, complementing the lower

bound of $2^{o(n/\log n)}$ time from [26] based on the Exponential Time Hypothesis (ETH). It was shown in [100] that (induced) pattern occurrences can even be *counted* in sub-exponential parameterized (i.e. $2^{o(k)}n^{O(1)}$) time.

Unfortunately, most of these methods do not immediately work for (induced) subgraph isomorphism on geometric intersection graphs: Even unit disk graphs of bounded ply[1] are not H-minor free for any graph H (which significantly undermines the bidimensionality theory approach), and unit disk graphs of unbounded ply can even have arbitrary large cliques. This hardness is inherent to the graph class: Independent Set is $W[1]$-hard on unit disk graphs and, unless ETH fails, it has no $f(k)n^{o(\sqrt{k})}$-time algorithm for any computable function f [45, Theorem 14.34]. On the positive side, it was shown in [101] that for bounded expansion graphs and fixed patterns, the subgraph isomorphism problem can be solved in linear time, which implies that subgraph isomorphism is fixed-parameter tractable on unit disk graphs; however, their method relies on Courcelle's theorem and hence the dependence of k in their running time is very large and far from optimal.

Therefore, a popular research topic has been to design such fast (parameterized) sub-exponential time algorithms for specific problems such as Independent Set, Hamiltonian Cycle and Steiner Tree [14, 50, 90, 121].

In this chapter we continue this research line by studying the complexity of the decision and counting version of (induced) subgraph isomorphism. While some general methods such as contraction decompositions [102] and pattern covering [92] were already designed for graph classes that include (bounded ply) unit disk graphs, the fine-grained complexity of the subgraph problem itself restricted to unit disk graphs has not been studied and is still far from being understood.

Our Results. To facilitate the formal statements of our results, we need the following definitions: Given two graphs P and G, we define

$$\text{ind}(P,G) = \{f : V(P) \to V(G) : f \text{ is injective}, uv \in E(P) \Leftrightarrow f(u)f(v) \in E(G)\},$$
$$\text{sub}(P,G) = \{f : V(P) \to V(G) : f \text{ is injective}, uv \in E(P) \Rightarrow f(u)f(v) \in E(G)\}.$$

We call elements of $\text{ind}(P,G)$ and $\text{sub}(P,G)$ *pattern occurrences*. Our main theorem reads as follows:

Theorem 3.1. *There is an algorithm that takes as input unit disk graphs P and G on k vertices and n vertices respectively, together with disk embeddings of ply p. It outputs $|\text{sub}(P,G)|$ and $|\text{ind}(P,G)|$ in $(pk)^{O(\sqrt{pk})}\sigma_{O(\sqrt{pk})}(P)^2 n^{O(1)}$ time.*

Note that this theorem can also be used to compute the number of (induced) subgraphs of G isomorphic to P by dividing $|\text{sub}(P,G)|$ with $|\text{sub}(P,P)|$ (and similarly dividing $|\text{ind}(P,G)|$ with $|\text{ind}(P,P)|$). In this theorem, the parameter σ_s is a somewhat technical parameter of the pattern graph P that is defined as follows:

Definition 3.2 ([100])**.** *Given a graph P, we say that (A,B) is a* separation *of P of order s if $A \cup B = V(P)$, $|A \cap B| = s$ and there are no edges between $A \setminus B$ and*

[1] The ply of an embedded unit disk graph is defined as the the maximum number of times any point of the plane is contained in a disk.

$B \setminus A$. *We say that* (A, B) *and* (C, D) *are* isomorphic separations *of* P *if there is a bijection* $f : V(P) \leftrightarrow V(P)$ *such that*

- *For any* $u, v \in V(P)$, $uv \in E(P)$ *if and only if* $f(u)f(v) \in E(P)$,
- $f(A) = C$, $f(B) = D$,
- *For any* $u \in A \cap B$, $f(u) = u$.

We denote by $\mathcal{S}_s(P)$ *a maximal set of pairwise non-isomorphic separations of* P *of order at most* s. *We define* $\sigma_s(P) = |\mathcal{S}_s(P)|$ *as the number of non-isomorphic separations of* P *of order at most* s.

Note that $\sigma_s(P) \geqslant \binom{k}{s} s^s$. For ply $p = O(1)$ and many natural classes of patterns such as independent sets, cycles or grids, it is easy to see that $\sigma_{O(\sqrt{k})}(P)$ is at most $2^{\tilde{O}(\sqrt{k})}$ and therefore Theorem 3.1 gives a $2^{\tilde{O}(\sqrt{k})} n^{O(1)}$ time algorithm for computing $|\mathrm{sub}(P, G)|$ and $|\mathrm{ind}(P, G)|$. We also give the following new non-trivial bounds on $\sigma_s(P)$ whenever P is a general (connected) unit disk graph:

Theorem 3.3. *Let* P *be a* k*-vertex unit disk graph with an embedding of ply* p, *and* s *be an integer. Then: (a)* $\sigma_s(P)$ *is at most* $2^{O(s \log k + pk/\log k)}$, *and (b) If* P *is connected, then* $\sigma_s(P) \leqslant 2^{O(s \log k)}$.

Using Theorem 3.1, this allows us to conclude the following result.

Corollary 3.1. *There is an algorithm that takes as input a* k*-vertex unit disk graph* P *and an* n*-vertex unit disk graph* G, *together with their unit disk embeddings of ply* p, *and outputs* $|\mathrm{sub}(P, G)|$ *and* $|\mathrm{ind}(P, G)|$ *in* $2^{O(pk/\log k)} n^{O(1)}$ *time.*

We show that this cannot be significantly improved even if P and G have ply two:

Theorem 3.4. *Assuming ETH, there is no algorithm to determine if* $\mathrm{sub}(P, G)$ *(*$\mathrm{ind}(P, G)$ *respectively) is nonempty for given* n*-vertex unit disk graph* G *and a unit disk graph* P *in* $2^{o(n/\log n)}$ *time, even when* P *and* G *have a given embedding of ply* 2.

Note that the ply of G is 1 if and only if G is an independent set so the assumption on the ply in the above statement is necessary. To our knowledge, this is the first lower bound based on the ETH excluding $2^{O(\sqrt{n})}$ time algorithms for problems on unit disks graphs (of bounded ply), in contrast to previous bounds that only exclude $2^{o(\sqrt{n})}$ time algorithms.

Clearly, a unit disk graph G of ply p has a clique of size p, i.e. its clique number $\omega(G)$ is at least p. On the other hand, it was shown in [70] that $p \geqslant \omega(G)/5$. In other words, parameterizations by ply and clique number are equivalent up to a constant factor.

3.1.1 Our techniques

Our approach heavily builds on the previous works [26,62,100]: Theorem 3.1 is proved via a combination of the dynamic programming technique from [26] that stores representatives of non-isomorphic separations to get the runtime dependence down from

$2^{O(k)}$ to $\sigma_{O(\sqrt{pk})}(P)$, and the *efficient inclusion-exclusion* technique from [100] to solve counting problems (on top of decision problems).[2] We combine these techniques with a divide and conquer strategy that divides the unit disk graph in smaller graphs using horizontal and vertical lines as separators. As a first step in our proof, we give a *Turing kernelization* for the counting versions of (induced) subgraph isomorphism that uses efficient inclusion-exclusion. Theorem 3.3 uses a proof strategy from [26] combined with a bound from [27] on the number of non-isomorphic unit disk graphs. Theorem 3.4 builds on a reduction from [26], although several alterations are needed to ensure the graph is a unit disk graph of bounded ply.

Organization. In Section 3.2 we provide additional notation and some preliminary lemmas. In Section 3.3 we provide a Turing kernel. In Section 3.4 we build on Section 3.3 to provide the proof of Theorem 3.1. In Section 3.5 we give a bound on the parameter σ_s. In Section 3.6 we give a lower bound, i.e. prove Theorem 3.4. We finish with some concluding remarks in Section 3.7.

3.2 Preliminaries

Given a unit disk graph G, we say that G has *ply* p if there is an embedding of G such that every point in the plane is contained in at most p disks of G. Given an embedding of a unit disk graph G and a vertex $v \in V(G)$, we define $D(v)$ as the unit disk corresponding to v. Let P and G be unit disk graphs and let $|V(G)| = n$, $|V(P)| = k$.

Definition 3.5. *For integers b, h, we say a unit disk graph G of ply p can be* drawn in a $(b \times h)$-box *with ply p if it has an embedding of ply p as unit disk graph in a $b \times h$ rectangle.*

Throughout this chapter, we assume that the sides of the box are axis parallel, and the lower left corner is at $(0, 0)$. We assume that if a graph G can be drawn in a $(b \times h)$-box then we are given such an embedding.

Lemma 3.6. *Given a unit disk graph G with a drawing in a $(b \times h)$-box with ply p, one can construct in polynomial time a path decomposition of G of width $4(\min\{b, h\}+1)p$.*

Proof. Without loss of generality, we may assume $h \leqslant b$. For $i = 1, \ldots b$ define

$$L_i = \{(x, y) \in \mathbb{R}^2 : \ i - 1 \leqslant x \leqslant i\},$$

$$X_i = \{v \in V(G) : \ D(v) \cap L_i \neq \emptyset\}.$$

It is easy to see that $X_1, \ldots, X_b$ is a path decomposition of G. Let us now bound the size of X_i. Let $S_i = \{(x, y) \in \mathbb{Z}^2 : \ x \in \{i - 2, i - 1, i, i + 1\}$ and $0 \leqslant y \leqslant h\}$. Each disk in X_i contains a point from S_i, so $|X_i| \leqslant p \cdot |S_i| = 4p(h + 1)$. ■

[2] Similar to what was discussed in [100], this technique seems to be needed even for simple special cases of Theorem 3.1 such as counting independent sets on subgraphs of (subdivided) grids.

Lemma 3.7 (Theorem 6.1 from [27]). *Let a non-decreasing bound $b = b(n)$ be given, and let $\mathcal{U}_n$ denote the set of unlabeled unit disk graphs on n vertices with maximum clique size at most b. Then $|\mathcal{U}_n| \leqslant 2^{12(b+1)n}$.*

Non-isomorphic Separations. In this chapter we will work with non-isomorphic separations of small order, as defined in Definition 3.2.

Observation 3.1. *Given a k-vertex graph P and separations (A, B) and (C, D) of P, one can check in quasi-polynomial time in k if (A, B) and (C, D) are isomorphic separations and, if yes, find the corresponding bijection.*

The above observation can be shown using the quasi-polynomial time algorithm for graph isomorphism from [4], extended to the colored subgraph isomorphism problem by a standard reduction (see [112, Theorem 1]).

For separations $(A, B), (A', B') \in \mathcal{S}_s(P)$, we define

$$\mu((A, B), (A', B')) = |\{(C, D) : (C, D) \text{ is a separation of } P \text{ such that}$$

$$C \subseteq A' \text{ and } (C, D) \text{ isomorphic to } (A, B)\}|.$$

Lemma 3.8. *Given a graph P, one can compute $\mathcal{S}_s(P)$ and for each pair of separations $(A, B), (A', B') \in \mathcal{S}_s(P)$ the multiplicity $\mu((A, B), (A', B'))$ in time $\sigma_s(P)n^{O(1)}$.*

Proof. The proof is identical the proof of [100], but recorded here for completeness. We start enumerating separations (C, D) by iterating over all candidates X for $C \cap D$. There are $\binom{k}{s}$ possibilities since we are interested in separations of order s. Now any separation (C, D) of P with $C \cap D = X$ is formed by selecting a subset of the connected components of $P - X$ and adding it to C (the remaining connected components will be added to D). Given two connected components P_1 and P_2 of $P - X$, we can check in quasi-polynomial time in k whether there is an isomorphism $f : V(P) \to V(P)$ such that $f|_X$ is the identity function and $f(P_1) = f(P_2)$: Indeed, we apply Observation 3.1 to $(P_1 \cup X, (P - P_1) \cup X)$ and $(P_2 \cup X, (P - P_2) \cup X)$. This isomorphism relation is an equivalence relation, and we define for each equivalence class a unique representative denoted with $\mathcal{P}_i$.

Let $\mathcal{P}_1, \ldots, \mathcal{P}_t$ be the computed set of representatives of all isomorphism classes of connected components of $P - X$, and let p_i be the number of connected components of $P - X$ isomorphic to $\mathcal{P}_i$.

Now we can construct $\mathcal{S}_s(P)$ as follows: For each vector t-dimensional vector $(v_1, \ldots, v_t) \in \{0, \ldots, p_1\} \times \cdots \times \{0, \ldots, p_t\}$, we add to $\mathcal{S}_s(P)$ the separation (C, D) where C is obtained by adding for each $i = 1, \ldots, t$ exactly v_i copies of a connected component isomorphic to $\mathcal{P}_i$ to C (and the remaining $p_i - v_i$ components to D). This concludes the construction of $\mathcal{S}_s$.

The multiplicity $\mu((A, B), (A', B'))$ can be computed in a similar way: For each candidate X for $A \cap B \subseteq A'$ we compute for each two connected components of $P - X$ that do not contain elements from B' whether they are isomorphic (analogous to how it was done above). Let $\mathcal{P}_1, \ldots, \mathcal{P}_t$ be the computed set of representatives of all isomorphism classes of connected components of $P - X$ that do not intersect B', and let p_i be the number of connected components of $P - X$ not intersecting B' that

are isomorphic to $\mathcal{P}_i$. If A contains v_i components that are isomorphic to $\mathcal{P}_i$, then we can compute

$$\mu((A,B),(A',B')) = \prod_{i=1}^{t} \binom{p_i}{v_i}.$$

It is easy to see that all the above can be done in the claimed time bound (since $\sigma_s(P) \geqslant \binom{k}{s} s^s$). ■

Solving instances where G has low pathwidth. The following lemma can be shown with standard dynamic programming over tree decompositions:

Lemma 3.9. *Given P, G, and $I \subseteq V(G)$, a path decomposition of G of width t, we can compute $|\{g \in \mathrm{sub}(P,G) : g \text{ extends } f\}|$ and $|\{g \in \mathrm{ind}(P,G) : g \text{ extends } f\}|$ for every $P' \subseteq V(P)$ of size at most $|I|$ and injective function $f : P' \to I$ in time $\sigma_t(P)(t+1)^t(|V(P)|+1)^{|I|}n^{O(1)}$.*

Proof. We present only the proof for the subgraph case, as the proof for induced case is completely analogous. It is well known (see for example [45, Lemma 7.2]) that each path decomposition can be modified to a path decomposition of the same width such that for each bag i either we have $X_i = X_{i-1} \cup \{v\}$ or $X_i = X_{i-1} \setminus \{v\}$. We say X_i is an *introduce* bag in the first case, and a *forget bag* in the second case.

For a bag i, we define $G_i = G[\cup_{j=1}^{i} X_j]$. For each bag i, separation (A,B) of P of order t, and functions $f : P' \to I$ (where $P' \subseteq V(P)$) and $h : A \cap B \to X_i$ we define

$$T_i[(A,B),f,h] = |\{g \in \mathrm{sub}(P[A],G_i) : g|_{A\cap P'} = f|_{A\cap P'} \text{ and } g \text{ extends } h\}|.$$

Leaf. This corresponds to $i = 1$ and G_1 being an empty graph. Then we have $T_1[(A,B),f,h] = 1$ for $A = \emptyset$ and f, h being the empty function, and it is equal to 0 otherwise.

Introduce Vertex. This corresponds to $G_i = G_{i-1} \cup v$ for some vertex v. If $h^{-1}(v) = \emptyset$ we have

$$T_i[(A,B),f,h] = T_{i-1}[(A,B),f,h].$$

Otherwise, if h is not an injective function, or there is a vertex $u \in A \cap N(h^{-1}(v))$ such that $\{h(u), v\} \notin E(G)$, or $f^{-1}(v)$ is non-empty but not equal to $h^{-1}(v)$, then $T_i[(A,B),f,h] = 0$ since h cannot be extended to any function counted in $T_i[(A,B),f,h]$.

Otherwise, let $h^{-1}(v) = \{a\}$. We have that

$$T_i[(A,B),f,h] = T_{i-1}[(A \setminus \{a\}, B \cup \{a\}), f, h|_{(A\cap B)\setminus\{a\}}]$$

since any function g counted in $T_i[(A,B),f,h]$ needs to map a vertex in A to v and the injectivity and adjacencies and extension properties involving v are satisfied by assumption.

Forget Vertex. This corresponds to $X_i = X_{i-1} \setminus \{v\}$ for some vertex v. In this case we need to decide the preimage of v (if any):

$$T_i[(A,B),f,h] = T_{i-1}[(A,B),f,h] + \sum_{u \in A \setminus B} T_{i-1}[(A, B \cup \{u\}), f, h'],$$

where h' is obtained from h by extending its domain with u and defining $h'(u) = v$. If u is the preimage of v is a function g contributing to $T_i[(A,B),f,h]$, then $u \in A$ since it is mapped to a vertex of G_i, and it is not in $A \cap B$ because h then already maps u to a vertex in X_i (and since X_i does not contain v, it is therefore it is not well-defined).

Note that currently the first index of the table entries T_i goes over all separations of P of order t, which could be too large for our purposes. We overcome this issue by showing that it suffices to compute the table entries indexed by separations in $\mathcal{S}_t(P)$. Formally, given a separation (A,B) of P of order at most t and functions $f : P' \to I$, $h : A \cap B \to X_i$, we claim that we can find in quasi-polynomial time a separation $(C,D) \in \mathcal{S}_t(P)$ and $f' : P'' \to I$ such that $T_i[(A,B),f,h] = T_i[(C,D),f',h]$. Indeed, using Observation 3.1, we can check for each separation in $\mathcal{S}_t(P)$ whether it is isomorphic to (A,B). Once we found a separation $(C,D) \in \mathcal{S}_t(P)$ that is isomorphic to (A,B) and the corresponding isomorphism $\phi : V(P) \to V(P)$ that maps (A,B) to (C,D), we define f' as the composition of ϕ and f and $P'' = \phi(P')$. Note that by construction we have $C \cap D = A \cap B$ and $\phi|_{A\cap B}$ is the identity function.

Now the desired value $|\{g \in \mathrm{sub}(P,G) : g \text{ extends } f\}|$ can be found as the table entry $T_i[(V(P),\emptyset),f,h]$, where i is the last bag of the path decomposition and h the empty function. The number of table entries is at most $\sigma_t(P)(t+1)^t(|V(P)|+1)^{|I|}n$, and therefore the running time of the algorithm is at most $\sigma_t(P)(t+1)^t(|V(P)|+1)^{|I|}n^{O(1)}$. ■

The following lemma simply states that a long product of matrices can be evaluated quickly, but is nevertheless useful as subroutine in the "efficient inclusion-exclusion" technique.

Lemma 3.10 ([100]). *Given a set A, an integer h and a value $T[x,x'] \in \mathbb{Z}$ for every $x,x' \in A$, the value*

$$\sum_{x_1,\ldots,x_h \in A} \prod_{i=1}^{h-1} T[x_i, x_{i+1}] \tag{3.1}$$

can be computed in $O(h|A|^2)$ time.

3.3 Turing Kernel

We will now present a preprocessing algorithm for computing $|\mathrm{sub}(P,G)|$ (the algorithm for $|\mathrm{ind}(P,G)|$ is analogous) that allows us to assume that G can be drawn in a $(O(k) \times O(k))$-box with ply p, i.e. that $|V(G)| = O(k^2 p)$. This can be seen as a polynomial Turing kernel in case p and $\sigma_0(P)$ are polynomial in k (note that $\sigma_0(P)$ could be exponentially large in k). A *Turing kernel* of size $f(k)$ is an algorithm that solves the given problem in polynomial time, when given access to an oracle that solves instances of size at most $f(k)$ in a single step.

Lemma 3.11 describes how to reduce the width of G (and analogously the height of G). To prove it, we use the shifting technique. This general technique was first used by Baker [5] for covering and packing problems on planar graphs and by Hochbaum and Maass [73] for geometric problems stemming from VLSI design and image processing.

Intuitively, we draw the graph on a grid, and delete all the disks that intersect certain columns of the grid. After doing that, the remaining graph will consist of several small disconnected "building blocks". Each connected component of the pattern will be fully contained in one of the blocks, and since the blocks are small we can use the oracle to count the number of these occurrences. We take advantage of the fact that we can group together connected components that are isomorphic. We use Lemma 3.11 twice, to reduce the width and height of G to $O(k)$.

Lemma 3.11. *Suppose we have access to an oracle that computes $|\mathrm{sub}(P, G')|$ in constant time, where the host graph G' can be drawn in a $(h \times O(k))$-box for some h with ply p. Then we can compute $|\mathrm{sub}(P, G)|$ for host graphs G of ply p in time $n \cdot k^{O(1)} \cdot \sigma_0(P)^2$.*

Proof. For $i \in \{0, \ldots, 2k\}$, let $C_i = \{(x, y) \in \mathbb{R}^2 : x \equiv i \pmod{2k+1}\}$. Informally, we draw a grid and select every $(2k+1)$th vertical gridline. Let P_i be the set of all pattern occurrences whose image is disjoint from C_i:

$$P_i = \{f : V(P) \to V(G) : f \in \mathrm{sub}(P, G) \text{ and } \forall v \in \mathbf{Im}(f) : D(v) \cap C_i = \emptyset\}.$$

Note that every disk intersects at most two grid lines, so by the pigeonhole principle we have that $\mathrm{sub}(P, G) = \cup_{i=0}^{2k} P_i$. By the inclusion-exclusion principle, $\left|\bigcup_{i=0}^{2k} P_i\right|$ equals

$$\sum_{\emptyset \subset C \subseteq \{0,\ldots,2k\}} (-1)^{|C|} \left|\bigcap_{i \in C} P_i\right| = \sum_{\ell=1}^{2k+1} (-1)^{\ell} \sum_{0 \leqslant c_1 < \cdots < c_\ell \leqslant 2k} \left|\bigcap_{j \in \{c_1,\ldots,c_\ell\}} P_j\right|. \tag{3.2}$$

Let us show how we can compute $|\cap_{j \in \{c_1,\ldots,c_\ell\}} P_j|$ quickly. For $a, b \in \{0, ..., 2k\}$, we define $B[a, b]$ to be

$$\begin{cases} \{(x, y) \in \mathbb{R}^2 : (\exists t \in \mathbb{N}_0)\, a + (2k+1)t < x < b + (2k+1)t\}, & \text{if } a \leqslant b, \\ \{(x, y) \in \mathbb{R}^2 : (\exists t \in \mathbb{N}_0)\, a + (2k+1)(t-1) < x < b + (2k+1)t\}, & \text{if } a > b. \end{cases}$$

We define $\mathcal{B}[a, b]$ as the induced subgraph of G such that all its disks are fully contained in $B[a, b]$, i.e. the subgraph induced by the vertex set $\{v \in V(G) : D(v) \subseteq B[a, b]\}$. These sets are our "building blocks": after deleting $C_{c_1}, \ldots C_{c_\ell}$, the remaining graph is $\cup_{\alpha=1}^{\ell} \mathcal{B}[c_\alpha, c_{\alpha+1}]$, where we define $c_{\ell+1} = c_1$.

Let t be the number of non-isomorphic connected components of P and let $\mathcal{C}_0(P) = \{\mathcal{P}_1, \ldots, \mathcal{P}_t\}$ be the set of representatives of all isomorphism classes of connected components of P. We can encode P as vector $\mathbf{p} = (p_1, \ldots, p_t)$, where p_i is the size of the isomorphism class of $\mathcal{P}_i$.

Let $U = \{0, \ldots, p_1\} \times \cdots \times \{0, \ldots, p_t\}$. For a t-dimensional vector $(v_1, \ldots, v_t) \in U$ we define $P[(v_1, \ldots, v_t)]$ as the subgraph of P that contains v_i copies of $\mathcal{P}_i$.

We would like to count in how many ways can we distribute the connected components of P to the building blocks. Equivalently, we can count the number of ways to assign a vector $\mathbf{v}^\alpha \in U$ to each block $\mathcal{B}[c_\alpha, c_{\alpha+1}]$ such that $\sum \mathbf{v}^\alpha = \mathbf{p}$.

Thus we have

$$\left|\bigcap_{j\in\{c_1,\dots,c_\ell\}} P_j\right| = \sum_{\mathbf{v}^1+\cdots+\mathbf{v}^\ell=\mathbf{p}} \prod_{\alpha=1}^{\ell} |\mathrm{sub}(P[\mathbf{v}^\alpha], \mathcal{B}[c_\alpha, c_{\alpha+1}])|. \tag{3.3}$$

Note that $|U| = (p_1+1)\cdots(p_t+1) = \sigma_0(P)$: indeed, every vector $\mathbf{u} \in U$ corresponds to a unique separation $(V(P'), V(P-P'))$ of order 0, where P' consists of u_i copies of $\mathcal{P}_i$. Combining (3.2) and (3.3), we get that

$$\left|\bigcup_{i=0}^{2k} P_i\right| = \sum_{\ell=1}^{2k} (-1)^\ell T_\ell,$$

where

$$T_\ell = \sum_{\substack{0\leqslant c_1<\cdots<c_\ell\leqslant 2k \\ \mathbf{v}^1+\cdots+\mathbf{v}^\ell=\mathbf{p}}} \prod_{\alpha=1}^{\ell} |\mathrm{sub}(P[\mathbf{v}^\alpha], \mathcal{B}[c_\alpha, c_{\alpha+1}])|. \tag{3.4}$$

Suppose for now that we have computed $|\mathrm{sub}(P[\mathbf{v}^\alpha], \mathcal{B}[a,b])|$ for all $a, b \in \{0,\dots,2k\}$, $\mathbf{v}^\alpha \in U$, and that we want to compute T_ℓ quickly.

To apply Lemma 3.10, we have to rewrite the sum (3.4) in such a way that the variables are pairwise independent. We replace the condition $c_i < c_{i+1}$ by multiplying with $[c_i < c_{i+1}]$. To replace the condition on the variables $\mathbf{v}^i$, we will re-index these variables by $\mathbf{u}^1,\dots,\mathbf{u}^\ell$, where $\mathbf{u}^i = \sum_{j=1}^{i} \mathbf{v}^j$ for $i \in [\ell-1]$ and $\mathbf{u}^\ell = \mathbf{p} - \mathbf{u}^{\ell-1}$, $\mathbf{u}^0 = \mathbf{0}$. Therefore, we have

$$T_\ell = \sum_{\substack{c_1,\dots,c_\ell\in\{0,\dots,2k\} \\ \mathbf{u}^1,\dots,\mathbf{u}^{\ell-1}\in U}} |\mathrm{sub}(P[\mathbf{p}-\mathbf{u}^{\ell-1}], \mathcal{B}[c_\ell, c_1])| \prod_{i=1}^{\ell-1} [c_i < c_{i+1}] \cdot [\mathbf{u}^{i-1} \leqslant \mathbf{u}^i]$$

$$\cdot |\mathrm{sub}(P[\mathbf{u}^i - \mathbf{u}^{i-1}], \mathcal{B}[c_i, c_{i+1}])|.$$

By Lemma 3.10, we can compute T_ℓ in time $\ell \cdot ((2k+1)\cdot\sigma_0(P))^2$ if we are given $|\mathrm{sub}(P[\mathbf{u}], \mathcal{B}[a,b])|$ for all $\mathbf{u} \in U$, $a, b \in \{0,\dots,2k\}$.

It remains to show how we can compute $|\mathrm{sub}(P[\mathbf{u}], \mathcal{B}[a,b])|$ for given $\mathbf{u} \in U$, $a, b \in \{0,\dots,2k\}$. Let $C_1,\dots,C_d$ be the connected components of $\mathcal{B}[a,b]$. Note that each C_i can be drawn in a $(h \times O(k))$-box with ply p for some h, so we can use the oracle to compute $|\mathrm{sub}(P[\mathbf{w}], C_i)|$ for all $\mathbf{w} \in U$, $i \in [d]$. We would like to distribute the connected components of $P[\mathbf{u}]$ to $C_1,\dots C_d$. We can do this by dynamic programming. For $i \in [d]$ and $\mathbf{w} \in U$, we define

$$T'[i, \mathbf{w}] = |\mathrm{sub}(P[\mathbf{w}], C_1 \cup \cdots \cup C_i)|$$

The recurrence is as follows:

$$T'[i, \mathbf{w}] = \sum_{\mathbf{w}'\leqslant\mathbf{w}} |\mathrm{sub}(P[\mathbf{w}'], C_i)| \cdot T[i-1, \mathbf{w}-\mathbf{w}'],$$

where $\mathbf{w} \leqslant \mathbf{w'}$ indicates that $\mathbf{w}$ is in each coordinate smaller than $\mathbf{w'}$. Thus we can compute $T'[i, \mathbf{u}]$ in time $d|U|^2 = d\sigma_0(P)^2 \leqslant (n/2k)\sigma_0(P)^2$.

Therefore, we can compute $|\mathrm{sub}(P, G)|$ in time $n \cdot k^{O(1)} \cdot \sigma_0(P)^2$. ■

Theorem 3.12. *For unit disk graphs P and G with given embedding of ply p, $|\mathrm{sub}(P, G)|$ can be computed in time $\sigma_0(P)^2 \cdot n \cdot k^{O(1)}$ when given access to an oracle that computes $|\mathrm{sub}(P, G)|$ where the host graph has size $O(k^2 p)$ in constant time. In particular, there is a Turing kernel for computing $|\mathrm{sub}(P, G)|$ when $\sigma_0(P)$ and p are polynomial in k.*

Proof. Using Lemma 3.11, we reduce the problem to computing $|\mathrm{sub}(P, G)|$ for graphs G that can be drawn in a $(h \times O(k))$-box with ply p. Applying Lemma 3.11 one more time, we reduce the problem to computing $|\mathrm{sub}(P, G)|$ for graphs G that can be drawn in a $(O(k) \times O(k))$-box with ply p, i.e. where $|V(G)| = O(k^2 p)$.

■

3.4 Proof of Theorem 3.1: Algorithm

We present only the proof for $\mathrm{sub}(P, G)$, since the proof for $\mathrm{ind}(P, G)$ is analogous. Before we start with the proof, we need to give a number of definitions: Suppose that a unit disk embedding of G in a $(b \times h)$-box with ply p is given.

For integers $0 \leqslant x \leqslant x' \leqslant b$, we define $V\langle x, x'\rangle \subseteq V(G)$ as the set of vertices $v \in V(G)$ such that $D(v) \cap \{(x_0, y_0) \in \mathbb{R}^2 : x \leqslant x_0 \leqslant x'\} \neq \emptyset$. Informally, $V\langle x, x'\rangle$ consists of vertices whose associated unit disks are (partially) between vertical lines x and x'. We define $G\langle x, x'\rangle = G[V\langle x, x'\rangle]$ and $V\langle x\rangle = V\langle x, x\rangle$, $G\langle x\rangle = G\langle x, x\rangle$.

Given functions $f_1 : D_1 \to R_1$ and $f_2 : D_2 \to R_2$, we say f_1 and f_2 are *compatible* if

- for all $u \in D_1 \cap D_2$, $f_1(u) = f_2(u)$, and
- for all $r \in R_1 \cap R_2$, we have $f_1^{-1}(r) = f_2^{-1}(r)$.

If f_1, f_2 are compatible, we define $f = f_1 \cup f_2$ as the function with domain $D_1 \cup D_2$ satisfying $f|_{D_1} = f_1$ and $f|_{D_2} = f_2$.

Note that in the above definition, the choice of R_1 and R_2 matter. For example, the identity functions $f_1 : \{1\} \to \{1\}$ and $f_2 : \{2\} \to \{2\}$ are compatible, but the identity functions $f_3 : \{1\} \to \{1, 2\}$ and $f_4 : \{2\} \to \{1, 2\}$ are not compatible, since $f_3^{-1}(1) = \{1\}$ but $f_4^{-1}(1) = \emptyset$.

Using Theorem 3.12, we can assume that G can be drawn in a $(O(k) \times O(k))$-box with ply p. We will use dynamic programming. We will first define the sets of partial solutions that are counted in this dynamic programming algorithm. For variables

- integers $0 \leqslant x < x' \leqslant b$,
- separation (A, B) of P of order at most $2\sqrt{pk}$,
- injective $f : A \cap B \to V\langle x\rangle \cup V\langle x'\rangle$ such that $|f^{-1}(V\langle x\rangle)|, |f^{-1}(V\langle x'\rangle)| \leqslant \sqrt{pk}$

we define

$$T[x, x', (A, B), f] = \{g \in \mathrm{sub}(P[A], G\langle x, x'\rangle) : g \text{ extends } f\}.$$

Note that T is indexed by *any* separation of P of order $2\sqrt{pk}$. We will later replace this with a set of non-isomorphic separations to obtain the claimed $\sigma_{2\sqrt{pk}}(P)$ dependence in the running time.

Informally, $T[x, x', (A, B), f]$ is the set of all occurrences of $P[A]$ in $G\langle x, x'\rangle$ such that f describes their behaviour on the "boundary" $V\langle x\rangle \cup V\langle x'\rangle$. We will now show how to compute the table entries. We consider two cases, depending on whether $x' - x$ is less than $\sqrt{k/p}$ or not.

Case 1: $x' - x \leqslant \sqrt{k/p}$
Note that in this case, the pathwidth t of $G\langle x, x'\rangle$ is $O(\sqrt{pk})$ by Lemma 3.6. For a fixed subset $I \subseteq V\langle x\rangle \cup V\langle x'\rangle$ of size $2\sqrt{pk}$, we apply Lemma 3.9 to compute $T[x, x', (A, B), f] = \{g \in \mathrm{sub}(P[A], G\langle x, x'\rangle) : g \text{ extends } f\}$ for every injection $f : A \cap B \to I$ in time $\sigma_t(P[A])(t+1)^t(|A|+1)^{2\sqrt{pk}}|V\langle x, x'\rangle|^{O(1)}$. Applying Lemma 3.9 to every subset $I \subseteq V\langle x\rangle \cup V\langle x'\rangle$ of size $2\sqrt{pk}$, we can compute $T[x, x', (A, B), f]$ for every f in time $\binom{2pk}{2\sqrt{pk}}\sigma_t(P)(\sqrt{pk})^{O(\sqrt{pk})}|V\langle x, x'\rangle|^{O(1)}$. Using $t = O(\sqrt{pk})$, $|V\langle x, x'\rangle| = O(k\sqrt{kp})$ and a well-known bound on binomial coefficients, $\binom{a}{b} \leqslant (\frac{ae}{b})^b$, the above running time is bounded by $(pk)^{O(\sqrt{pk})}\sigma_{O(\sqrt{pk})}(P)$.

Case 2: $x' - x > \sqrt{k/p}$
Let $g \in T[x, x', (A, B), f]$, and let $Q = \mathbf{Im}(g)$. For $m \in \{x+1, \ldots, x'-1\}$, we say that Q is *sparse at* m if $|Q \cap V\langle m\rangle| \leqslant 2\sqrt{pk}$, i.e. the vertical line at m intersects at most $2\sqrt{pk}$ disks in Q. Since $|Q| \leqslant k$, every disk intersects at most two grid lines and $x' - x > \sqrt{k/p}$, there is at least one m such that Q is sparse at m by the averaging principle. Therefore,

$$T[x, x', (A, B), f] = \bigcup_{m=x+1}^{x'-1} \{g \in T[x, x', (A, B), f] : g(A) \text{ is a sparse at } m\}.$$

By the inclusion-exclusion principle, $|T[x, x', (A, B), f]|$ is equal to

$$\sum_{\emptyset \subset X \subseteq \{x+1, \ldots, x'-1\}} (-1)^{|X|} \left|\{g \in T[x, x', (A, B), f] : g(A) \text{ is sparse at all } m \in X\}\right|.$$

Denoting $X = \{x_1, \ldots, x_\ell\}$, where $x_1 < \cdots < x_\ell$, we further rewrite this into

$$\sum_{\ell=1}^{x'-x-2} (-1)^\ell \sum_{x < x_1 < \cdots < x_\ell < x'} |\{g \in T[x, x', (A, B), f] : g(A) \text{ is sparse at } x_1, \ldots, x_\ell\}|.$$

Now we claim that, since $Q \cap V\langle m\rangle$ is a separator of $G[Q]$, $|T[x, x', (A, B), f]|$ can be further rewritten to express it recursively as follows:

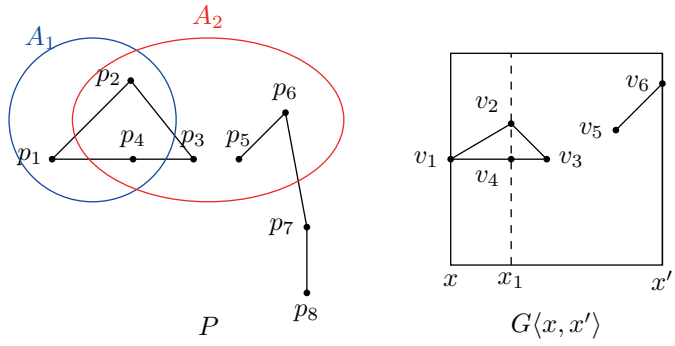


Figure 3.1: The function $g : \{p_1, \ldots, p_6\} \to V\langle x, x'\rangle$ defined by $g(p_i) = v_i$, corresponds to functions $g_1 : A_1 \to V\langle x, x_1\rangle$ and $g_2 : A_2 \to V\langle x_1, x'\rangle$, where $g_1(p_i) = v_i$ and $g_2(p_i) = v_i$.

Claim 3.1.

$$|T[x, x', (A, B), f]| = \sum_{\ell=1}^{x'-x-2} (-1)^\ell \sum_{(*)} \prod_{i=0}^{\ell} |T[x_i, x_{i+1}, (A_i, B_i), f_i]|,$$

where we let $x_0 = x$ and $x_{\ell+1} = x'$ for convenience and the sum $(*)$ goes over

- integers $x < x_1 < \cdots < x_\ell < x'$,
- separations (A_i, B_i) of P of order $4\sqrt{pk}$ for each $i = 0, \ldots, \ell$, such that
  - $\cup_{i=0}^{\ell} A_i = A$, and
  - $A_i \setminus B_i$ and $A_j \setminus B_j$ are disjoint for each $0 \leqslant i < j \leqslant \ell$,
- functions $f_i : A_i \cap B_i \to V\langle x_i\rangle \cup V\langle x_{i+1}\rangle$ for each $i = 0, \ldots, \ell$ such that $f, f_1, \ldots, f_\ell$ are pairwise compatible and $|f_i^{-1}(V\langle x_i\rangle)|, |f_i^{-1}(V\langle x_{i+1}\rangle)| \leqslant 2\sqrt{pk}$.

Proof of Claim: To prove this claim, consider first a function $g \in T[x, x', (A, B), f]$ such that $g(A)$ is sparse at $x_1, \ldots, x_\ell$. We describe how to find the separations (A_i, B_i) and functions f_i that correspond to g. Let $A_i = g^{-1}(V\langle x_i, x_{i+1}\rangle)$, $B_i = (V(P) \setminus A_i) \cup g^{-1}(V\langle x_i\rangle) \cup g^{-1}(V\langle x_{i+1}\rangle)$. Note that, since $g(A)$ is sparse at x_i and x_{i+1}, (A_i, B_i) is a separation of order at most $4\sqrt{pk}$. It is easy to see that $\cup A_i = g^{-1}(V\langle x, x'\rangle) = A$. Also, note that $A_i \setminus B_i = g^{-1}(V\langle x_i, x_{i+1}\rangle) \setminus (g^{-1}(V\langle x_i\rangle) \cup g^{-1}(V\langle x_{i+1}\rangle))$, so for any $i \neq j$, $A_i \setminus B_i$ and $A_j \setminus B_j$ are disjoint. We define $f_i : A_i \cap B_i \to V\langle x_i\rangle \cup V\langle x_{i+1}\rangle$ as $f_i = g|_{A_i \cap B_i}$. By construction, $f, f_1, \ldots, f_\ell$ are pairwise compatible.

Conversely, given pairwise compatible functions $g_0, \ldots, g_\ell$ such that $g_i \in T[x_i, x_{i+1}, (A_i, B_i), f_i]$, we show how to construct a function $g \in T[x, x', (A, B), f]$. Since the g_i's are compatible, we can define $g = g_0 \cup \cdots \cup g_\ell : A \to V\langle x, x'\rangle$. Since f, g_1, g_ℓ are pairwise compatible, g extends f. It is easy to see that this correspondence is one to one, which proves the claim. □

The next step is to rewrite the sum $(*)$ to match the form of Lemma 3.10. The only difference is that in (3.1) the summation is over variables that are pairwise independent.

Formally, let us define a square matrix M whose indices M_{ind} are of the form $(x_i, (A_i, B_i), f_i)$, where $x_i \in \{x, \ldots, x'\}$, $(A_i, B_i) \in \mathcal{S}_{2\sqrt{pk}}(P)$ and $f_i : A_i \cap B_i \to V\langle x_i\rangle \cup V\langle x_0\rangle$. Let $I_i = f_i^{-1}(V\langle x_i\rangle)$, $I_j = f_j^{-1}(V\langle x_j\rangle)$.

If $x_j \geqslant x_i$, f_i, f_j compatible and $|I_i|, |I_j| \leqslant \sqrt{pk}$ we define $M[(x_i, (A_i, B_i), f_i), (x_j, (A_j, B_j), f_j)]$ as

$$\mu((A_i, B_i), (A_j, B_j)) \cdot T[x_i, x_j, ((A_j \setminus A_i) \cup I_i, B_j \cup A_i), f_i|_{I_i} \cup f_j|_{I_j}],$$

and zero otherwise.

Intuitively, $M[(x_i, (A_i, B_i), f_i), (x_j, (A_j, B_j), f_j)]$ describes the number of ways to embed $P[A_j \setminus A_i]$ between lines x_i and x_j, where f_i and f_j describe the behaviour of these embeddings on lines x_i an x_j respectively. We observe that we can group isomorphic separations together, i.e. that instead of indexing by every separation, we can index by their representatives and take into account the multiplicities, which are described by μ.

Now we can rewrite the sum $(*)$ as

$$\sum_{(**)} M[(x_\ell, (A \setminus A_{\ell-1}, B \setminus B_{\ell-1}), f_{\ell-1}), (x', (A, B), f)] \prod_{i=0}^{\ell-2} M[(x_i, (A_i, B_i), f_i), (x_{i+1}, (A_{i+1}, B_{i+1}, f_{i+1}))],$$

where the sum $(**)$ goes over $(x_0, (A_0, B_0), f_0), \ldots, (x_{\ell-1}, (A_{\ell-1}, B_{\ell-1}), f_{\ell-1}) \in M_{ind}$. Now by Lemma 3.10, we can compute the sum $(*)$ in time $\ell \cdot |M_{ind}|^2$. Let us bound the size of M_{ind}. Recall that $|\mathcal{S}_{2\sqrt{pk}}(P)| = \sigma_{2\sqrt{pk}}(P)$ and note that $V\langle x_i\rangle$ contains at most $O(k^2 p)$ disks (since we can assume that G can be drawn in a $(O(k) \times O(k))$-box with ply p by Theorem 3.12). Thus we have

$$|M_{ind}| \leqslant k^2 \sigma_{2\sqrt{pk}}(P) \cdot (Ck^2 p)^{\sqrt{pk}}$$

for some constant C. Therefore, we can compute $|\mathrm{sub}(P, G)|$ in time

$$k^{O(\sqrt{pk})} \cdot p^{O(\sqrt{pk})} \cdot \sigma_{O(\sqrt{pk})}(P)^2,$$

which concludes the proof of Theorem 3.1.

3.5 Proof of Theorem 3.3: Bounding σ_s

In this section, we bound the parameter σ_s. We first present a bound for connected graphs.

Theorem 3.13. *For a connected unit disk graph P of ply p with k vertices, and an integer s, we have $\sigma_s(P) \leqslant 2^{O(s \log k)}$.*

Proof. For a separations (A, B) of P of order s, there are $\binom{k}{s} \leqslant k^s \leqslant 2^{s \log k}$ possibilities for $S = A \cap B$. Since any vertex in a unit disk graph can be adjacent to at most 6

pairwise non-adjacent neighbors, the number of connected components of $P - A \cap B$ is at most $6|A \cap B|$. Since (A, B) is a separation, each connected component of $P - S$ is either fully contained in A, or disjoint from A. Therefore we only have at most $2^{6|A \cap B|}$ separations $(A, B) \in \mathcal{S}_s(P)$ with $A \cap B = S$. ■

Theorem 3.14. *For a unit disk graph P of ply p with k vertices, we have that $\sigma_s(P)$ is at most $2^{O(s \log k + pk/\log k)}$.*

Proof. For a separations (A, B) of P of order s, there are $\binom{k}{s}$ possibilities for $S = A \cap B$. We define $t = \frac{\log k}{3p}$. We split all connected components C of $P - S$ into 3 categories as follows:

- *Adjacent Components*: there is an edge between C and S;
- *Small Components*: there is no edge between C and S, and C contains at most t vertices;
- *Large Components*: there is no edge between C and S, and C contains more than t vertices.

Let $\mathcal{C}_{\text{adj}}$ and $\mathcal{C}_{\text{lar}}$ be the sets of all adjacent and large connected components of $G - S$ respectively. Let $\mathcal{C}_{\text{sma}}$ be a maximal set of *non-isomorphic* small connected components. That is, for each small component C of $P - S$ there is exactly one component $C' \in \mathcal{C}_{\text{sma}}$ such that $P[C]$ is isomorphic to $P[C']$.

Note we can encode (A, B) as a quadruple consisting of S, a subset of $\mathcal{C}_{\text{lar}}$, $\mathcal{C}_{\text{adj}}$, and for each $C \in \mathcal{C}_{\text{sma}}$ a number that indicates how many isomorphic copies of C are contained in A. The next step is to bound the number of such quadruples:

$$\binom{k}{s} 2^{|\mathcal{C}_{\text{lar}}| + |\mathcal{C}_{\text{adj}}|} k^{|\mathcal{C}_{\text{sma}}|}. \tag{3.5}$$

We have that $\mathcal{C}_{\text{lar}}$ has at most k/t elements since the components are disjoint and in total amount to at most k vertices. We have that $|\mathcal{C}_{\text{adj}}| \leqslant O(s)$ since each vertex of S has at most 6 neighbors that are pairwise non-adjacent (since P is a unit disk graph).

By Lemma 3.7, the number of unlabelled unit disk graphs with t vertices and ply p is at most $2^{(p+1)t}$ and therefore $|\mathcal{C}_{\text{sma}}| \leqslant 2^{(p+1)t}$. Thus (3.5) is at most

$$2^{s \log k + 3pk/\log k + s + k^{1/3} \log k} = 2^{O(s \log k + pk/\log k)},$$

as claimed. ■

3.6 Proof of Theorem 3.4: Lower Bound

In this section, we give a proof of Theorem 3.4, showing that under ETH there is no algorithm deciding whether $|\text{sub}(P, G)| > 0$ ($|\text{ind}(P, G)| > 0$ respectively) in time $2^{o(n/\log n)}$ even when the ply is two. We will use a reduction from the String 3-Groups problem similar to the one in [26].

Definition 3.15. *The* String 3-Groups *problem is defined as follows. Given sets* $A, B, C \subseteq \{0,1\}^{6\lceil \log n \rceil + 1}$ *of size* n*, find* n *triples* $(a, b, c) \in A \times B \times C$ *such that for all* i*,* $a_i + b_i + c_i \leqslant 1$ *and each element of* A, B, C *occurs exactly once in a chosen triple.*

We call the elements of A, B, C strings. It was shown in [26] that, assuming the ETH, there is no algorithm that solves String 3-Groups in time $2^{o(n)}$. Before stating the formal proof of Theorem 3.4, we give an outline of the main ideas. Given an instance (A, B, C) of String 3-Groups problem, we construct the corresponding host graph G and pattern P as follows. Firstly, we modify slightly the strings in A, B, C to facilitate the construction of P and G. Let m be the length of the (modified) strings. For each $a \in A$, the connected component in G that corresponds to it consists of two paths $p_1 \dots p_m$ and $q_1 \dots q_m$, where p_i and q_i are connected by paths of length 3 if $a_i = 0$. For each $b \in B$, the connected component in P that corresponds to it consists of a path $t_1 \dots t_m$, where there is a path of length two attached to t_i if $b_i = 1$. The connected components corresponding to elements in C are constructed in a similar way. Finally, we add gadgets (triangles and 4-cycles) to each connected component in P and G to ensure we cannot "flip" the components in P. For an example, see Figure 3.2.

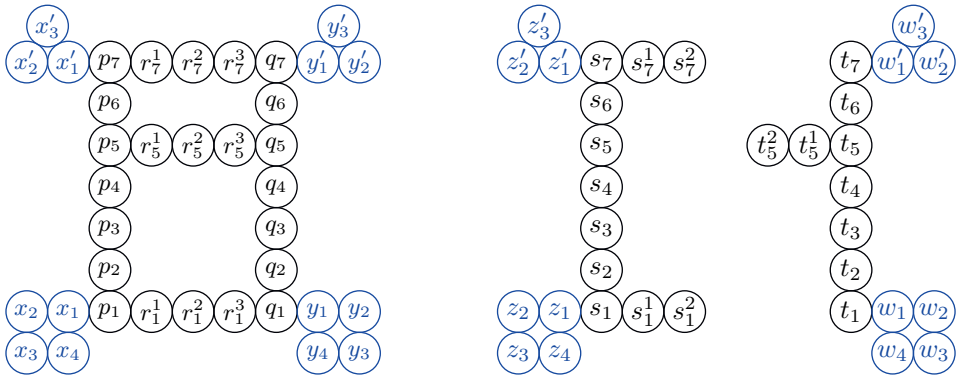


Figure 3.2: Connected components corresponding to $a = 0111010 \in A$ (left), $b = 1000001 \in B$ (middle), $c = 0000100 \in C$ (right).

We present only the proof for the ind case, as the proof for sub is analogous.

Theorem 3.16. *Assuming ETH, there is no algorithm to determine if* $\mathrm{ind}(P, G)$ *is nonempty for given unit disk graphs* P *and* G *in* $2^{o(n/\log n)}$ *time (where* $n = |V(G)|$*), even when* P *and* G *have a given embedding of ply 2.*

Proof. Given an instance A_0, B_0, C_0 of the String 3-Groups problem, we show how to construct an equivalent instance of subgraph isomorphism with host graph G and pattern P. We will first construct another instance A, B, C of the String 3-Groups problem by modifying the given strings as follows:

$$A = \{\tilde{a} : a \in A_0\},\ B = \{\tilde{b} : b \in B_0\},\ C = \{\tilde{c} : c \in C_0\},$$

where

- $\tilde{a}$ is constructed from a by inserting a 1 between every two consecutive characters and 10 at the end;
- $\tilde{b}$ is constructed from b by inserting a 0 between every two consecutive characters and 01 at the end;
- $\tilde{c}$ is constructed from c by inserting a 0 between every two consecutive characters and 00 at the end.

For example, if $a = 010 \in A_0$, $b = 100 \in B_0$, $c = 001 \in C_0$, then $\tilde{a} = 0\underline{1}1\underline{1}0\underline{1}0$, $\tilde{b} = 1\underline{0}0\underline{0}0\underline{1}$, $\tilde{c} = 0\underline{0}0\underline{0}1\underline{0}0$.

It is easy to see that for all $(a, b, c) \in A_0 \times B_0 \times C_0$ we have $a + b + c \leqslant \mathbf{1}$ if and only if $\tilde{a} + \tilde{b} + \tilde{c} \leqslant \mathbf{1}$. Therefore, it is enough to construct an instance of subgraph isomorphism equivalent to A, B, C. Let $m = 12\lceil \log n \rceil + 3$ denote the length of strings in A, B, C.

The graph G will consist of n connected components, representing the elements of A. The graph P will consist of $2n$ connected components, representing elements of B and C. For each string $a \in A$, add to G a connected component that is constructed as follows: Construct two paths, $p_1 \dots p_m$ and $q_1 \dots q_m$. Whenever $a_i = 0$, add a path $p_i r_i^1 r_i^2 r_i^3 q_i$. Construct four cycles: $x_1x_2x_3x_4$, $y_1y_2y_3y_4$, $x'_1x'_2x'_3$ and $y'_1y'_2y'_3$ and add edges x_1p_1, x'_1p_m, y_1q_1, y'_1q_m.

Note that since $a_2 = 0$, so the above paths are connected. By construction, there is no i such that $a_i = a_{i+1} = 0$ (i.e. there is no i such that both $r_i^1 r_i^2 r_1^3$ and $r_{i+1}^1 r_{i+1}^2 r_{i+1}^3$ exist), so we can embed G on the plane as shown in Figure 3.2.

Let us now construct the graph P. For each string $b \in B$, add to P a connected component that is constructed as follows. Construct a path $s_1 \dots s_m$. Whenever $b_i = 1$, add a path $s_i s_i^1 s_i^2$. Construct cycles $z_1z_2z_3z_4$ and $z'_1z'_2z'_3$ and add edges z_1s_1 and $s_mz'_1$.

Similarly, for each $c \in C$, construct a path $t_1 \dots t_m$. Whenever $c_i = 1$, add a path $t_i t_i^1 t_i^2$. Construct cycles $w_1w_2w_3w_4$ and $w'_1w'_2w'_3$ and add edges w_1t_1 and $t_mw'_1$.

Again by construction, there is no i such that $b_i = b_{i+1} = 1$ or $c_i = c_{i+1} = 1$, so we can embed P in the plane as in Figure 3.2.

Given a solution to this String 3-Groups instance, we can embed P into G as follows. If $(a, b, c) \in A \times B \times C$ is a triple in the solution, we map the components corresponding to b and c to the component corresponding to a, by mapping:

- s_i to q_i and t_i to p_i for $i = 1, \dots, m$
- z_i to y_i, w_i to x_i for $i \in \{1, 2, 3, 4\}$ and z'_i to y'_i, w'_i to x'_i for $i \in \{1, 2, 3\}$
- s_i^1, s_i^2 to r_i^3, r_i^2, and t_i^1, t_i^2 to r_i^1, r_i^2 for all $s_i^1, s_i^2, t_i^1, t_i^2 \in V(P)$

This map is well defined: indeed, $a_i + b_i + c_i \leqslant 1$ for each i, so whenever $s_i^1, s_i^2 \in V(P)$ or $t_i^1, t_i^2 \in V(P)$, we have $r_i^1, r_i^2, r_i^3 \in V(G)$. Also, at most one of b_i and c_i is equal to one, so we either have $s_i^1, s_i^2 \in V(P)$ or $t_i^1, t_i^2 \in V(P)$ (or possibly neither), i.e. the map is injective. It is easy to check that the above map maps P to an induced subgraph of G.

Conversely, given an injective homomorphism $f : P \to G$, we can construct a solution to the STRING 3-GROUPS instance as follows. For each $a \in A$, we describe how to find $b \in B$ and $c \in C$ such that $a_i + b_i + c_i \leqslant 1$ for all i.

By a counting argument it is impossible to map more than two connected components of P to the same connected component of G. Since the number of connected components in G and P is n and $2n$ respectively, each connected component in G is therefore the image of exactly two connected components in P. Let $a \in A$, and let G_a be the corresponding connected component in G. We denote the paths of length m in G_a by $p_1 \dots p_m$ and $q_1 \dots q_m$, the paths between p_i and q_i by $r_i^1 r_i^2 r_i^3$, the cycles by $x_1x_2x_3x_4$, $y_1y_2y_3y_4$, $x'_1x'_2x'_3$, $y'_1y'_2y'_3$ as in the above construction.

Let P_1 and P_2 the connected components of P that map to G_a. Note that G_a has exactly two 4-cycles and two triangles, while P_1 and P_2 have one 4-cycle and one triangle each. Therefore, without loss of generality we can assume that the 4-cycle in P_1 is mapped to $y_1y_2y_3y_4$, and the 4-cycle in P_2 to $x_1x_2x_3x_4$. Denote the paths of length m in P_1 (P_2 respectively) by $\bar{s}_1 \dots \bar{s}_m$ ($\bar{t}_1 \dots \bar{t}_m$ respectively). Denote the paths of length 3 by $\bar{s}_i \bar{s}_i^1 \bar{s}_i^2$ ($\bar{t}_i \bar{t}_i^1 \bar{t}_i^2$ respectively).

Claim 3.2. We have $f(\bar{s}_i) = q_i$ for all $i \in [m]$.

Proof of Claim: We know that $f(\bar{s}_1) = q_1$ (since it is the only vertex that is adjacent to the image of the 4-cycle in P_1). Similarly, $f(\bar{t}_1) = p_1$, and $\bar{s}_m$ and $\bar{t}_m$ are mapped to q_m and p_m (not necessarily in that order). Suppose that $f(\bar{s}_m) = p_m$ and $f(\bar{t}_m) = q_m$. Every path of length m between q_1 and p_m contains a "vertical line", i.e. a subpath $\pi = q_i r_i^1 r_i^2 r_i^3 p_i$ for some i. Note that $G_a - \pi$ has two connected components, one containing p_1 and the other containing q_m. This leads to a contradiction, since $f(\bar{t}_1) = p_1$ and $f(\bar{t}_m) = q_m$ need to be connected in $f(P_1) \subseteq G_a - \pi$. Therefore, we have $f(\bar{s}_m) = q_m$ and $f(\bar{t}_m) = p_m$. Using a similar argument, we can conclude that $f(\bar{s}_i) = q_i$ for all $i \in [m]$, which proves the claim. □

Therefore, we have $f(\bar{s}_i^j) = r_i^j$, $f(\bar{t}_i^j) = r_i^j$ for all i and $j = 1, 2, 3$. Let us now look at the strings in $B \cup C$ that correspond to P_1 and P_2. Suppose both P_1 and P_2 correspond to strings in B. Since all strings in B end with a 1, we have both $\bar{s}_m^2 \in V(P_1)$ and $\bar{s}_m^2 \in V(P_2)$, both of which are mapped to r_i^2, which leads to a contradiction. Therefore, at most one of P_1 and P_2 corresponds to a string in B. By a counting argument, we conclude that exactly one of P_1 and P_2 corresponds to a string in B, while the other one corresponds to a string in C. Let P_1 correspond to $b \in B$ and P_2 to $c \in C$.

It remains to show that $a + b + c \leqslant \mathbf{1}$. By construction, whenever $a_i = 1$, the vertex r_i^2 does not exist in G_a, so neither $\bar{s}_i^2$ nor $\bar{t}_i^2$ exist in P. Therefore, $b_i = c_i = 0$. Since f is injective, whenever $a_i = 0$, at most one of $\bar{s}_i^2$ and $\bar{t}_i^2$ exists in $P_1 \cup P_2$, i.e. at most one of b_i and c_i is equal to one. Hence this way we can construct the n triples $(a, b, c) \in A \times B \times C$ such that $a + b + c \leqslant \mathbf{1}$. ■

3.7 Conclusion

We gave (mostly) sub-exponential parameterized time algorithms for computing $|\mathrm{sub}(P, G)|$ and $|\mathrm{ind}(P, G)|$ for unit disk graphs P and G. Since the fine-grained

parameterized complexity of the subgraph isomorphism problem was only recently understood for planar graphs, we believe our continuation of the study for unit disk graphs is very natural, and we hope it inspires further general results.

While our algorithms are tight in many regimes, they are not tight in all regimes. In particular, the (sub)-exponential dependence of the runtime in the ply is not always necessary: We believe the answer to this question may be quite complicated: For detecting some patterns, such as paths, $2^{O(\sqrt{k})}n^{O(1)}$ time algorithms are known [121], but it seems hard to extend it to the counting problem (and to all patterns with few non-isomorphic separations of small order).

For counting induced occurrences with bounded clique size our method can be easily adjusted to get a better dependence in the ply: namely, our method can be used to a get a $(kp)^{O(\sqrt{k})}$ time algorithm for counting independent sets of size k in unit disk graphs of ply p (which is optimal under the ETH by [91]); is there such an improved independence on the ply for each pattern P?

It would be interesting to study the complexity of computing $|\mathrm{sub}(P, G)|$ and $|\mathrm{ind}(P, G)|$ for various pattern classes and various other geometric intersection graphs as well. Our results can be adapted to disk graphs where the ratio of the largest and smallest radius is constant (using a slight modification of Lemma 3.7).

A possible direction for further research would be to determine for which patterns can one compute the above values on bounded ply disk graphs? Recent work [90] shows some problems admit algorithms running in sub-exponential time parameterized time.

Another direction would be to study the subgraph isomorphism problem for more general graphs, e.g. intersection graphs of fat objects. Most of our proofs rely only on a small subset of properties of bounded ply unit disk graphs, e.g. that a line segment of fixed length can only intersect a small number of disks. These properties are not unique to unit disk graphs of bounded ply, but also hold for a larger class of graphs, namely intersection graphs of fat objects.

An even more general question would be to study subgraph isomorphism in higher dimensions. In [36], it was shown that given an intersection graph G of n fat objects in dimension d and a k-vertex graph X_k, one can determine whether X_k is a subgraph of G in time $O(n \log n)$ for constant k. Since the proof in [36] uses several techniques which are similar to those in this chapter, it is natural to ask whether these can be used to obtain a parameterized algorithm for this more general graph class.

Chapter 4

Parameterized Algorithms for Covering by Arithmetic Progressions

4.1 Introduction

In the Set Cover problem we are given a universe U and a set system $\mathcal{S} \subseteq 2^U$ of subsets of U, along with an integer k, and we are asked to determine whether there exist sets $S_1, \ldots, S_k \in \mathcal{S}$ such that $\bigcup_{i=1}^{k} S_i = U$. This problem generalizes many well-known problems, such as Vertex Cover and Feedback Vertex Set (see [45]), albeit the second problem requires an exponential number of elements. Unfortunately, Set Cover is $W[2]$-hard parameterized by k [45, Theorem 13.28], and thus we do not expect an algorithm with FPT runtime.

While the above special cases are FPT parameterized by k, many other special cases of Set Cover remain $W[1]$-hard, and the boundary between special cases that are solvable in FPT and $W[1]$-hard has been thoroughly studied already (see e.g. Table 1 in [31]). An especially famous special case is the Point Line Cover problem, in which one is given points $U \subseteq \mathbb{Z}^2$ and asked to cover them with at most k line segments. While it is a beautiful and commonly used exercise to show this problem is FPT parameterized by k,[1] many slight geometric generalizations (such as generalizing it to arbitrary set systems of VC-dimension 2 [31]) are already $W[1]$-hard.

In this chapter we study another special case of Set Cover, related to *Arithmetic Progressions (APs)*. Recall that an AP is a sequence of integers of the form $a, a + d, a + 2d, \ldots, a + xd$, for some integers x, *start value* a and *difference* d. We study two computational problems: Cover by APs (abbreviated with CAP) and Exact Cover by APs (abbreviated with XCAP). In both problems we are given a set of (distinct) integers $X = \{x_1, x_2, \ldots, x_n\}$, and our goal is to find the smallest number of APs $s_1, s_2, \ldots, s_k$ consisting only of elements of X such that their union covers[2]

[1] A crucial insight is that any line containing at least $k + 1$ points must be in a solution.

[2] We frequently interpret APs as sets by omitting the order, and "covers" can be read as "contains".

exactly the set X. In the XCAP problem, we additionally require that the APs do not have common elements. While CAP and XCAP are already known to be weakly NP-complete since the 90's [71], the problems have been studied surprisingly little since then.

The study of the parameterized complexity of CAP and XCAP can be motivated from several perspectives.

- **The Set Cover perspective:** CAP and XCAP are natural special cases of Set Cover for which the parameterized complexity is a priori not obvious. While the problem looks somewhat similar to Point Line Cover, the crucial insight[1] towards showing that it is FPT in k does not apply since an AP can be covered with 2 other APs. In order to understand the jump in complexity from FPT to $W[1]$-hardness of restricted Set Cover problems better, it is natural to wonder whether properties weaker than the one of Point Line Cover[1] also are sufficient for getting FPT algorithms.

- **The practical perspective:** There is a connection between CAP and XCAP and some problems that arise during the manufacture of VLSI chips [68]. The connection implies the NP-hardness of the latter problems. Bast and Storandt [7, 8] used heuristics for these problems to compress bus timetables and speed up the process of finding the shortest routes in public transportation networks.

- **The additive number theory perspective**: The extremal combinatorics of covers with (generalized) APs is a very well studied topic in the field of additive combinatorics. It started in the 50's with conjectures made by, among others, Erdős (see [42] and the references therein), and recently results in spirit of covering sets of integers with sets of high additive energy (of which APs are a canonical example) such as Freiman's Theorem and the Balog-Szemerédi-Gowers Theorem also found algorithmic applications [32, 37].

- **The "not about graphs" perspective:** Initially, applications of FPT algorithms were mostly limited to graph problems.[3] More recently, FPT algorithms have significantly expanded the realm of their applicability. It now includes geometry, computational social choice, scheduling, constraint satisfiability, and many other application domains. However, at this stage the interaction of number theory and FPT algorithms seems to be very limited.

Our Contributions. The main results of this chapter are FPT algorithms for CAP and XCAP:

Theorem 4.1. CAP *admits an algorithm running in time* $2^{O(k^2)}n^{O(1)}$.

On a high level, the above theorem is proved using the bounded search tree technique (similar to Point Line Cover): In each recursive call we branch on which AP to use. Since k is the number of APs in the solution, the recursion depth is at most k. The difficulty however is to narrow down the number of recursive calls made in

[3] There has even been a series of workshops titled *Parameterized Complexity: Not-About-Graphs* (link) to extend the FPT framework to other fields.

each iteration: As mentioned earlier, the crucial insight[1] from Point Line Cover does not apply since an AP can be covered with two other APs. We achieve this by relying non-trivially on a result (stated in Theorem 4.5) by Crittenden and Vanden Eynden [42] about covering an interval of integers with APs, originally conjectured by Erdős. The proof of Theorem 4.1 is outlined in Section 4.3.

We also give an FPT algorithm for XCAP:

Theorem 4.2. XCAP *admits an algorithm running in time* $2^{O(k^3)}n^{O(1)}$.

The algorithm for XCAP relies on similar, but slightly more involved techniques as the one for CAP. Namely, in order to accommodate the requirement that the selected APs are disjoint we need a more refined recursion strategy. The proof of this theorem is outlined in Section 4.4.

Our next result concerns hardness of variants of CAP and XCAP. The weak NP-hardness proofs in [71] use numbers that are exponentially large in $|X|$, so it is natural to ask if these large numbers are necessary for the reduction, or can they be made sufficiently small, thus proving strong NP-completeness.

While we do not directly make progress on this question, we show that two closely related problems *are* strongly NP-hard. Specifically, if p is an integer, we define an AP in $\mathbb{Z}_p$ as a sequence of the form

$$a, a + d \pmod p, a + 2d \pmod p, \ldots, a + xd \pmod p.$$

In the Cover by Arithmetic Progressions in $\mathbb{Z}_p$ problem one is given as input an integer p and a set $X \subseteq \mathbb{Z}_p$ and asked to cover X with APs in $\mathbb{Z}_p$ that are contained in X and cover X. In Exact Cover by Arithmetic Progressions in $\mathbb{Z}_p$ we additionally require the APs to be disjoint. We show the following:

Theorem 4.3. Cover by Arithmetic Progressions in $\mathbb{Z}_p$ *and* Exact Cover by Arithmetic Progressions in $\mathbb{Z}_p$ *are strongly NP-complete.*

While this may hint at strong NP-completeness for CAP and XCAP as well, since often introducing a (big) modulus does not incur big jumps in complexity in number theoretic computational problems (confer e.g. k-SUM, Partition, etc.), we also show that our strategy cannot directly be used to prove CAP and XCAP to be strongly-NP. Thus this still leaves the mentioned question of Heath [71] open. This result is proven in Section 4.5.

Finally, we describe a variant of Set Cover that generalizes CAP and allows an FPT algorithm for a certain below guarantee parameterization. In particular, CAP always has a solution consisting of $|X|/2$ APs that cover all sets.

Theorem 4.4. *There is an* $2^{O(k)}n^{O(1)}$ *time algorithm that detects if a given set* X *of integers can be covered with at most* $|X|/2 - k$ *APs.*

This result is proved in Section 4.6. Note that Theorem 4.4 typically gives a faster algorithm when there are few (non-trivial) APs in the input set, whereas Theorem 4.1 gives a faster algorithm when there are many APs needed to cover the input set. We finish this chapter with some concluding remarks in 4.7.

4.2 Preliminaries

For integers a, b we write $a|b$ to denote that a divides b. For integers $a_1, \ldots, a_n$ we denote their greatest common divisor by $gcd(a_1, \ldots, a_n)$. Recall that every integer divides zero, so $gcd(0, a_1, \ldots, a_n) = gcd(a_1, \ldots, a_n)$

An *arithmetic progression* (AP) is a sequence of numbers such that the difference of two consecutive elements is the same. While an AP is a sequence, we will often identify an AP with the set of its elements. Given an AP s and an AP s' which is a subset of s, we say s' *stops between* $l \in s$ and $r \in s$ if the largest element of s' is between l and r (including both boundaries). We say it *covers* a set A if all integers in A occur in it. Given an AP $a, a+d, a+2d, \ldots$ we call d the *difference*. Given a finite AP $T = \{a, a+d, \ldots, a+xd\}$, we define its infinite extension as $T' = \{a + yd : y \in \mathbb{N} \cup \{0\}\}$ (if T consists of one element, T' will also have only one element, i.e. it will be the infinite AP with difference 0). We record the following easy observation:

Observation 4.1. *The intersection of two APs is an AP.*

If X is a set of integers, we denote

$$X^{>c} = \{x | x \in X, x > c\}, \quad X^{\geqslant c} = \{x | x \in X, x \geqslant c\},$$
$$X^{<c} = \{x | x \in X, x < c\}, \quad X^{\leqslant c} = \{x | x \in X, x \leqslant c\}.$$

Given a set X and an AP $A = \{a, a+d, a+2d, \ldots\}$, we denote by $A \sqcap X$ the longest prefix of A that is contained in X. In other words, $A \sqcap X = \{a, a+d, \ldots, a+\ell d\}$, where ℓ is the largest integer such that $a + \ell' d \in X$ for all $\ell' \in [0, \ell]$.

For a set of integers X and integer p we denote by $X_p = \langle x \bmod p | x \in X \rangle$. Here the $\langle\rangle$ symbols indicate that we build a *multiset* instead of a set (so each number is replaced with its residual class mod p and we do not eliminate copies).

We call an AP s *infinite* if there are integers a, d such that $s = (a, a+d, a+2d, \ldots)$. Note that under this definition, the constant AP containing only one number and difference 0 is also infinite.

The following result by Crittenden and Vanden Eynden will be crucial for many of our results:

Theorem 4.5 ([42]). *Any k infinite APs that cover the integers $\{1, \ldots, 2^k\}$ cover the whole set of positive integers.*

4.3 Algorithm for Cover by Arithmetic Progressions

Before describing the algorithm, we introduce an auxiliary lemma. This lemma will be crucially used to narrow down the number of candidates for an AP to include in the solution to at most 2^k.

Lemma 4.6. *Let s_0 be an AP with at least $t+1$ elements. Let $s_1, \ldots, s_k$ be APs that cover the elements $s_0(0), \ldots, s_0(t-1)$, but not $s_0(t)$. The APs $s_1, \ldots, s_k$ may contain other elements. Suppose that each AP $s_1, \ldots, s_k$ has an element larger than $s_0(t)$. Then we have $t < 2^k$.*

Proof. Suppose for contradiction that $t \geqslant 2^k$. Let t_i be the intersection of s_i and s_0 for $i \in [k]$. Note that by Observation 4.1, $t_1, \ldots, t_k$ are APs as well. For $i \in [k]$, we define $T_i = \{j \in [0, t-1] : s_0(j) \in t_i\}$.

We claim each T_i is an AP: To see this, let a, b, c be consecutive elements (such that $a < b < c$) of T_i (if $|T_i| \leqslant 2$, it is trivially an AP). The elements $s_0(a), s_0(b), s_0(c)$ are consecutive elements of t_i, so $s_0(b) - s_0(a) = s_0(c) - s_0(b)$. Let d_0 be the difference of s_0. The above equality can be written as

$$(s_0(0) + bd_0) - (s_0(0) + ad_0) = (s_0(0) + cd_0) - (s_0(0) + bd_0).$$

Thus we get $b - a = c - b$, as required.

Note that the APs $T_1, \ldots, T_k$ cover $[0, t-1]$. For $i \in [k]$, denote by T_i' the infinite extension of T_i. By Theorem 4.5, $T_1', \ldots, T_k'$ cover the whole set $\mathbb{N}$. In particular, $t \in T_i'$ for some i. By assumption, s_i has an element larger than $s_0(t)$, so $s_0(t)$ is covered by s_i, which leads to a contradiction. ∎

Equipped with Lemma 4.6 we are ready to prove our first main theorem in this chapter:

Theorem 4.1. CAP *admits an algorithm running in time* $2^{O(k^2)} n^{O(1)}$.

Proof. Denote the set of integers given in the input by X. Without loss of generality we can consider only solutions where all APs are inclusion-maximal, i.e. solutions where none of the APs can be extended by an element of X. In particular, given an element a and difference d, the AP is uniquely determined: it is equal to $\{a - \ell d, \ldots, a - d, a, a + d, \ldots, a + rd\}$, where ℓ, r are largest integers such that $a - \ell' d \in X$ for all $\ell' \in [\ell]$ and $a + r'd \in X$ for all $r' \in [r]$.

Before describing the algorithm, we introduce an auxiliary procedure called MakeAP, returning the maximal AP with given one element and difference. Formally, the procedure MakeAP(a, d) returns the AP $\{a - \ell_1 d, \ldots, a - d, a, a + d, \ldots, a + \ell_2 d\}$, where ℓ_1 (respectively ℓ_2) are the largest numbers such that $a - \ell' d \in X$ for all $\ell' \leqslant \ell_1$ (respectively $a + \ell' d \in X$ for all $\ell' \leqslant \ell_2$).

Our algorithm consists of a recursive function Covering(C, k_1, k_2). The algorithm takes as input a set $C \subseteq X$ of elements and assumes there are APs $s_1, \ldots, s_{k_1}$ whose union equals C. With this assumption, it will detect correctly whether there exist k_2 additional APs that cover $X \setminus C$, i.e. all remaining elements. We call the elements of C covered, and the elements of $X \setminus C$ uncovered. Thus Covering$(\emptyset, 0, k)$ indicates whether the instance is a Yes-instance of CAP. Algorithm 1 describes our algorithm in pseudocode.

In Line 4 we take care of the small instances, i.e. we solve the subproblem of covering $X \setminus C$ with k_2 APs if $|X \setminus C| \leqslant k^2$ in time $2^{k^2} n^{O(1)}$. This can be easily done by rewriting the subproblem as an equivalent instance of Set Cover (with universe $U = X \setminus C$ and a set for each AP in U) and running the $2^{|U|} n^{O(1)} \leqslant 2^{k^2} n^{O(1)}$ time algorithm for Set Cover from [15].

For larger instances, we guess an AP s in the solution and recursively call Covering$(C \cup s, k_1 + 1, k_2 - 1)$. In order to do this, we consider the $k^2 + 1$ smallest uncovered elements $u_1, \ldots, u_{k^2+1}$. By the pigeonhole principle, there exist u_i and u_j

Algorithm 1 Algorithm for CAP

```
1: Algorithm Covering(C, k_1, k_2)
2: Let k = k_1 + k_2
3: if |X \ C| ⩽ k^2 then
4:     Use the algorithm for Set Cover from [15]
5: else
6:     Let u_1, ..., u_{k^2+1} be the k^2 + 1 smallest elements of X \ C
7:     for i = 1 ... k^2 do
8:         for j = i + 1 ... k^2 + 1 do
9:             Let D = u_j − u_i
10:            for ℓ = 1 ... 2^k do
11:                if ℓ divides D then
12:                    Let s = MakeAP(u_i, D/ℓ)
13:                    if Covering(C ∪ s, k_1 + 1, k_2 − 1) then
14:                        return true
15:    return false
```

which belong to the same AP s in the solution, and we can guess (i.e. branch over all possibilities) such u_i and u_j.

However, we cannot guarantee that u_i and u_j are consecutive elements of s: we only know that the difference of s divides $u_j - u_i$. The naive approach would be to go over all divisors of $u_j - u_i$, but this is too slow (i.e. would not lead to an FPT algorithm).

Instead, we reduce the number of candidates for the difference of s by using Lemma 4.6 as follows. Intuitively, Claim 4.1 says that in some iteration of the **for** loops, the AP that we construct in Line 12 will belong to the solution.

Claim 4.1. Let X be a finite set of integers and let $s_1, \ldots, s_k$ be inclusion-maximal APs that cover X. Let $k_1 \in [0, k]$ and $C = \cup_{i=1}^{k_1} s_i$. Let $u_1, \ldots, u_{k^2+1}$ be the $k^2 + 1$ smallest elements of $X \setminus C$. Then there exist $i, j \in [k^2 + 1]$ and $\ell \in [2^k]$ such that the AP $s = \textsc{MakeAP}(u_i, (u_j - u_i)/\ell)$ is equal to s_h for some $h \in [k_1 + 1, k]$.

Proof of Claim: Let $B = \{u_1, \ldots, u_{k^2+1}\}$. By the pigeonhole principle, there is an $h \in [k_1 + 1, k]$ such that s_h covers at least $k + 1$ elements of B. Thus, there are two consecutive elements of $B \cap s_h$, denoted with $s_h(\alpha)$ and $s_h(\beta)$, between which no AP $s_1, \ldots, s_{k_1}$ ends. Let $u_i = s_h(\alpha)$ and $u_j = s_h(\beta)$ be the closest such pair (i.e. the pair such that $|s_h(\alpha) - s_h(\beta)|$ is minimal). By applying Lemma 4.6 (for s_0 equal to $s_h(\alpha+1), \ldots, s_h(\beta-1)$ and $s_1, \ldots, s_{k_1}$), we conclude that there are at most $2^k - 1$ elements of s_h between u_i and u_j. In other words, the difference of s_h is a divisor of $u_j - u_i$ and at least $(u_j - u_i)/2^k$, i.e. there exists an $\ell \in [2^k]$ such that $s_h = \textsc{MakeAP}(u_i, (u_j - u_i)/\ell)$. □

Let us now prove the correctness of our algorithm. It is easy to see that if Algorithm 1 outputs **true**, we indeed have a covering of size k: We only add elements to C if they are indeed covered by an AP, and each time we add elements to C because of an AP we decrease our budget k_2.

For the other direction of correctness, using Claim 4.1 it directly follows by induction on $k_2 = 0, \ldots, k$ that if $s_1, \ldots, s_k$ is a covering of X with inclusion-maximal APs such that $\cup_{i=1}^{k_1} s_i = C$, then the function COVERING$(C, k - k_2, k_2)$ returns true.

Let us now analyse the running time of Algorithm 1. The recursion tree has height k (since we reduce k by one on every level). The maximum number of children of a node is $2^k k^4$, so the total number of nodes is $2^{O(k^2)}$. The running time at each node is at most $2^{O(k^2)} n^{O(1)}$. Therefore, the overall running time is $2^{O(k^2)} n^{O(1)}$. ■

4.4 Algorithm for Exact Cover by Arithmetic Progressions

In this section we show that XCAP is FPT as well. The key difference between CAP and XCAP is that in XCAP we cannot assume that APs are inclusion-maximal: Consider for example $X = \{0, 4, 6, 7, 8, 9\}$, where the optimal solution uses the AP 0,4 rather than 0,4,8. This also means that in XCAP we cannot describe an AP using only one element and its difference. While we still use a recursive algorithm, we need to significantly modify our structure lemma (shown below in Lemma 4.8) and our algorithm.

Before describing an FPT algorithm, let us describe an easy, but slow (i.e. not FPT) algorithm. In order to obtain an FPT algorithm, we will need to modify a part of it. Let $X = \{a_1, \ldots, a_n\}$ be the input set, where $a_1 < \cdots < a_n$. We will construct APs $s_1, \ldots, s_k$ that are pairwise disjoint and cover X.

Naive Algorithm. Intuitively, our algorithm works as follows. As mentioned earlier, as opposed to CAP, we cannot uniquely describe an AP by one element and its difference, since we do not know if our AP will be "interrupted" by another one. We will instead maintain two sets for each AP, the set of "truly covered" and a set of "potentially covered" elements. Namely, if we know two elements $a < b$ of an AP s and its difference d, we say that the elements $a, a + d, \ldots, b - d, b$ are truly covered by s, while elements $b + d, b + 2d, \ldots$ are potentially covered by s.

Our recursive function will have parameters that correspond to the sets of truly and potentially covered elements (T_i and P_i respectively), as well as the difference (d_i) for each AP. Initially, we set all these sets to be the empty set, and all the differences to 0. In each recursive call, we consider the smallest element of X that is neither covered nor potentially covered, and we guess (i.e. branch) which AP covers it. Whenever there is an AP which has two truly covered elements but we do not know its difference (i.e. the corresponding d_i is zero), we branch to discover its difference.

Once we assign a non-zero number to some d_i, we do not change it in further recursive calls. Similarly, once we add an element to some T_i, we do not remove it from T_i (however, elements of P_i can be removed).

Before giving the algorithm, we describe an auxiliary function UPDATE. It takes as input sets A and B, and returns elements of A that are smaller than all elements of $A \cap B$ (see Algorithm 2).

Our algorithm for XCAP is described in pseudocode in Algorithm 3.

Algorithm 2 Auxiliary Function UPDATE

UPDATE(A, B)
if $A \cap B = \emptyset$ **then**
    return A
else
    $x \leftarrow \min(A \cap B)$
    return $A^{<x}$

Initially, we call the function PARTITION with parameters $T_1 = \cdots = T_k = P_1 = \cdots = P_k = \emptyset, d_1 = d_2 = \cdots = d_k = 0$.

Algorithm 3 Algorithm for XCAP

Algorithm PARTITION$(X, T_1, \ldots T_k, P_1, \ldots, P_k, d_1, \ldots, d_k)$
$T \leftarrow \cup_i T_i$
$P \leftarrow \cup_i P_i$
if $X \setminus (P \cup T) = \emptyset$ and $P_i \cap P_j = \emptyset$ for all $i \neq j$ **then**
    return $s_1 = T_1 \cup P_1, \ldots, s_k = T_k \cup P_k$
else if $d_i \neq 0$ for all i **then**
    abort
else if there is an i such that $|T_i| \leqslant 1$ **then**
    $a_\beta \leftarrow \min(X \setminus (P \cup T))$
    for $i \in [k]$ such that $|T_i| \leqslant 1$ **do**
        $T_i' \leftarrow T_i \cup \{a_\beta\}$
        PARTITION$(X, T_1, \ldots, T_i', \ldots T_k, P_1, \ldots, P_k, d_1, \ldots, d_k)$
else if there is an i such that $|T_i| = 2$ and $d_i = 0$ **then**
    $a_\alpha \leftarrow \min(T_i)$
    $a_\beta \leftarrow \max(T_i)$
    $D \leftarrow$ all divisors of $a_\beta - a_\alpha$
    for $d \in D$ **do**
        $d_i \leftarrow d$
        $T_i \leftarrow \{a_\alpha, a_\alpha + d_i, \ldots, a_\beta\}$
        $C^\infty \leftarrow \{a_\beta + d_i, a_\beta + 2d_i, \ldots\}$
        for $j \in [k]$ **do**
            $P_j' \leftarrow P_j$
        $P_i' \leftarrow C^\infty \sqcap X$
        if for all $j \in [k] \setminus \{i\}$ we have $T_i \cap T_j = \emptyset$ **then**
            for $j \in [k] \setminus \{i\}$ **do**
                $P_j' \leftarrow$ UPDATE(P_j, T_i)
            PARTITION$(X, T_1, \ldots, T_k, P_1', \ldots, P_k', d_1, \ldots, d_k)$
else if there are $i, j \in [k]$ s.t. $i \neq j$ and $P_i \cap P_j \neq \emptyset$ **then**
    Let c be the smallest element s.t. $\exists i, j$, $c \in P_i \cap P_j$, and $i < j$
    PARTITION$(X, T_1, \ldots, T_k, P_1, \ldots, P_{i-1}, P_i^{<c}, P_{i+1}, \ldots, P_k, d_1, \ldots, d_k)$
    PARTITION$(X, T_1, \ldots, T_k, P_1, \ldots, P_{j-1}, P_j^{<c}, P_{j+1}, \ldots, P_k, d_1, \ldots, d_k)$

Formally, the function PARTITION$(X, T_1, \ldots, T_k, P_1, \ldots, P_k, d_1, \ldots, d_k)$ returns APs $s_1, \ldots, s_k$ with differences $d'_1, \ldots, d'_k$ respectively, such that the following is true: For all $i \in [k]$, we have that $T_i \subseteq s_i$, and if $d_i \neq 0$, we have $d_i = d'_i$ and $s_i \subseteq T_i \cup P_i$. If the above conditions are satisfied, we call the APs $s_1, \ldots, s_k$ *consistent* with $T_1, \ldots, T_k, P_1, \ldots, P_k, d_1, \ldots, d_k$. The function PARTITION consists of four parts:

- Lines 4-7: we return the solution if all elements are covered and the APs don't intersect, or abort the branch if it does not lead to a solution.
- Lines 8-12: we look for the smallest element that is neither covered nor potentially covered, and guess to which AP it belongs. Note that, by construction, it cannot belong to an AP which covers two or more elements (i.e. $|T_i| \geqslant 2$).
- Lines 13-27: we consider the case where an AP has two truly covered elements, but we do not know its difference yet (i.e. $a_\alpha, a_\beta \in T_i$, $d_i = 0$). We know that the difference of this AP divides $a_\beta - a_\alpha$, so we can guess it by branching over all divisors of $a_\beta - a_\alpha$. We update the corresponding T_i and P_i. If the newly created T_i disjoint from all other T_js, we update all other P_js to make sure that they do not intersect T_i.
- Lines 28-31: we resolve conflicts between intersecting APs by guessing which AP stops before the intersection, and which continues.

Lemma 4.7. *Let $d_{max} \in \mathbb{N}$ be such that in each recursive call of Algorithm 3 we have $|D| \leqslant d_{max}$. Then Algorithm 3 solves* XCAP *in time $2^{O(k^2)} \cdot d_{max}^k \cdot n^{O(1)}$.*

Proof. It is easy to see that in each recursive call, $T_i \cup P_i$ is an AP (or an empty set) for each $i \in [k]$, and that the APs returned by the algorithm form a valid solution to XCAP.

Consider an iteration of PARTITION with arguments $X, T_1, \ldots, T_k, P_1, \ldots, P_k, d_1, \ldots, d_k$. Let $s_1, \ldots, s_k$ be APs with increments $d'_1, \ldots, d'_k$ that form a solution to XCAP and are consistent with $T_1, \ldots, T_k, P_1, \ldots, P_k, d_1, \ldots, d_k$. Clearly, if $X \setminus (P \cup T) = \emptyset$ (and the APs do not intersect), we return $s_1, \ldots, s_k$. Suppose that $X \setminus (P \cup T) \neq \emptyset$.

We claim that at least one of the children in the recursion tree remains consistent with $s_1, \ldots, s_k$. In Line 12, we go over all possibilities for the AP that contains a_β (note that a_β cannot belong to s_i such that $|T_i| \geqslant 2$, since in that case it would belong to $T_i \cup P_i$). In particular, one of the children remains consistent with $s_1, \ldots, s_k$. In Line 16, we know that a_α and a_β belong to s_i, so we know that d'_i divides $a_\beta - a_\beta$, i.e. $d'_i \in D$. Thus in one of the recursive calls in Line 27 we will have $d_i = d'_i$. In Lines 30 and 31, we have a recursive call for both possible cases.

Let us now compute the number of nodes in the recursion tree: Consider a path from the root to a leaf. It contains at most $2k$ nodes with k children (adding first two elements to each T_i), at most k^2 nodes with two children (resolving intersections of APs) and at most k nodes with at most d_{max} children (determining the difference of an AP with two known elements). The number of leaves (i.e. the number of such paths) is at most $3^{k^2+3k} \cdot k^{2k} \cdot 2^{k^2} \cdot d_{max}^k = 2^{O(k^2)} \cdot d_{max}^k$. Thus the running time of the algorithm is $2^{O(k^2)} \cdot d_{max}^k \cdot n^{O(1)}$. ∎

In order to obtain an FPT algorithm, we will change Line 16 to obtain a smaller set D. Namely, we will use the following lemma to bound the number of possibilities for the difference of an AP with two known elements.

Lemma 4.8. *Let $s_1, s_2, \ldots, s_k$ be a solution of an instance of* XCAP *with input set X. Let s be an AP that is contained in X. For each $i \in [k]$, we denote by t_i the intersection of s and s_i. Suppose that for some i, t_i has at least $k+1$ elements. Then there are at most $2^{k-1}-1$ elements of s between any two consecutive elements of t_i.*

Proof. Without loss of generality, we may assume that $i = 1$. Note that t_1 is a subset of s and an AP. Hence between any two consecutive members of t_1 we have the same number of elements of s. Suppose for contradiction that this number is at least 2^{k-1}. Let us consider the first $k+1$ members of t_1, denoted by $t_1(0), t_1(1), \ldots, t_1(k)$. For any $j \in [0, k-1]$ all elements of s between $t_1(j)$ and $t_1(j+1)$ must be covered by $s_2, s_3, \ldots, s_k$. Therefore, by Lemma 4.6 one of the arithmetic progressions $s_2, s_3, \ldots, s_k$ must stop before $t_1(1)$. Similarly, another AP from $s_2, s_3, \ldots, s_k$ must stop before $t_1(2)$. By repeatedly applying this argument, we conclude that all APs $s_2, s_3, \ldots, s_k$ must stop before $t_1(k-1)$. Hence, the elements of s between $t_1(k-1)$ and $t_1(k)$ are uncovered which leads to a contradiction. ■

Now we have all the ingredients to prove the main theorem of this section:

Theorem 4.2. XCAP *admits an algorithm running in time $2^{O(k^3)}n^{O(1)}$.*

Proof. We replace Line 16 in Algorithm 3 by the following expression:

$$D = \{g/t : \ g = gcd(a_\beta - a_\alpha, b_1 d_1, \ldots, b_k d_k)$$
$$\text{for some } (b_1, \ldots, b_k) \in [0, 2^k + 1]^k \text{ and } t \in [k(k+1)]\}.$$

Let us now show that it is indeed enough to consider only the elements of the above set D for the possible difference. Consider the recursive call of PARTITION where we have $|T_i| = 2$ and $d_i = 0$. Suppose that in this call, the sets $T_1, \ldots, T_k$ are consistent with a solution $s_1, \ldots, s_k$ for XCAP, and let d'_j be the difference of s_j.

Informally, for all known differences d_j up to this point (i.e. all $d_j \neq 0$) we branch on the smallest positive value b_j such that $d'_i | b_j d_j$. We treat all cases when $b_j > 2^k$ at once, and instead of the actual value we assign 0 to the variable responsible for storing value of b_j. Intuitively, b_j describes the number of elements of s_j between two consecutive interruptions by s_i. By Lemma 4.8, if b_j is large (larger than $2^k + 1$), each of these interruptions implies that an AP stops.

Formally, we apply the following claim:

Claim 4.2. Consider a recursive call of PARTITION where $|T_i| = 2$ and $d_i = 0$. Suppose $T_1, \ldots, T_k, P_1, \ldots, P_k, d_1, \ldots, d_k$ is consistent with a solution $s_1, \ldots, s_k$ to XCAP, and let d'_i be the difference of s_i. Then there exist $b_1, \ldots, b_k \in [0, 2^k]$ such that $d'_i \geqslant \frac{g(b_1,\ldots,b_k)}{k(k+1)}$, where $g(b_1, \ldots, b_k) = gcd(a_\beta - a_\alpha, b_1 d_1, \ldots, b_k d_k)$.

Proof of Claim: Let $T_i = \{a_\alpha, a_\beta\}$, where $a_\alpha < a_\beta$. Note that a_β was added to T_i in Line 9 as the smallest element that was neither in T nor in P. Therefore, the elements $a_{\alpha+1}, \ldots, a_{\beta-1}$ belong to $T \cup P$. Let $S_i = \{a \in s_i : \ a_\alpha < a < a_\beta\}$. By the

above observation, all elements of S_i are covered, i.e. in $T \cup P$. Since the solution $s_1, \ldots, s_k$ is consistent with the parameters of the current recursive call, we know that no element of S_i is trully covered (i.e. in some T_j). In other words, each element of S_i belongs to some P_j for $j \neq i$. We consider two cases:

Case 1: Suppose that each P_j contains at most k elements of S_i. Then S_i contains at most k^2 elements, so $d'_i \geqslant (a_\beta - a_\alpha)/(k^2+1) \geqslant (a_\beta - a_\alpha)/k(k+1)$. Note that $g(0, \ldots, 0) = a_\beta - a_\alpha$, so we have $d'_i \geqslant \frac{g(0,\ldots,0)}{k(k+1)}$.

Case 2: Suppose that some of the sets P_j contain at least $k+1$ elements of S_i. Let $q_1, \ldots, q_u$ be distinct integers such that for each $j \in [u]$, $|P_{q_j} \cap S_i| \geqslant k+1$ and for each $j \notin \{q_1, \ldots, q_u\}$, $|P_j \cap S_i| \leqslant k$. Let b_{q_j} be the number of elements of P_{q_j} (strictly) between two consecutive elements of the AP $P_{q_j} \cap s_i$. The difference of $P_{q_j} \cap s_i$ is equal to $(b_{q_j}+1)d_{q_j}$, so d'_i divides $(b_{q_j}+1)d_{q_j}$. By Lemma 4.8 (applied to $s_1, \ldots, s_k$ and $s = P_{q_j}$), we get $b_{q_j} \leqslant 2^{k-1} - 1$.

Let $g = gcd(a_\beta - a_\alpha, (b_{q_1}+1)d_{q_1}, \ldots, (b_{q_u}+1)d_{q_u})$. Applying the above observation to each $j \in [u]$, we conclude that d'_i divides g. It remains to show that $d'_i \geqslant g/k(k+1)$. In other words, we need to show that s_i has less than $k(k+1)$ elements in the interval $(a_\alpha, a_\alpha + g)$. Suppose that for some q_j, $P_{q_j} \cap s_i$ contains at least two elements, c_1 and c_2, in the interval $(a_\alpha, a_\alpha + g)$. Note that $|c_2 - c_1|$ is divisible by $(b_{q_j}+1)d_{q_j}$, so $|c_2 - c_1|$ is divisible by g. This leads to a contradiction, since $|c_2 - c_1| < g$. Thus for each j, $P_{q_j} \cap s_i$ contains at most one element in $(a_\alpha, a_\alpha + g)$. For $j \notin \{q_1, \ldots, q_u\}$, we know that $|P_j \cap s_i| \leqslant k$, so there are at most $k(k-1) + k < k(k+1)$ elements of s_i in $(a_\alpha, a_\alpha + g)$, which proves the claim. □

Using Lemma 4.7, the total complexity of our algorithm is $2^{O(k^3)} n^{O(1)}$. ■

4.5 Strong NP-hardness of Cover by Arithmetic Progressions in $\mathbb{Z}_p$

A natural question to ask, in order to prove Strong NP-hardness for CAP, is whether we can replace an input set X with an equivalent set which has smaller elements. Specifically, could we replace the input number with numbers polynomial in $|X|$, while preserving the set of APs? This intuition can be further supported by results on Simultaneous Diophantine approximation that exactly achieve results in this spirit (though with more general properties and larger upper bounds) [64]. However, it turns out that this is not always the case in our setting, as we show in Lemma 4.9. This means that one of the natural approaches for proving strong NP-hardness of CAP does not work: namely, not all sets X can be replaced with set a X' of polynomial size which preserves all APs in X.

Lemma 4.9. *Let $x_1 = 0$, $x_i = 2^{i-2}$ for $i \geqslant 2$ and $X_n = \{x_1, \ldots, x_{n+2}\}$. Then for any polynomial p there exists an integer n such that there is no set $A_n = \{a_1, \ldots, a_{n+2}\}$ that satisfies the following criteria:*

- *$a_i \leqslant p(n)$ for all $i \in [n+2]$,*

- *For all $i, j, k \in [n+2]$, the set $\{x_i, x_j, x_k\}$ forms an AP if and only if $\{a_i, a_j, a_k\}$ forms an AP.*

Proof. Assume for contradiction that there is a polynomial p such that for any n, we can construct a set A_n with elements smaller than $p(n)$ which preserves the APs of size 3 in X_n. Without loss of generality, we may assume that $a_1 = 0$ (by subtracting a_1 from all elements of A_n and using $2p$ instead of p).

Clearly if $|X_n| \geqslant 4$ we must have $a_2 \neq 0$. If a set $\{0, a, b\}$ generates an AP, then $b \in \{2a, -a, \frac{a}{2}\}$. We know that x_1, x_j, x_{j+1} generate an AP for each $j \in [2, n-1]$, which implies that for $i \geqslant 3$, a_i is equal to a_{i-1} multiplied by $2, -1$ or $\frac{1}{2}$. Note that we cannot have $q_1 < q_2$ such that $a_{q_1} = a_{q_2}$. Indeed, if $q_2 > q_1 + 1$ then a_1, a_{q_1}, a_{q_2} generates an AP while x_1, x_{q_1}, x_{q_2} does not. It is left to consider case $q_2 = q_1 + 1$. If $q_2 < n$, X_n contains the AP x_1, x_{q_2}, x_{q_2+1}, so $\{a_1, a_{q_2}, a_{q_2+1}\}$ is an AP. Since $a_{q_2} = a_{q_1}$, we have that $\{a_1, a_{q_1}, a_{q_2+1}\}$ also generates an AP which contradicts the fact that x_1, x_{q_1}, x_{q_2+1} is not an AP. If $q_2 = n$, then instead of the triple $\{a_1, a_{q_2}, a_{q_2+1}\}$ we consider the triple $\{a_1, a_{q_1-1}, a_{q_1}\}$ and instead of x_1, x_{q_1}, x_{q_2+1} we consider x_1, x_{q_1-1}, x_{q_2} and get the contradiction as $n \geqslant 4$. Therefore all numbers $a_1, a_2, \ldots, a_n$ must be different.

Recall that for each $j > 1$ we have either $a_j = 2^{p_j} a_2$ or $a_j = -2^{p_j} a_2$ for some integer p_j. As all numbers in the set $\{a_1, a_2, \ldots, a_n\}$ are different, we have that there is an index j such that $p_j > \frac{n}{8}$ or $p_j < -\frac{n}{8}$. Since all numbers in A must be integers we conclude that either $a_2 > 2^{\frac{n}{8}}$ or $a_{p_j} > 2^{\frac{n}{8}}$. Therefore we must have $p(n) \geqslant 2^{\frac{n}{8}}$, which is not true if we take n sufficiently large. ■

Unfortunately, we do not know how to directly circumvent this issue and improve the weak NP-hardness proof of Heath [71] to *strong* NP-hardness. Instead, we work with another variant of the problem in which we work in $\mathbb{Z}_p$. The definition of APs naturally carries over to $\mathbb{Z}_p$. It is easy to see that APs are preserved:

Observation 4.2. *Let p be a prime and let X be a set of integers that forms an AP. Then the multiset X_p generates an AP in the field $\mathbb{Z}_p$.*

Indeed, note that $y - x = z - y$ implies $(y \bmod p) - (x \bmod p) \equiv_p (z \bmod p) - (y \bmod p)$, so we have that X_p is an AP in $\mathbb{Z}_p$.

However, the converse does not hold. Formally, if for some p the multiset X_p is an AP in $\mathbb{Z}_p$ it is not necessarily true that X is an AP in $\mathbb{Z}$. For example, consider $X = \{3, 6, 18\}$ and $p = 3$: we have $X_p = \{0, 0, 0\}$ which is a (trivial) AP, while X is not an AP.

We now show strong NP-completeness for the modular variants of CAP and XCAP. In the Cover by Arithmetic Progressions in $\mathbb{Z}_p$ problem one is given as input an integer p and a set $X \subseteq \mathbb{Z}_p$ and asked to cover X with APs in $\mathbb{Z}_p$ that are contained in X that cover X. In Exact Cover by Arithmetic Progressions in $\mathbb{Z}_p$ we additionally require the APs to be disjoint.

Theorem 4.3. Cover by Arithmetic Progressions in $\mathbb{Z}_p$ *and* Exact Cover by Arithmetic Progressions in $\mathbb{Z}_p$ *are strongly NP-complete.*

Proof. We recall that Heath [71] showed that CAP and XCAP are weakly NP-complete via reduction from Set Cover. Moreover, the instances of CAP and

XCAP, obtained after reduction from SET COVER, consist of numbers that are bounded by $2^{q(n)}$ for some polynomial $q(n)$. To show that COVER BY ARITHMETIC PROGRESSIONS IN $\mathbb{Z}_p$ and EXACT COVER BY ARITHMETIC PROGRESSIONS IN $\mathbb{Z}_p$ are strongly NP-complete we take a prime p and convert instances of CAP and XCAP with set S into instances of COVER BY ARITHMETIC PROGRESSIONS IN $\mathbb{Z}_p$ and EXACT COVER BY ARITHMETIC PROGRESSIONS IN $\mathbb{Z}_p$ respectively with a set S_p (we can guarantee that the multiset S_p contains no equal numbers) and modulo p.

As shown in Claim 4.2, under such transformation a YES-instance is converted into a YES-instance. However, if we take an arbitrary p then a NO-instance can be mapped to a YES-instance or S_p can become a multiset instead of a set. In order to prevent this, we carefully pick the value of p.

We need to guarantee that if Y is not an AP then Y_p also does not generate an AP in $\mathbb{Z}_p$. Suppose $Y_p = \{y_1, y_2, \ldots, y_k\}$ generates an AP in $\mathbb{Z}_p$ exactly in this order. We assume that x_i maps into y_i, i.e. $x_i \equiv_p y_i$. Since Y_p is an AP in $\mathbb{Z}_p$, we have

$$y_2 - y_1 \equiv_p y_3 - y_2 \equiv_p \cdots \equiv_p y_k - y_{k-1}.$$

Since Y is not an AP there exists an index j such that $x_j - x_{j-1} \neq x_{j+1} - x_j$. Therefore, we have that $2x_j - x_{j-1} - x_{j+1} \neq 0$ and $2y_j - y_{j-1} - y_{j+1} \equiv_p 0$. Since $x_i \equiv_p y_i$ we conclude that p divides $2x_j - x_{j-1} - x_{j+1}$. Hence if we want to choose p that does not transform a NO-instance into a YES-instance, it is enough to choose p such that p is not a divisor of $2x - y - z$ where x, y, z are any numbers from the input and $2x - y - z \neq 0$. Similarly, if we want S_p to be a set instead of a multiset, then for any different x, y the prime p should not be a divisor of $x - y$. Note that the number of different nonzero values of $2x - y - z$ and $x - y$ is at most $O(n^3)$. Since all numbers are bounded by $2^{q(n)}$ the values of $2x - y - z \neq 0$ and $x - y$ are bounded by $2^{q(n)+2}$. Note that any integer N has at most $\log N$ different prime divisors. Therefore at most $O(n^3)(q(n) + 2)$ prime numbers are not suitable for our reduction. In order to find a suitable prime number we do the following:

- for each number from 2 to $n^6(q(n) + 2)^2$ check if it is prime (it can be done in polynomial time [1]),
- for each prime number $p' \leqslant n^6(q(n) + 2)^2$ check if there are integers x, y, z from the input such that ($2x - y - z \neq 0$ and $2x - y - z \equiv_p 0$) or $x \equiv_p y$ if such x, y, z exist go to the next prime number,
- when the desired prime p' is found output instance $S_{p'}$ with modulo p'.

Note that number of primes not exceeding N is at least $\frac{N}{2 \log N}$ for large enough N. Hence by pigeonhole principle we must find the desired p' as we consider all numbers smaller than $n^6(q(n)+2)^2$ and $\frac{n^6(q(n)+2)^2}{\log(n^6(q(n)+2)^2)} > O(n^3)(q(n)+2)$ for sufficiently large n.

Therefore, in polynomial time we can find p' that is polynomially bounded by n and COVER BY ARITHMETIC PROGRESSIONS IN $\mathbb{Z}_p$ with input $(S_{p'}, p')$ is equivalent to CAP with input S (similarly for EXACT COVER BY ARITHMETIC PROGRESSIONS IN $\mathbb{Z}_p$ and XCAP). Hence, COVER BY ARITHMETIC PROGRESSIONS IN $\mathbb{Z}_p$ and EXACT COVER BY ARITHMETIC PROGRESSIONS IN $\mathbb{Z}_p$ are strongly NP-complete.

■

4.6 Parameterization Below Guarantee

In this section we present an FPT algorithm parameterized below guarantee for a problem that generalizes CAP, namely t-Uniform Set Cover. The t-Uniform Set Cover is a special case of the Set Cover problem in which all instances $\mathcal{S}, U$ satisfy the property that $\{A \subseteq U : |A| = t\} \subseteq \mathcal{S}$. Clearly the solution for the t-Uniform Set Cover problem is at most $\lceil \frac{n}{t} \rceil$ where n is the size of universe. Note that CAP is a special case of 2-Uniform Set Cover since any pair of element forms an AP. Thus we focus on presenting a fixed-parameter tractable algorithm for the t-Uniform Set Cover problem parameterized below $\lceil \frac{n}{t} \rceil$ (we consider t to be a fixed constant). We note that for a special case with $t = 1$ the problem was considered in works [6,43].

We will use the deterministic version of color-coding, which uses the following standard tools:

Definition 4.10. *For integers n, k a (n,k)-*perfect hash family *is a family $\mathcal{F}$ of functions from $[n]$ to $[k]$ such that for each set $S \subseteq [n]$ of size k there exists a function $f \in \mathcal{F}$ such that $f(S) = [k]$.*

Lemma 4.11 ([99]). *For any $n, k \geqslant 1$, one can construct an (n,k)-perfect hash family of size $e^k k^{O(\log k)} \log n$ in time $e^k k^{O(\log k)} n \log n$.*

Now we are ready to state and prove the main result of this section:

Theorem 4.12. *There is an $2^{O(k)} n^{O(1)}$-time algorithm that for constant t and a given instance of t-Uniform Set Cover determines the existence of a set cover of size at most $\lceil \frac{n}{t} \rceil - k$ where k is an integer parameter and n is the size of the universe.*

Proof. In the first stage of our algorithm we start picking sets greedily (i.e. in each step, we pick the set that covers the largest number of previously uncovered elements) until there are no sets that cover at least $t+1$ previously uncovered elements. If during this stage we pick s sets and cover at least $st + tk$ elements then our instance is a Yes-instance. Indeed, we can cover the remaining elements using $\lceil \frac{n-st-tk}{t} \rceil$ sets. In total, such a covering has at most $\lceil \frac{n-st-tk}{t} \rceil + s = \lceil \frac{n}{t} \rceil - k$ subsets. Intuitively, each set picked during this greedy stage covers at least one additional element. Therefore, if we pick more than tk subsets then our input instance is a Yes-instance. This means that after the greedy stage we either immediately conclude that our input is a Yes-instance or we have used at most tk subsets and covered at most $tk \cdot t + tk = O(k)$ elements, for a fixed t. Let us denote the subset of all covered elements by G. Note that there is no subset that covers more than t elements from $U \setminus G$.

If there is a covering of size $\lceil \frac{n}{t} \rceil - k$ then there are $s' \leqslant |G| \leqslant tk \cdot t + tk$ subsets that cover G and at least $s't + tk - |G|$ elements of $U \setminus G$. Moreover, if such subsets exist then our input is a Yes-instance.

Hence, it is enough to find s' such subsets. For each $s'' \in [|G|]$ we attempt to find subsets $S_1, S_2, \ldots, S_{s''}$ such that $G \subset S_1 \cup S_2 \cup \cdots \cup S_{s''}$ and $S_1 \cup S_2 \cup \cdots \cup S_{s''}$ contains at least $s''t + tk - |G|$ elements from $U \setminus G$. Assume that for some s'' such sets exist. Let H be an arbitrary subset of $(S_1 \cup S_2 \cup \cdots \cup S_{s''}) \setminus G$ of size $s''t + tk - |G|$. Note that we do not know the set H. However, we employ the color-coding technique, and

construct a $(n, |H|)$-perfect hash family $\mathcal{F}$. We iterate over all $f \in \mathcal{F}$. Now using dynamic programming we can find H in time $2^{O(k)}$ as follows.

We consider a new universe U' which contains elements from the set G and elements corresponding to $|H|$ colors corresponding to values assigned by f to $U \setminus G$. Moreover, if a subset P was a subset that can be used for covering in t-Uniform Set Cover then we replace it with $(P \cap G) \cup \{a : a \in f(P \cap (U \setminus G))$ for some $f \in \mathcal{F}\}$. We replace our t-Uniform Set Cover instance with an instance of Set Cover with a universe of size $|G|+|H| \leqslant |G|+s''t+tk-|G| \leqslant |G| \cdot t+tk \leqslant (tk \cdot t+tk) \cdot t+tk = O(k)$. It is easy to see that our original instance is a Yes-instance if and only if the constructed instance of Set Cover admits a covering by at most s'' sets (under the assumption that f indeed assigns distinct values to elements of H). If f does not assign distinct values then a Yes-instance can be become a No-instance. However, a No-instance cannot become a Yes-instance. Since $|H| = O(k)$, the overall running time is $2^{O(k)}n^{O(1)}$. ■

As a corollary of the previous theorem we get the following result.

Theorem 4.4. *There is an $2^{O(k)}n^{O(1)}$ time algorithm that detects if a given set X of integers can be covered with at most $|X|/2 - k$ APs.*

Proof. The result immediately follows from Theorem 4.12 with $t = 2$. ■

4.7 Conclusion

We presented FPT algorithms for Cover by Arithmetic Progressions and Exact Cover by Arithmetic Progressions. These are one of rare parameterized algorithms that have been used for solving problems for number theory, and we hope our results inspire further research in this direction.

A natural direction for further research would be to obtain faster parameterized algorithms for CAP and XCAP. Our algorithm for XCAP is slower than the one for CAP, which coincides with the intuition that XCAP is a harder problem than CAP. However, it is unclear whether this difference in complexity is just a property of the techniques we used, or is it inherent to the problems. Another question that remains open is the strong NP-hardness of CAP and XCAP.

Chapter 5

XNLP-hardness of Parameterized Problems on Planar Graphs

5.1 Introduction

In classical complexity theory, we can classify problems depending on their space complexity or their time complexity. These two classifications are intertwined as follows: $L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXP$. In particular, the class NL (Nondeterministic Logarithmic-space) is contained in the class P (Polynomial-time). It is natural to ask whether there is a similar chain of inclusions for the parameterized analogues of these classes. In particular, what can we say about the relationship between FPT (analogue of P) and XNL (analogue of NL)?

This question was first posed by Elberfeld et al. [55]. They defined the class $N[fpoly, flog]$ (here called XNLP) as the class of (parameterized) problems that can be solved with a nondeterministic Turing machine in $f(k)n^{O(1)}$ time and $g(k)\log n$ space and gave the first problems complete for this class. They also present several problems that are complete for this class, stating: *"The real challenge lies in finding problems together with* natural *parameterizations that are complete"*.

In the last decade, this question has been resolved: many standard graph problems and their generalizations were shown to be XNLP-complete (see Table 5.1 for examples of such problems). This class turned out to be a natural home for several problems that are $W[t]$-hard for all t but unlikely to belong to $W[P]$, such as Bandwidth and List Coloring (parameterized by pathwidth) [23].

Recently, in [22] a "tree variant" of the class XNLP was introduced. This class is called XALP, and many problems that are XNLP-complete parameterized by pathwidth are shown to be XALP-complete parameterized by treewidth (see Table 5.1). The class XALP contains problems that can be solved by a nondeterministic Turing machine with access to an auxiliary stack in $f(k)n^{O(1)}$ time and $g(k)\log n$ space.

A natural question to ask is whether we can say something about hardness for

other graph parameters. Several XNLP-hard problems for treewidth are known to be easy (i.e. in FPT) parameterized by the so-called stable gonality parameter [18]. In this chapter, we study the hardness of problems on planar graphs, parameterized by outerplanarity, treewidth and pathwidth.

Outerplanarity is a natural parameter to study for problems on planar graphs. As graphs of outerplanarity k have treewidth at most $3k-1$ (see e.g. [17]), algorithmic results for graphs of bounded treewidth carry over to planar graphs of bounded outerplanarity. This has also been exploited in the well-known and often applied layering technique of Baker [5], resulting in approximation schemes for many problems on planar graphs.

There are several celebrated results in the field of parameterized complexity where the complexity of a problem significantly decreases when we move from general graphs to planar graphs, for instance DOMINATING SET parameterized by solution size then drops from being W[2]-complete [53] to a fixed-parameter tractable problem with a linear kernel [2]. The results of this chapter show that for several problems that are in XP for treewidth as parameter, we have no complexity drop when we move to the realm of planar graphs.

Our results. An overview of the results of this chapter can be seen in Table 5.1. For all the membership results, we assume that we are given the tree decomposition or path decomposition of the input graph. In addition to these results, we also show the XNLP-completeness of BINARY CSP for $k \times n$-grids (parameterized by k).

Organization. In Section 5.2 we introduce some definitions and notations. In Section 5.3 we strengthen two previous results concerning BINARY CSP and two related problems. Section 5.4 discusses the SCATTERED SET problem, whose hardness can be shown by a reduction from BINARY CSP. In Section 5.5, we consider the complexity of the ALL-OR-NOTHING FLOW problem, and prove it to be XNLP-hard with outerplanarity as parameter by a reduction from BINARY CSP. In Section 5.6 we describe several problems whose hardness can be shown by a reduction from ALL-OF-NOTHING FLOW. We finish with some concluding remarks in Section 5.7.

5.2 Definitions and Notation

5.2.1 Graph notions

Throughout this chapter, graphs are undirected unless specified otherwise. A planar graph is outerplanar (or 1-outerplanar) if it has na embedding such that all its vertices are on the outer face. Informally speaking, a planar graph G is k-outerplanar if its vertices are on k "layers". Formally, k-outerplanar graphs are defined as follows.

Definition 5.1. *An embedding of a planar graph G outerplanar if all its vertices are on the outer face. An embedding of a planar graph G is k-outerplanar if, after deleting the vertices on the outer face, the remaining embedding is $(k-1)$-outerplanar. A planar graph is* k-outerplanar *if it has a k-outerplanar embedding.*

	outerplanarity	treewidth	pathwidth
Binary CSP	XALP-c. (5.3.2)	XALP-c. [22]	XNLP-c. [23]
List Coloring	XALP-c. (5.3.2)	XALP-c. [22]	XNLP-c. [23]
Precoloring Extension	XALP-c. (5.3.2)	XALP-c. [22]	XNLP-c. [23]
Scattered Set	XALP-c. (5.4)	XALP-c. (5.4)	XNLP-c. (5.4)
All-or-Nothing Flow	XNLP-h. (5.5)	XALP-c. [22]	XNLP-c. [18]
All-or-Nothing Flow with cap. $\leqslant 2$	XNLP-h. (5.6.1)	XALP-c. (5.6.1)	XNLP-c. (5.6.1)
Target Outdegree Orientation	XNLP-h. (5.6.2)	XALP-c. [22]	XNLP-c. [18]
Capacitated (Red-Blue) Dominating Set	XNLP-h. (5.6.3)	XALP-c. [22]	XNLP-c. [21]
Capacitated Vertex Cover	XNLP-h. (5.6.4)	XALP-c. [22]	XNLP-c. [21]
f-Dominating Set	XNLP-h. (5.6.5)	XALP-c. (5.6.5)	XNLP-c. (5.6.5)
k-Dominating Set	XNLP-h. (5.6.5)	XALP-c. (5.6.5)	XNLP-c. (5.6.5)
Target Set Selection	XNLP-h. (5.6.6)	XALP-c. (5.6.6)	XNLP-c. (5.6.6)

Table 5.1: An overview of results of this chapter and results from the literature. Results of this chapter are stated with the corresponding section numbers.

In other words, a graph is k-outerplanar if after k operations consisting of removing all the vertices on the outer face, we obtain an empty graph. The outerplanarity of a graph can be computed in polynomial time [77].

Kloks [81] introduced the notion of nice tree and path decompositions. Throughout this chapter, we assume that we are given a (nice) tree decomposition and a (nice) path decomposition of the input graph. We will use a variant of nice path decompositions, which we will describe with a sequence of operations on terminal graphs. Here, a terminal graph is a triple (V, E, X), together with a binary relation $\prec$ on X which is a strict total ordering. We call elements of X terminals. We consider the following operations on terminal graphs:

- **Introduce**. Given a terminal graph $G = (V, E, X)$, the introduce operation adds a new isolated vertex v to V and to X, with v the smallest vertex regarding the ordering in X: $G' = (V \cup \{v\}, E, \{v\} \cup X)$, with $v \prec x$ for all $x \in X$.

- **Forget**. Given a terminal graph $G = (V, E, X)$ with $X \neq \emptyset$, the largest element of X is no longer a terminal: $G' = (V, E, X \setminus \{x\})$, with $y \prec x$ for all $y \in X \setminus \{x\}$.

- **Add-Edge**(i). Given a terminal graph $G = (V, E, X)$ with $|X| > i$, we add an edge between the ith and the $(i+1)$st terminal: V and X are unchanged, and one edge is added to E.

- **Swap**(i). Given a terminal graph $G = (V, E, X)$ with $|X| > i$, swap in the ordering in X the ith and the $(i+1)$st vertex.

To build graphs of small treewidth, we need one more operation, that now has two terminal graphs as input.

- **Join**. Given two terminal graphs $G_1 = (V_1, E_1, X)$ and $G_2 = (V_2, E_2, X)$ that only intersect in their terminal nodes, i.e., $V_1 \cap V_2 = X$ and $E_1 \cap E_2 = \emptyset$, we build the graph $G = (V_1 \cup V_2, E_1 \cup E_2, X)$. The operation 'fuses' the two terminal graphs, identifying for each i, the ith terminal node of the first graph with the ith terminal node of the second graph.

Note that with a sequence of Swap operations, we can order the terminals in any manner; one can thus easily observe that a nice path decomposition of a graph $G = (V, E)$ of pathwidth at most k (see [81]) can be transformed into a sequence of Introduce, Forget, Add-Edge and Swap operations that creates the terminal graph $G = (V, E, \emptyset)$ with $O(kn)$ operations, and with each intermediate terminal graph having at most $k+1$ terminals. Similarly, a nice tree decomposition can be transformed to a similar sequence of Introduce, Forget, Add-Edge, Swap and Join operations. From this sequence of operations, we can go back to a *generalized tree decomposition*, i.e., we have a rooted tree, with each bag consisting of the set of terminals, and of one of the following types: Introduce, Forget, Add-Edge, Swap and Join. We omit the simple details; one can also carry out the above transformations in logarithmic space.

5.2.2 The classes XNLP and XALP

A *parameterized problem* is a language $L \subseteq \Sigma^* \times \mathbb{N}$ for some finite alphabet Σ. The class XNLP consists of parameterized problems that can be solved by a nondeterministic algorithm which uses $f(k)n^{O(1)}$ time and $g(k) \log n$ space for some computable function f (where k is the parameter and n is the size of the input instance).

As in the classical complexity setting, in order to define XNLP-hardness and XNLP-completeness, we need the notion of *reduction*.

Definition 5.2. *A* parameterized reduction *from a problem $L_1 \subseteq \Sigma_1^* \times \mathbb{N}$ to a problem $L_2 \subseteq \Sigma_2^* \times \mathbb{N}$ is a function $f : \Sigma_1^* \times \mathbb{N} \to \Sigma_2^* \times \mathbb{N}$ such that the following two conditions are satisfied:*

- *For all $(x, k) \in \Sigma_1^* \times \mathbb{N}$, $(x, k) \in L_1$ if and only if $f((x, k)) \in L_2$,*

- *There is a computable function $g : \mathbb{N} \to \mathbb{N}$ such that for all $(x, k) \in \Sigma_1^* \times \mathbb{N}$ we have $k' \leqslant g(k)$, where $f((x, k)) = (y, k')$.*

We call f a parameterized logspace reduction *or* pl-reduction *if there is an algorithm that computes $f((x,k))$ in space $O(g'(k)+\log|x|)$, where $g':\mathbb{N}\to\mathbb{N}$ is a computable function and $|x|$ the size of x (i.e. number of bits needed to store x).*

We call a problem XNLP-hard if any other problem in XNLP can be pl-reduced to it, and XNLP-complete if it is XNLP-hard and in XNLP.

We define XALP as the class of problems that can be solved by a non-deterministic Turing machine with a stack in $f(k)n^{O(1)}$ time and $g(k)\log n$ space. For other equivalent definitions of XALP we refer the reader to [22].

Membership in XNLP (respectively XALP) can usually be shown by standard dynamic programming techniques on path decompositions (respectively tree decompositions). Intuitively, the log space complexity can be achieved by guessing the entry (rather than storing all the entries). Examples of membership proofs can be found in [21–23].

In our hardness proofs, we skip the details that show the logarithmic space bounds for the pl-reductions. The standard technique to realize such space is to recompute needed values instead of storing them. For example, if the reduction uses a Sidon set, we do not store the Sidon set, but each time we need the ith element of the set, we recompute it. This increases the time of the reduction by a polynomial factor, but keeps the used space small.

5.3 Binary CSP

The Constraint Satisfaction Problems (CSP) form a well-known class of problems with a wide set of applications, ranging from artificial intelligence to operations research. In this section, we study Binary CSP, a CSP where all constraints are binary. Formally, the problem is defined as follows.

Binary CSP
Input: A graph $G=(V,E)$, a set $\mathcal{C}$, set $C(v)\subseteq\mathcal{C}$ for each $v\in V$, set $C(u,v)\subseteq\mathcal{C}\times\mathcal{C}$ for each ordered pair $(u,v)\in V\times V$ such that $\{u,v\}\in E$
Task: Is there a function $f:V\to\mathcal{C}$ such that for every $v\in V$, $f(v)\in C(v)$ and for every $uv\in E$, we have $(f(u),f(v))\in C(u,v)$?

We call the elements of $\mathcal{C}$ colors, the sets $C(v)$ domains (or vertex constraints) and the sets $C(u,v)$ edge constraints. We emphasize that $C(u,v)$ contains ordered pairs, i.e. even though the graph is undirected, the ordering of vertices matters for the edge constraints.

List Coloring and Precoloring Extension are special cases of Binary CSP, defined as follows.

List Coloring
Input: A graph $G=(V,E)$, a set $\mathcal{C}$, set $C(v)\subseteq\mathcal{C}$ for each $v\in V$
Task: Is there a function $f:V\to\mathcal{C}$ such that for every $v\in V$, $f(v)\in C(v)$ and for every $uv\in E$, $f(u)\neq f(v)$?

Precoloring Extension
Input: A graph $G = (V, E)$, a set $\mathcal{C}$, a subset $W \subseteq V$, a function $f' : W \to \mathcal{C}$
Task: Is there a function $f : V \to \mathcal{C}$ such that for every $v \in W$, $f(v) = f'(v)$ and for every $uv \in E$, $f(u) \neq f(v)$?

Note that an instance of List Coloring can be seen as an instance of Binary CSP where for each edge uv the edge constraint is $C(u, v) = \{(a, b) : a \neq b\}$. An instance of Precoloring Extension can be seen as an instance of Binary CSP where each vertex has a domain of size 1 or $|\mathcal{C}|$. Note that we may assume without loss of generality that $C(u, v) \subseteq C(u) \times C(v)$.

Binary CSP, List Coloring and Precoloring Extension were shown to be XNLP-complete parameterized by pathwidth [23]. In the first part of this section, we will show that Binary CSP is also XNLP-complete for a more restrictive graph class, namely $k \times n$-grids (parameterized by k). As vertices whose lists are larger than their degree plus one can always be colored, and List Coloring is FPT for graphs of bounded treewidth plus maximum list size [76], List Coloring and Precoloring Extension are FPT for $k \times n$-grid, parameterized by k.

Binary CSP, List Coloring and Precoloring Extension were shown to be XALP-complete parameterized by treewidth and by treewidth plus degree [22]. In the second part of this section, we will show for each of these three problems XALP-completeness for a more restrictive graph class, namely k-outerplanar graphs (parameterized by k). We remark that List Coloring is in L for trees [20].

5.3.1 XNLP-completeness for $k \times n$-grids

In order to prove the XNLP-completeness for $k \times n$-grids, we will use an equivalent definition of pathwidth (see Lemma 2.7 or standard textbooks), namely that a graph G has pathwidth k if and only if it is a subgraph of an interval graph whose clique number is $k + 1$.

Note that, if we are given a path decomposition we can easily construct the corresponding interval graph H (for details see [17]). Now we are ready to prove the main theorem of this section.

Theorem 5.3. Binary CSP *is XNLP-complete for $k \times n$-grids parameterized by k.*

Proof. We will reduce from Binary CSP parameterized by pathwidth. Let G be a graph of pathwidth $k-1$. By Lemma 2.7, there is an interval graph H' such that G is a subgraph of H' and $\omega(H') = k$. Let $\mathcal{I}$ be a collection of intervals whose intersection graph is H'. Intuitively, we will draw the intervals in $\mathcal{I}$ on a grid. Formally, we construct an instance of Binary CSP with underlying graph H as follows. We start with H being a $k \times n$-grid. To each $v \in V(G)$, we assign a horizontal path I_v and a vertex ℓ_v in H such that the following conditions are met:

- ℓ_v is the left endpoint of I_v;
- $I_v \cap I_u = \emptyset$ for all $u \neq v$;
- ℓ_u and ℓ_v are in different columns for all $u \neq v$;

- For all edges $uv \in E(G)$, there is a column in H that contains a vertex in I_v and a vertex in I_u.

We will now make the grid H twice finer, i.e. we will add a row between every two adjacent rows and a column between every two adjacent columns. We will call the newly added rows, columns and vertices green, and the original vertices black. Each green vertex whose left and right neighbours are in I_v for some $v \in V(G)$ is added to I_v.

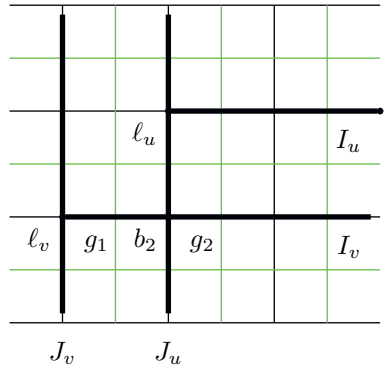


Figure 5.1: A part of the graph H showing an intersection of intervals corresponding to u and v

For each vertex v, let J_v be the column that contains ℓ_v. Consider the green vertices on the paths I_v and J_v for some vertex $v \in V(G)$. We will add constraints to these vertices and to edges between adjacent vertices in I_v and J_v to ensure that all of them get the same color as ℓ_v. Denote the vertices on I_v from left to right by $b_1, g_1, b_2, g_2, \ldots, b_{t-1}, g_{t-1}, b_t$, where $b_1 = \ell_v$, the vertices g_i are green, and b_i are black (see Figure 5.1). For g_1, we set $C(g_1) = C(v)$ and $C(\ell_v, g_1) = \{(c, c) : c \in C(v)\}$. For $i = 2, ..., t$, we define the domains of g_i and b_i, as well as the constraints of the adjacent edges as follows. We define $C(g_i) = C(v)$. If b_i is in column J_u for some $u \in V(G)$, we define $C(b_i) = \{c_{xy} : (x, y) \in C(v, u)\}$ (if $uv \notin E(G)$, we define $C(u, v) = C(u) \times C(v)$). We define $C(g_{i-1}, b_i) = \{(x, c_{xy}) : xy \in C(v, u)\}$ and $C(b_i, g_i) = \{(c_{xy}, x) : xy \in C(v, u)\}$. If b_i is not in column J_u for any $u \in V(G)$, we define $C(b_i) = \{c_{xx} : x \in C(v)\}$, $C(g_{i-1}, b_i) = \{(x, c_{xx}) : x \in C(v)\}$ and $C(b_i, g_i) = \{(c_{xx}, x) : x \in C(v)\}$. Note that now all of the green vertices on I_v have the same color as ℓ_v. We define the domains of vertices on J_v and the constraints between adjacent vertices analogously. For all the other vertices v and edges uv of H, we define $C(v) = \mathcal{C}$ and $C(u, v) = \mathcal{C} \times \mathcal{C}$.

Given a valid coloring of H, we can color each vertex $v \in V(G)$ with the color of ℓ_v. For every edge $uv \in E(G)$, we have that I_v and J_u intersect at a black vertex or that I_u and J_v intersect at a black vertex. By our choice of domain for the black vertex in the intersection, we ensured that the constraint of the edge uv is satisfied. Thus we get a valid coloring of G. It is easy to see that given a valid coloring of G, we can construct a valid coloring of H. ■

5.3.2 XALP-completeness parameterized by outerplanarity

In this section, we show the XALP-completeness of BINARY CSP parameterized by outerplanarity.

Theorem 5.4. BINARY CSP *is XALP-complete parameterized by outerplanarity.*

Proof. We will reduce from BINARY CSP parameterized by treewidth. Given a graph G with a nice tree decomposition of treewidth k, we will construct a k-outerplanar (plane) graph H. We first transform the nice tree decomposition to a generalized one, i.e., each bag is of one of the following types: Introduce, Forget, Add-Edge, Swap or Join.

We denote by $\prec_X$ the ordering of terminals corresponding to the bag X. For each bag X and each terminal v in X, we add to H a vertex v_X. We draw the vertices of H in the plane as follows. For each bag X, the terminals in X are on the same horizontal line, ordered from left to right according to $\prec_X$. For each bag X and its child Y, the vertices in H corresponding to X are above vertices corresponding to Y.

Denote by B_X the set $B_X = \{v_X : v \in X\} \subseteq V(H)$. For each node X of the tree decomposition of G, we add the following edges to H. For each $v \in V(G)$, if both X and its child Y contain v, we add the edges $v_X v_Y$. In addition, if X is an Add-Edge node corresponding to edge vw, we add the edge $v_X w_X$ to H (see Figure 5.2).

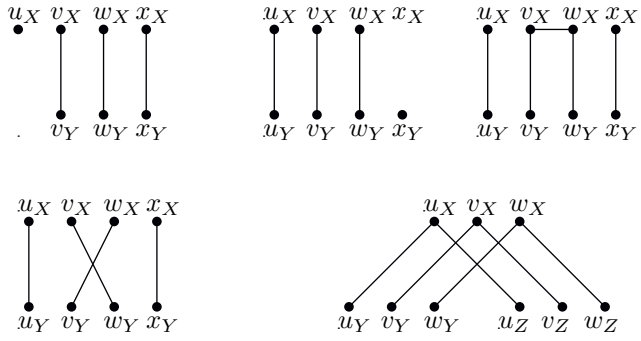


Figure 5.2: Representation of nodes: Introduce u, Forget x, Add-Edge vw, Swap vw, Join.

For each bag B_X and each vertex $v_X \in B_X$, set $C(v_X) = C(v)$. For each edge $v_X v_Y$ in H, set $C(v_X, v_Y) = \{(d, d) : d \in C(v)\}$. For each edge $v_X w_X$ in H set $C(v_X, w_X) = C(v, w)$.

Note that the graph H is not planar yet. We will now turn it into a planar graph by using crossover gadgets. Consider edges ab and cd that intersect at a point. Add a vertex m at the intersection point and subdivide the edges mb and md by adding vertices a' and c' respectively. Set $C(a') = C(a)$, $C(c') = C(c)$, $C(m) = \{m_{ij} : i \in C(a), j \in C(c)\}$, $C(m, a) = C(m, a') = \{(m_{ij}, i) : i \in C(a), j \in C(c)\}$, $C(m, c) = C(m, c') = \{(m_{ij}, j) : j \in C(c), i \in C(a)\}$, $C(a', b) = C(a, b)$, $C(c', d) = C(c, d)$. This way, we ensure that a' (respectively c') has the same color as a (respectively c).

In any valid coloring of H, all vertices that correspond to the same vertex in G must have the same color. Also, for each edge in G, we have an Add-Edge node

corresponding to it, and in that node we check whether the edge consraint is satisfied. Therefore, any valid coloring of H defines a valid coloring of G and vice versa. It remains to show that H has bounded outerplanarity. For each bag X, the vertices v_X are on the outer face after at most $k/2$ rounds. The subgraphs corresponding to join nodes are subgraphs of a $k \times n$-grid (with some subdivided edges) and thus $k/2$-outerplanar. The vertices corresponding to the intersection of edges in the Swap nodes will be on the other face in at most $k/2+1$ rounds. Therefore, the graph H is $(k/2+1)$-outerplanar. ■

Corollary 5.1. List Coloring *and* Precoloring Extension *are XALP-complete parameterized by outerplanarity.*

Proof. We observe that the existing transformations increase the outerplanarity by at most 1, see e.g. [23]. An instance of Binary CSP can be transformed to an equivalent instance of List Coloring by renaming some colors, and replacing edges uv by a number of vertices of degree 2, each adjacent to u and to v. From List Coloring we go to Precoloring Extension by adding to each vertex v precolored neighbors of degree 1; one such vertex for each color in $\mathcal{C} \setminus C(v)$. ■

5.4 Scattered Set

In this section, we will consider the Scattered Set problem, which is a generalization of the Independent Set problem and is defined as follows.

Scattered set
Input: $G = (V, E)$, $k \in \mathbb{N}$, $d \in \mathbb{N}$
Task: Is there a set $I \subseteq V$ of size k such that for any pair of distinct vertices $u, v \in I$ the distance between u and v is at least d?

For a fixed $d \geqslant 3$, the problem with parameter k is W[1]-hard, even for bipartite graphs [58]. The case when $d = 2$ gives the Independent Set problem which is W[1]-complete [53]. For fixed d, the problem is FPT when parameterized by treewidth, by standard dynamic programming techniques (see [80]). In this section, we consider the case that d is part of the input.

Theorem 5.5. Scattered Set *is XALP-complete parameterized by treewidth and by outerplanarity and it is XNLP-complete parameterized by pathwidth.*

Proof. Membership in XNLP and XALP, respectively, follows with standard arguments, cf. the discussion in Section 5.2. A state of the machine will contain as information the current bag in the decomposition, and for each vertex in this bag its distance to an element in the scattered set (i.e. $k+1$ integers in $[0, d]$).

In both cases, we will reduce from Binary CSP. Given an instance of Binary CSP with underlying graph G and d colors, we will construct an instance H of Scattered Set as follows. The distance between solution vertices d equals the number of colors in $\mathcal{C}$.

Firstly, for every vertex v of G construct a cycle C_v of length equal to the degree of v. For every edge $\{u, v\} \in E(G)$, construct an edge in H between a vertex in C_u and

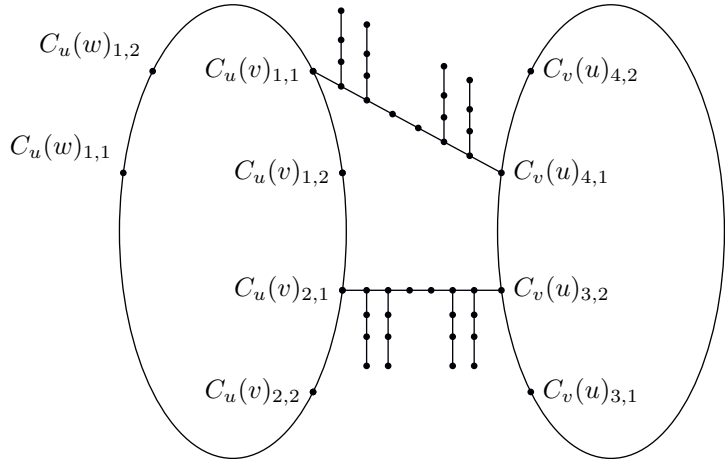


Figure 5.3: Cycles C_u and C_v and representation of the forbidden pairs $(1,1)$ and $(1,2)$ corresponding to the edge uv.

C_v such that the order of edges around the cycle C_v is the same as the order of edges around v and each vertex of C_v has degree 3. Note that the obtained graph H remains planar. For an edge $\{u,v\} \in E(G)$, let $C_v(u) \in C_v$ and $C_u(v) \in C_u$ be its endpoints. For each edge $\{u,v\} \in E(G)$, replace $C_v(u)$ and $C_u(v)$ with paths $C_v(u)_{1,1}...C_v(u)_{d^2,d}$ and $C_u(v)_{1,1}...C_u(v)_{d^2,d}$ such that the following are true (see Figure 5.3):

- $C_v(u)_{1,1}$ and $C_v(u)_{d^2,d}$ ($C_u(v)_{1,1}$ and $C_u(v)_{d^2,d}$ respectively) are adjacent to exactly one neighbour of $C_v(u)$ in C_v (neighbour of $C_u(v)$ in C_u respectively);
- The vertices $C_v(u)_{i,j}$ ($C_u(v)_{i,j}$ respectively) are lexicographically ordered clockwise around the cycle.

For each edge $\{u,v\} \in E(G)$, each $i \in [d]$ and each color $j \notin C(u)$, add a path of length $d-1$ to $C_u(v)_{i,j}$. For each edge $\{u,v\} \in E(G)$, consider the forbidden pairs of colors, i.e. the pairs that do not belong to $C(u,v)$. Intuitively, for each such pair (i,j), we select a vertex in $C_u(v)$ which corresponds to color i and a vertex in $C_v(u)$ which corresponds to color j. We connect the selected vertices by a gadget which will ensure that we cannot have both of the selected vertices in S.

Formally, let $(i,j) \notin C(u,v)$ and let $a = (i-1) \cdot d + j$. We construct a path $P(u,v,i,j)$ from $C_u(v)_{a,i}$ to $C_v(u)_{d^2+1-a,j}$ of length $2d+1$. Note that the graph remains planar: we have d^2 "blocks" in $C_u(v)$ (i.e. paths on the cycle which contain one vertex for each color) on both sides, and the ath block has an outgoing edge if and only if the ath pair (lexicographically ordered) (i,j) does not belong to $C(u,v)$. To each vertex of $P(u,v,i,j)$ except the middle two, add paths of length $d-1$.

Consider a d-scattered set S in H. We claim that it corresponds to a solution to the given Binary CSP instance. For any vertex v and a color $i \notin C(v)$, without loss of generality we may assume that none of the vertices $C_v(u)_{a,i}$ belong to S. Also, note that we cannot have $C_v(u)_{a,i} \in S$ and $C_v(u')_{b,j} \in S$ for $i \neq j$. Therefore, there is exactly one color i such that $C_v(u)_{a,i} \in S$ for some u, and this will be the

color that we will assign to v. It remains to check whether the edge constraints are satisfied. Consider an edge $uv \in E(G)$ and its constraints $C(u,v)$. Suppose that in the above process we selected a color i for u and j for v, such that $(i,j) \notin C(u,v)$. Then there exist $a, b \in [d^2]$ such that $C_u(v)_{a,i}, C_v(u)_{b,j} \in S$ and $C_u(v)_{a,i}$ and $C_v(u)_{b,j}$ are connected by a path. Note that on that path no vertex belongs to S, which leads to a contradiction.

It remains to bound the outerplanarity, treewidth and pathwidth of H. Let G be k-outerplanar and let $p(v)$ denote the layer of a vertex v, i.e. the number of times the outer face of G needs to be removed in order to have v on the outer face. Note that any vertex of C_v of degree 2 belongs to the same layer as v. Therefore, in round $p(v)+1$ all vertices of C_v are on the outer face. In round $p(v)+2$, all vertices of $P(u,v,i,j)$ are on the outer face. Therefore, the graph H is $k+2$-outerplanar. Note that k-outerplanar graphs have treewidth at most $3k-1$ [17].

Let us now compute the pathwidth of H. Consider a path decomposition of G of width k'. Given a bag $P'_i \subseteq V(G)$, construct $P_i \subseteq V(H)$ as follows. For every $v \in P'_i$, add to P_i all the vertices on C_v, and for every edge $\{u,v\} \in E(G)$ such that $u, v \in P'_i$, add all the vertices on paths between C_u and C_v, as well as all the vertices on paths with an endpoint at C_u or C_v. It is easy to see that the sets P_i form a path decomposition of H, and since d is a constant, the resulting path decomposition has width $O(k')$. ■

5.5 All-or-Nothing Flow

The All-or-Nothing Flow problem asks if there is an all-or-nothing flow (i.e. a flow where every edge has either zero flow or flow equal to its capacity) with a given value. Formally, the problem is defined as follows (for the definition of flow, see Section 2.2).

All-or-Nothing Flow
Input: A flow network (G, cap, s, t), and an integer r.
Task: Is there an all-or-nothing flow from s to t with value exactly r in G?

In this section, we show that All-or-Nothing Flow is XNLP-hard with the outerplanarity plus the pathwidth of the graph as parameter. The problem is known to be NP-complete [3], XNLP-complete with pathwidth as parameter [18] and XALP-complete with treewidth as parameter [22]. The problem is a good starting point for further hardness proofs, and a natural generalisation of network flow.

We remark that the variant where we ask if there is a flow whose value is at least r is equally hard, since we can add a new source s' and an arc from s' to the old source s with capacity r.

Theorem 5.6. All-or-Nothing Flow *is XNLP-hard for planar graphs with outerplanarity as parameter.*

Proof. We reduce from Binary CSP with pathwidth as parameter, and employ the technique of modelling the choice of a colour for a vertex by choosing a number from a Sidon set (or Golomb ruler), see e.g. [57].

A *Sidon set* is a set of positive integers $\{a_1, \ldots, a_n\}$ with the property that each different pair of integers from the set has a different sum: $a_{i_1} + a_{i_2} = a_{j_1} + a_{j_2}$ implies $\{i_1, i_2\} = \{j_1, j_2\}$. Sidon sets are also known as Golomb rulers. Erdős and Turán [57] have shown that for each n, there is a Sidon set of size n with all numbers in the set at most $4n^2$; one can also observe from their proof that this set can be constructed with logarithmic space.

Now, suppose we are given an instance of Binary CSP with pathwidth as parameter. That is, we have a graph $G = (V, E)$, a set D of colours, for each vertex $v \in V$, a subset $C(v) \subseteq D$, and for each edge $\{v, w\} \in E$, for the ordered pair vw, a set of allowed colour pairs $C(v, w) \subseteq C(v) \times C(w)$. Also, we are given a path decomposition of G of width at most k.

Our aim is to construct an instance of All-or-Nothing Flow with underlying graph H such that it has a solution if and only if the given Binary CSP instance has a solution.

First, we modify the instance as follows. By a simple modification of the colour sets, we can assume that the sets of allowed colours for vertices are disjoint, i.e., for $v \neq w$, $C(v) \cap C(w) = \emptyset$. Second, we change the path decomposition to a series of Introduce, Forget, Swap, and Add-Edge operations that give $G = (V, E, \emptyset)$ when starting with the empty graph.

Without loss of generality, we may assume that $D = \bigcup_{v \in V} C(v)$. Take a Sidon set with $|D|$ positive integers in $O(|D|^2)$. Suppose these are $a_1, \ldots, a_{|D|}$. Let $L = \max\{a_1, \ldots, a_{|D|}\} + 1$. Note that the set $\{a_1 + L, \ldots, a_{|D|} + L\}$ is also a Sidon set. Now, we assign to each colour c in D a unique element $s(c)$ from $\{a_1 + L, \ldots, a_{|D|} + L\}$.

For a vertex $v \in V(G)$, we write $S(v) = \{s(c) : c \in C(v)\}$: the set of Sidon numbers of the colours we can give to v. For a pair of vertices $v, w \in V(G)$, we write $S(v) + S(w) = \{s(c) + s(c') : c \in C(v) \wedge c' \in C(w)\}$. For an ordered pair of endpoints of an edge $\{v, w\} \in E$, we write $S(v, w) = \{s(c) + s(c') : c \in C(v) \wedge c' \in C(w) \wedge (c, c') \in C(v, w)\}$.

Gadgets. Before describing the construction of H, we introduce three auxiliary gadgets, *XOR*, *Edge* and *Swap*. In order to simplify the drawing, instead of parallel arcs, we draw one arc whose capacity is denoted by the (multi)set of capacities of the parallel arcs. We denote the multiset consisting of t ones by $\mathbf{1}_t$.

Informally, given a vertex $v \in V(G)$, the $XOR(v)$ gadget will "select" exactly one value $s(c) \in S(v)$, which corresponds to coloring v with color c. Formally, given a vertex $v \in V(G)$, $XOR(v)$ is a graph that consists of three nodes, v_1, v_2, v_3 and has $2L$ edges of capacity 1 from v_1 to v_2 and for each element of $S(v)$ an arc from v_2 to v_3 with that capacity (see Figure 5.4). Note that, since all elements of $S(v)$ are between L and $2L - 1$ and the incoming flow to v_2 is at most $2L$, we cannot have more than one arc from v_2 to v_3 with nonzero flow.

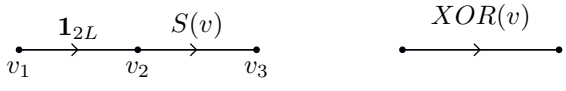


Figure 5.4: $XOR(v)$ gadget and its schematic representation.

The $Edge(u, v)$ gadget checks whether the constraints of an edge $\{u, v\} \in E(G)$ are satisfied and creates a "copy" of u and v. It consists of vertices $u', v', e_1, e_2, u'', v''$ which are connected as follows (see Figure 5.5). The arcs from u' (respectively v') to e_1 have capacities $S(v)$ (respectively $S(u)$) and the arcs from e_1 to e_2 have capacities $S(u, v)$. We add an $XOR(u)$ (respectively $XOR(v)$) gadget from e_2 to u'' (respectively v'').

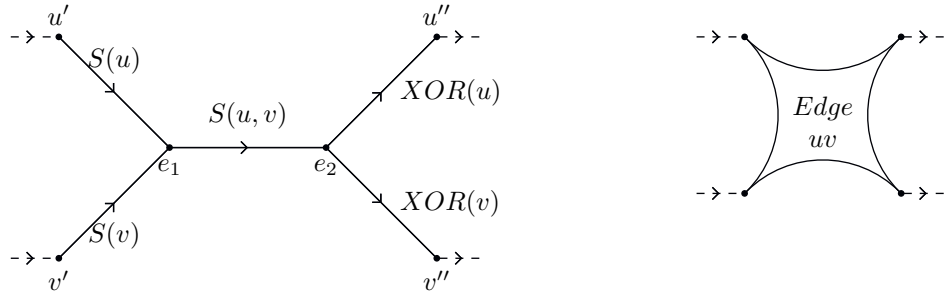


Figure 5.5: The *Edge* gadget and its schematic representation.

In our construction, the vertices u', v' will have no other outgoing arcs, and the vertices u'', v'' will have no other incoming arcs. The following claim follows from the Sidon set property:

Claim 5.1. If there is nonzero incoming flow to u' or v', then the outgoing flow of u'' (v'' respectively) is the same as the incoming flow to u' (v' respectively).

Proof of Claim: If the incoming flow to u' is not an element of $S(u)$, the flow conservation property fails for u' (and analogously for v'). In other words, we can suppose that the incoming flow to u' is equal to $s(c)$ for some $c \in S(u)$ and the incoming flow to v' is equal to $s(c')$ for some $c' \in S(v)$.

Thus the incoming flow to e_1 is equal to $s(c) + s(c')$. If $(c, c') \notin C(u, v)$, there is no outgoing arc from e_1 through which we can send the incoming flow. Namely, by the Sidon set property, there are no colors $(d, d') \in C(u, v)$ such that $s(c) + s(c') = s(d) + s(d')$ and $\{c, c'\} \neq \{d, d'\}$. Note that we also cannot split the outgoing flow into multiple outgoing arcs, since the incoming flow is between $2L$ and $4L - 2$, and each outgoing arc has capacity at least $2L$. Therefore, we must have $(c, c') \in C(u, v)$.

Let us now consider the outgoing flow from e_2. Since it is at least $2L + 1$, both $XOR(v)$ and $XOR(u)$ will have nonzero flow. By the Sidon set property and the fact that $C(u)$ and $C(v)$ are disjoint, we conclude that the incoming flow to u'' is $s(c)$, and the incoming flow to v'' is $s(c')$. □

The *Swap* gadget is identical to the *Edge* gadget with no constraints and u' and v' swapped (see Figure 5.6). Analogously, the outgoing flow from u'' (respectively v'') is the same as the incoming flow to u (respectively v).

Construction of H. We will now construct the graph H step by step, according to the series $\mathcal{S}$ of Introduce, Forget, Swap and Add-Edge operations that give $G =$

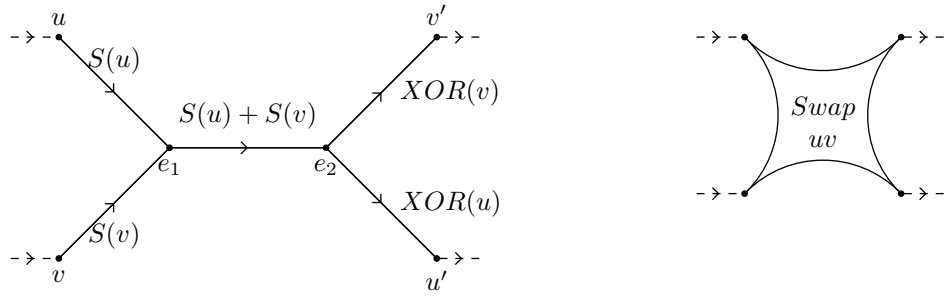


Figure 5.6: The *Swap* gadget and its schematic representation.

$(V, E, \emptyset)$. We will denote the ith operation in this sequence by σ_i. We will denote by $X_i \subseteq V(G)$ the set of terminals after step i and by $\prec_i$ the ordering of X_i.

Informally, the graph H will consist of several parts. For each operation σ_i, we add a column C_i containing $|X_i|$ vertices to H. Each vertex $v \in C_i$ will have a *label* $\ell(v) \in X_i$, and the vertices in C_i will be ordered from bottom to top according to the ordering $\prec_i$ of their labels.

If σ_i is an Add-Edge (respectively Swap) operation, we will add the Edge (respectively Swap) gadget between the corresponding vertices in C_{i-1} and C_i. We will also add arcs between the remaining vertices in C_{i-1} and C_i with the same label, which will ensure that all the vertices with the same label have the same flow. The graph H will also contain a path on the bottom which will start at the source and will introduce vertices according to the order in which they are introduced in $\mathcal{S}$. Lastly, we will add arcs to the sink to enforce flow conservation (see Figure 5.7).

Let us now formally describe H and its embedding in the plane. To facilitate the description, we will specify the coordinates of some vertices. Firstly, we draw the source vertex s at $(0, 0)$ and the sink t at $(0, -1)$. For each $i \in \{1, ..., |\mathcal{S}|\}$ and $j \in \{1, ..., |X_i|\}$, we draw a vertex h_{ij} at (i, j). We denote the ith column by $C_i = \{h_{i1}, \ldots, h_{i|X_i|}\}$. We set $\ell(h_{ij})$ to be the jth smallest element of X_i according to $\prec_i$.

We introduce an auxiliary variable *last* and set $last = 1$. Set $p_0 = s$. For $i = 1, ..., |\mathcal{S}|$, we add the following vertices and arcs depending on σ_i:

- **Insert** a vertex $v \in V(G)$: construct a vertex p_{last} at $(i, 0)$ and add an arc of capacity $(2n - last + 1)L$ from p_{last-1} to p_{last}. Add a gadget $XOR(v)$ starting from p_{last} and ending at h_{i1} (recall that $\ell(h_{i1}) = v$). Add $L - 1$ arcs of capacity 1 from p_j to t. For $j \in \{1, ..., |X_{i-1}|\}$, add L arcs of capacity 1 from h_{ij} to $h_{i+1,j+1}$. Increase *last* by 1.

- **Forget** a vertex $v \in V(G)$: let j be such that $\ell(h_{ij}) = v$. Add L arcs from h_{ij} to t of capacity 1. Since v is the maximum element of X_i, there are no vertices above h_{ij}, which implies that the arcs from h_{ij} to t can be drawn without intersecting any other arcs.

- **Add-Edge** $\{u, v\} \in E(G)$: let j be such that $\ell(h_{i-1,j}) = u$, $\ell(h_{i-1,j+1}) =$

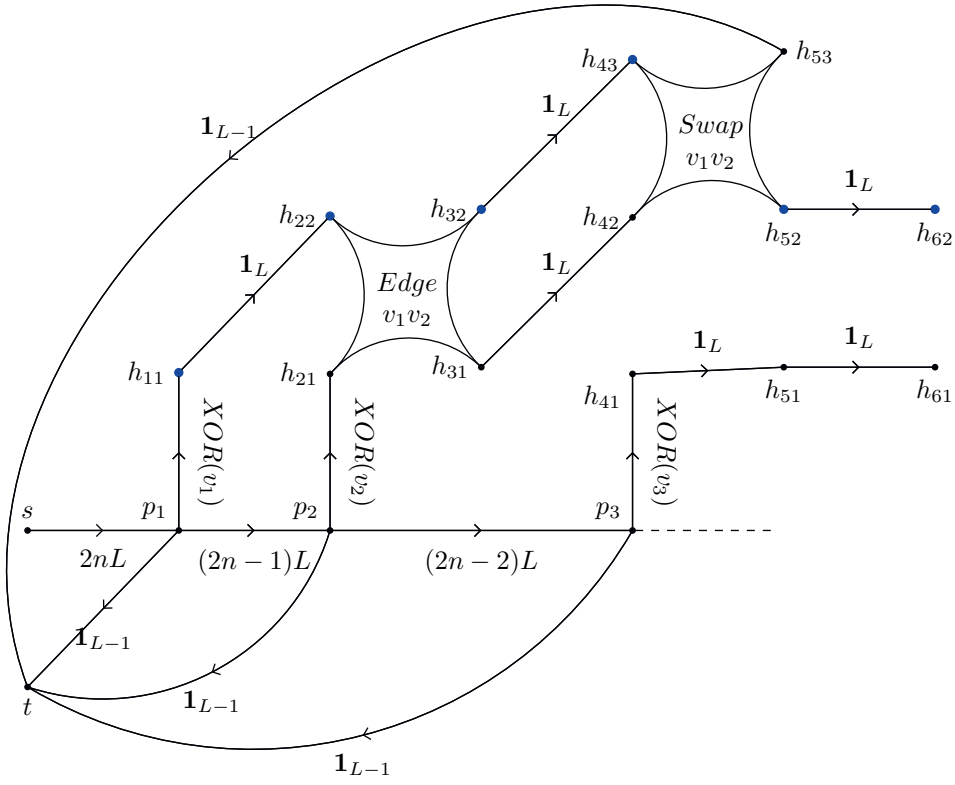


Figure 5.7: Graph H corresponding to the following sequence of operations: $Introduce(v_1)$, $Introduce(v_2)$, $Add-Edge(v_1, v_2)$, $Introduce(v_3)$, $Swap(v_1, v_2)$, $Forget(v_2)$. The blue vertices are labelled with v_1.

v. Add the gadget Edge between the vertices $h_{i-1,j}, h_{i-1,j+1}, h_{i,j}, h_{i,j+1}$ as in Figure 5.7.

- **Swap** u and v: analogous to the Add-Edge case, except that we add the Swap gadget instead.

Finally, we add nL arcs of capacity 1 from p_n to t. We also add L arcs of capacity 1 from each vertex in the last column (i.e. $h_{|\mathcal{S}|,j}$) to t.

Claim 5.2. For all $j \in \{1, \ldots, n\}$, the arc p_{j-1} to p_j has nonzero flow, and there is nonzero flow going out from p_j via the corresponding XOR gadget.

Proof of Claim: We will prove the claim by induction. Clearly, the arc from $p_0 = s$ to p_1 is used, since that is the only outgoing arc from the source. Suppose now that for all $j' < j$, the arc from $p_{j'-1}$ to $p_{j'}$ is used. Let the label of p_j be $\ell(p_j) = v$ and let p_j be introduced in the operation σ_i. By induction, the incoming flow of p_j is equal to $(2n - j + 1)L$. There are three ways for the flow to go out from p_j: through

the $XOR(v)$ gadget, through the arc to p_{j+1} and through the arcs to t. The total capacity of the arcs in the gadget and the arcs to t is at most $2L$, so the arc from p_j to p_{j+1} has to be used. Suppose that the outgoing flow through the $XOR(v)$ is zero. In that case, the maximum possible outgoing flow is $(2n-j)L+L-1 = (2n-j+1)L-1$, which leads to a contradiction. By the properties of the XOR gadget, the outgoing flow from p_j through $XOR(v)$ is equal to $s(c)$ for some color c, which proves the claim. □

Note that this also implies that the incoming flow to $h_{i,1}$ is equal to $s(c)$. It is easy to see that the same amount of flow will go through all the other vertices with the same label. Thus we can assign the color c to the vertex $v \in V(G)$. For each edge in G, the corresponding *Edge* gadget checks whether the edge constraint is satisfied, so we obtain a valid coloring of G.

It remains to show that the pathwidth and outerplanarity of H are bounded by $O(k)$. The ith bag of the path decomposition will contain t and all the vertices in the strip $\{(x, y) \in \mathbb{R}^2 : i - 1 \leqslant x \leqslant i\}$. It is easy to see that this gives a valid path decomposition of width at most $2k + 10$.

The vertex t is on the outer face of H. After deleting t, the vertices s and $p_1, \ldots, p_n$ are on the outer face, as well as the highest vertex in each column C_i. In each of the following steps, the highest remaining vertex in each column will be on the outer face (together with the gadgets adjacent to it). Since each column has at most k elements, the outerplanarity of H is at most $k + 1$. ■

5.6 Reductions from All-or-Nothing Flow

In this section, we build up on results of Section 5.5 and prove hardness of several problems. Figure 5.8 shows the reductions used in this section.

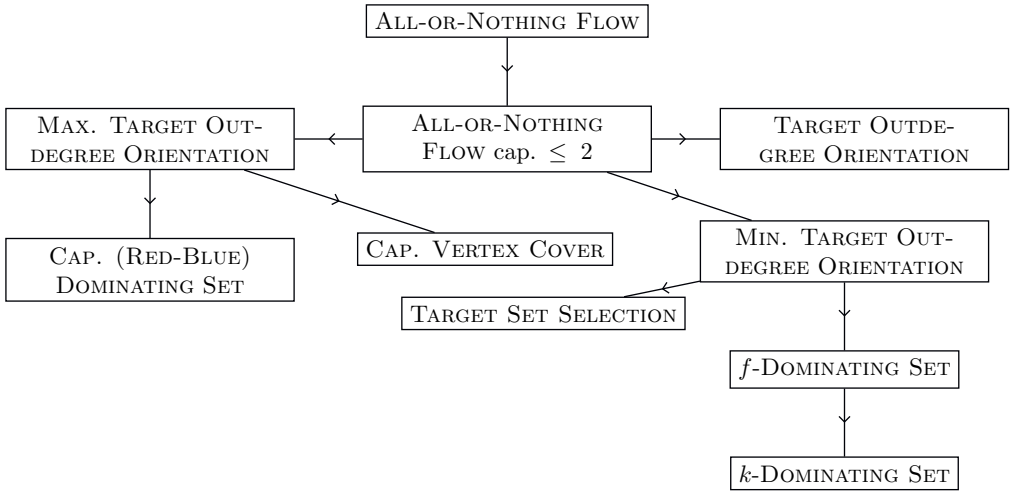


Figure 5.8: Reductions from All-or-Nothing Flow.

In all cases, the reductions replace edges or arcs by a small gadget with outerplanarity at most 2, and thus the graph stays planar and the outerplanarity increases by at most 2. This way, XNLP-hardness is obtained for each of the following problems on planar graphs with outerplanarity as parameter. Values are always given in unary.

5.6.1 All-or-Nothing Flow with Small Arc Capacities

We now look at the hardness of All-or-Nothing Flow for the case that all arc capacities are small. Note that when all arc capacities are 1, then a maximum all-or-nothing flow has the same value as a maximum flow, and we can use a standard flow algorithm like the Ford-Fulkerson algorithm to solve the problem in polynomial time. The next case, where capacities are 1 and 2 is already equally hard as the case where capacities are given in unary.

We will see that a simple transformation can transform an arc to an equivalent subgraph which is 2-outerplanar, and with all arc capacities 1 or 2. As the transformation to Target Outdegree Orientation given in the next section has edge weights that equal the arc capacities, the hardness results carry over to that problem.

Suppose xy is an arc with integer capacity $\gamma \geqslant 3$. We replace the arc xy by the following subgraph (see Figure 5.9):

- Take a directed path with $2\gamma - 4$ new vertices, say $v_1, v_2, \ldots, v_{2\gamma} - 4$.
- If $i \in [1, 2\gamma - 5]$ is odd, then the arc $v_i v_{i+1}$ has capacity 1.
- If $i \in [2, 2\gamma - 6]$ is even, then the arc $v_i v_{i+1}$ has capacity 1.
- Take an arc from x to v_1 of capacity 2.
- For each even $i \in [2, 2\gamma - 4]$, take an arc from x to v_i of capacity 1.
- For each odd $i \in [1, 2\gamma - 5]$, take an arc from v_i to y of capacity 1.
- Take an arc from $v_{2\gamma-4}$ to y of capacity 2.

Lemma 5.7. *Suppose we have a subgraph, as described above, with only x and y adjacent to vertices outside the gadget. Then for each all-or-nothing flow, either all arcs in the gadgets have flow 0, or all arcs in the gadget have flows equal to their capacity.*

Proof. Each vertex v_i in the gadget has either two incoming arcs of capacity 1 and one outgoing arc of capacity 2, or one incoming arc of capacity 2 and two outgoing arcs of capacity 1. So either all three arcs incident to v_i are used, or all three have flow 0. Suppose for contradiction that the claim from the lemma does not hold. Then there must be a vertex v_i with inflow and outflow 0, and a vertex $v_{i'}$ with inflow and outflow 2. Moreover, there must be such a pair that is adjacent ($|i - i'| = 1$), which gives a contradiction. ■

Corollary 5.2. *The* All-or-Nothing Flow *problem with all arcs of capacity 1 or 2 is:*

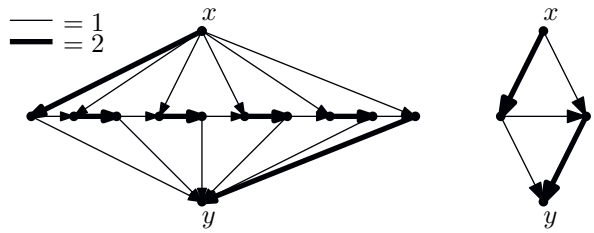


Figure 5.9: Examples of the gadget for arcs with capacities > 2. Fat arcs have capacity 2; thin arcs have capacity 1. The left gadget in the example functions as an arc with capacity 7; the right for an arc with capacity 3.

1. *XNLP-hard for planar graphs with outerplanarity as parameter.*
2. *XNLP-complete with pathwidth as parameter.*
3. *XALP-complete with treewidth as parameter.*

Proof. For each of the hardness proofs, we replace each arc with capacity more than 2 by the gadget described above. Notice that this step increases the outerplanarity of a planar graph by at most 1 (the vertices v_i are at most one level more away from the outer face than x or y) and the pathwidth of a graph by at most 2 (take a bag containing x and y, replace it by $2\gamma - 5$ copies, and add v_i, v_{i+1} to the ith copy). Also, the operation does not increase the treewidth of a graph whose treewidth is at least three: the gadget has treewidth at most 3, and we can attach a tree decomposition of a gadget to a bag that contains x and y. ■

5.6.2 Target Outdegree Orientation

In this section, we look at the problems Target Outdegree Orientation, Maximum Outdegree Orientation and Minimum Outdegree Orientation, and show that they are XNLP-hard when parameterized by outerplanarity. Given a directed graph $G = (V, E)$ with edge weights $w(e)$, the *weighted outdegree* of a vertex is the sum of the weights of all arcs with v as tail: $\sum_{vx \in E} w(vx)$.

Target Outdegree Orientation
Input: Undirected graph $G = (V, E)$, a positive integer weight $w(e)$ for each edge $e \in E$, and for each vertex $v \in V$, a positive integer target $t(v)$.
Task: Is there an orientation of G, such that for each v, the weighted outdegree of v equals $t(v)$?

The Minimum Outdegree Orientation and Maximum Outdegree Orientation problem are defined as above, except that now the weighted outdegree of each vertex v must be at least $t(v)$, respectively at most $t(v)$. Each of these three problems was shown to be XNLP-complete with pathwidth as parameter [18], FPT with tree partition width as parameter [18], and XALP-complete with treewidth as parameter [22].

To show hardness of each of the three problems, we use a simple transformation from ALL-OR-NOTHING FLOW: drop directions of all arcs, take weights equal to capacities, and choose targets for vertices in an appropriate way. Our transformation is similar to, but somewhat simpler than the transformation given in [18].

Suppose we have a flow network (G, s, t, cap) with $G = (V, E)$ a directed graph, $c : E \to \mathbb{Z}^+$ the capacity function, $s, t \in V$. Suppose we want to decide if there is a flow from s to t of value exactly r.

We assume that there are no parallel arcs, and also for each pair of vertices $x, y \in V$, at most one of the arcs xy and yx is in E; if not, we can obtain an equivalent instance by subdividing arcs and giving both new arcs the same capacity as the old arc.

Let $H = (V, F)$ be the undirected graph, obtained by dropping directions of arcs in E, and let the weight $w(e)$ of an edge be equal to the capacity of the directed variant of e in G.

For each vertex $v \in V \backslash \{s, t\}$, set $t(v) = \sum_{xv \in E} \mathsf{cap}(xv)$. Set $t(s) = \sum_{xs \in E} \mathsf{cap}(xs) + r$, and $t(t) = \sum_{xt \in E} \mathsf{cap}(xt) - r$.

Lemma 5.8. *Let G and H be as above. The following are equivalent.*

1. *There is an all-or-nothing flow of value r in G.*
2. *H has on orientation with each vertex weighted outdegree exactly $t(v)$.*
3. *H has on orientation with each vertex weighted outdegree at least $t(v)$.*
4. *H has on orientation with each vertex weighted outdegree at most $t(v)$.*

Proof. $1 \Rightarrow 2$: Consider a flow f. If an arc xy has positive flow, then we direct the edge $\{x, y\}$ in H in the same way as the arc in G, i.e. from x to y. If xy has 0 flow, then we direct the edge in the opposite direction, i.e. from y to x.

Consider a vertex $v \in V \setminus \{s, t\}$. Suppose the total flow sent to v by f (i.e. the *inflow* $\sum_{xv \in E} f(xv)$) is α. Then, $t(v) - \alpha$ weight of incoming edges is not used; thus, the incoming arcs in G give weighted outdegree $t(v) - \alpha$ in H. The total weight of used outgoing arcs from v is α; these are directed out of v, while all other outgoing arcs are directed in the opposite direction; this gives a contribution of α to the outdegree. So, the total outdegree equals $t(v) - \alpha + \alpha = t(v)$.

If the inflow of s by f is α, then the outflow of s by f is $\alpha + r$; we thus have $t(s) - r - \alpha$ weight of incoming edges not used by f, thus incoming arcs amount to weighted outdegree $t(s) - r - \alpha$; The analysis for t is similar.

$2 \Rightarrow 1$: Suppose we have an orientation with each vertex meeting its target. For each arc xy, send flow equal to its capacity over it, if and only if the edge $\{x, y\}$ is oriented from x to y (i.e., in the same way as the corresponding arc in H); otherwise we send 0 flow over the arc. Consider a vertex v. Suppose the total weight of arcs xv whose edges are directed as xv is β. Then, the weight of the edges with an incoming arc to v which are directed out of v is $t(v) - \beta$, so the weight of the edges vy with an outgoing arc from v who are directed as vy is β; so the inflow and outflow of v equal β. The analysis for s and t is similar.

$2 \Rightarrow 3, 4$ is trivial: use the same orientation.

$3 \Rightarrow 2$: we can use the same orientation. Note that the sum over all vertices of the weighted outdegree must equal the sum of the weights of all edges (as each edge is directed out of exactly one vertex in an orientation), and the latter sum equals $\sum_{v \in V} t(v)$. Thus if we have a vertex whose weighted outdegree is strictly greater than its target, then there must be another vertex whose weighted outdegree is strictly smaller than its target, which leads to a contradiction.

$4 \Rightarrow 2$ is similar as the previous case. ■

Corollary 5.3. *The* Target Outdegree Orientation, Minimum Outdegree Orientation, *and* Maximum Outdegree Orientation *problems with all arcs of capacity 1 or 2 is:*

1. *XNLP-hard for planar graphs with outerplanarity as parameter.*
2. *XNLP-complete with pathwidth as parameter.*
3. *XALP-complete with treewidth as parameter.*

Proof. Start with an instance of All-or-Nothing Flow with all capacities 1 or 2, see Corollary 5.2. Then, set the targets as described above and drop the directions of arcs. ■

5.6.3 Capacitated (Red-Blue) Dominating Set

We can use the Minimum Outdegree Orientation and Maximum Outdegree Orientation problems as starting problems for reductions to several problems, including Capacitated Red-Blue Dominating Set, Capacitated Dominating Set, Capacitated Vertex Coverm Target Set Selection, and f-Dominating Set. In each of these cases, we take an instance of Minimum Outdegree Orientation or Maximum Outdegree Orientation and replace each edge by an appropriate planar subgraph. We start with the capacitated variants of Dominating Set and Vertex Cover, which use existing transformations from the literature.

Capacitated Red-Blue Dominating Set
Input: Bipartite graph $G = (R \cup B, E)$, with each "blue" vertex in $v \in B$ a positive integer capacity $c(v)$, and an integer k
Task: Is there a set $S \subseteq B$ of at most k blue vertices, and an assignment $f : R \to S$ of each red vertex to a (blue) neighbour in S, such that each vertex in $v \in S$ has at most $c(v)$ red neighbours assigned to it?

Capacitated Dominating Set
Input: Graph $G = (V, E)$, with each vertex in $v \in R$ a positive integer capacity $c(v)$, and an integer k
Task: Is there a set $S \subseteq V$ of at most k vertices, and an assignment $f : V \setminus S \to S$ of each vertex not in S to a neighbour in S, such that each vertex in $v \in S$ has at most $c(v)$ neighbours assigned to it?

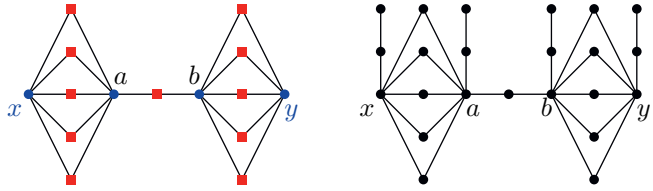


Figure 5.10: Left: Example of edge transformation for Capacitated Red-Blue Dominating Set. Right: Example of edge transformation for Capacitated Dominating Set. The edge has weight 5, a and b have capacity 6. x has capacity $t(x)$; y has capacity $t(y)$. All other black vertices (right figure) have capacity 1.

Capacitated Red-Blue Dominating Set and Capacitated Dominating Set are well studied problems. Both problems are XNLP-complete with pathwidth as parameter [21] and XALP-complete with treewidth as parameter [22]. An earlier W[1]-hardness proof for treewidth as parameter can be found in [52]. Capacitated Dominating Set was shown to be W[1]-hard for planar graphs with the solution size as parameter [24]. It follows from a lower bound proof in [61] that Capacitated Red-Blue Dominating Set is W[1]-hard with feedback vertex set as parameter.

In [21], the following transformation from Maximum Outdegree Orientation to Capacitated Red-Blue Dominating Set is given: each edge $e = \{v, w\}$ with capacity γ is replaced by a subgraph with $2\gamma + 3$ additional vertices. Take γ red vertices adjacent to v and γ red vertices adjacent to w. Take a new blue vertex adjacent to the each vertex in the first set of γ red vertices, and a new blue vertex adjacent to the each vertex in the second set of γ red vertices. Then take one red vertex, adjacent to each of the latter two blue vertices. Finally, we colour all original vertices blue. The blue vertices in the edge gadgets have capacity $\gamma + 1$; the original vertices $v \in V$ have capacity $t(v)$. We ask if there is a red-blue dominating set of size $|V| + |E|$. See Figure 5.10 (left side) for an example.

Correctness of the transformation is easy to see (cf. [21]); the intuition is that there is always an optimal solution S that contains all original vertices in V, and for each edge gadget, one of the blue vertices (marked a and b in Figure 5.10) is chosen. If a is chosen, we direct the edge from y to x; if b is chosen, the edge is directed from x to y. A vertex dominates all neighbouring vertices in edge gadgets of outgoing edges - a solution of the Maximum Outdegree Orientation thus has the property that original vertices $v \in V$ dominate at most $t(v)$ neighbours.

Transforming to Capacitated Dominating Set is done by dropping all colours, and giving each vertex that was blue an adjacent P_2, with the first vertex on this P_2 having weight 2. All vertices that were originally red have capacity 1. On each of these P_2's, one vertex must be placed in the solution; we can choose the vertex incident to the originally blue vertex, which thus is dominated. See [21]. An example can be found in Figure 5.10 (right side).

Note that the transformations keep the planarity of the graph invariant, and thus we can conclude:

Theorem 5.9. Capacitated Red-Blue Dominating Set *and* Capacitated Dom-

INATING SET *are XNLP-hard for planar graphs with outerplanarity as parameter.*

5.6.4 Capacitated Vertex Cover

Another well studied "capacitated" variant of a classic graph problem is CAPACITATED VERTEX COVER. It is XNLP-complete for pathwidth [21], XALP-complete for treewidth [21], improving upon an earlier W[1]-hardness proof [52]. Dom et al. [52] give an $O(2^{O(tw \log k)} n^{O(1)})$ time algorithm with k the solution size and tw the treewidth and thus show the problem to be FPT for the combined parameter of treewidth and solution size.

CAPACITATED VERTEX COVER
Input: Graph $G = (V, E)$, positive integer capacity $c(v)$ for each $v \in V$, integer k
Task: Is there a set of vertices $S \subseteq V$, with $|S| \leqslant k$, and an assignment of each edge to an incident vertex in S such that no vertex $v \in S$ has more than $c(v)$ edges assigned to it?

In [21], the following reduction from MAXIMUM OUTDEGREE ORIENTATION to CAPACITATED VERTEX COVER is discussed: replace each edge $\{x, y\}$ of weight γ by a gadget, with new vertices a, b, γ paths of length 3 from x to a, one edge from a to b, and γ paths of length 3 from b to y. For each original vertex $v \in V$, add one pendant neighbour. Set for all new vertices their capacity equal to their degree, and for the original vertices $v \in V$, their capacity equal to their target outdegree plus 1. See Figure 5.11 for an example.

The correctness proof of the transformation is similar to that for CAPACITATED RED-BLUE DOMINATING SET but was not given in [21].

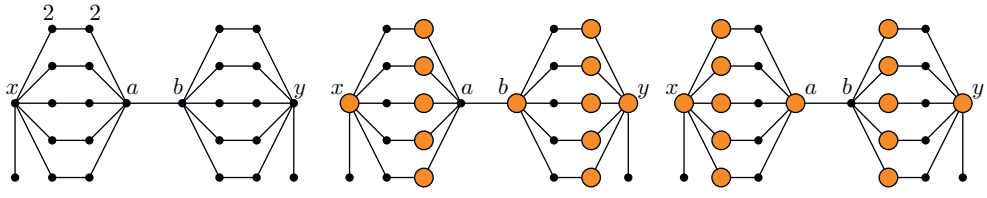


Figure 5.11: Edge gadget for CAPACITATED VERTEX COVER. If the edge $\{x, y\}$ has weight γ, then $c(a) = c(b) = \gamma + 1$. The capacity of the degree-2 vertices in the gadget is 2. $c(x) = t(x) + 1$; $c(y) = t(y) + 1$. Middle: corresponds to directing the edge from x to y. Right: corresponds to directing the edge from y to x.

Lemma 5.10. *Let H be the graph obtained from G by applying the reduction as described above to each edge of G, with the given capacities. G has an orientation with each vertex $v \in V$ weighted outdegree at most $t(v)$, if and only if H has a capacitated vertex cover of size at most $|V| + \sum_{e \in E}(2w(e) + 1)$.*

Proof. Suppose we have an orientation of G with each vertex outdegree at most its target. Take the following set S: place all original vertices (in V) in S, and for each

edge xy, if it is directed from x to y, all vertices in the gadget with even distance to x, and if it is directed from y to x, all vertices with odd distance to y, see Figure 5.11.

All vertices inside the gadget have capacity equal to their degree, and when in S, get all incident edges assigned to it. A vertex $x \in V$ has assigned to it its incident vertex of degree one, and the neighbours in edge gadgets of outgoing edges. Note that the number of the latter type of neighbours equals the weighted outdegree of v in the orientation. Thus, we have a capacitated vertex cover of the correct size.

Now, suppose H has a capacitated vertex cover S of size at most $|V|+\sum_{e\in E}(2w(e)+1)$. For each $x \in V$, either $x \in S$ or the degree-one neighbor of x is in S. Also, note that in an edge gadget, we have one vertex in S for each pair of adjacent degree-two vertices, and $a \in S$ or $b \in S$. Therefore, an edge gadget of an edge of weight γ must contain at least $2\gamma + 1$ vertices in S. It follows that we cannot place both a vertex $x \in V$ and its degree-one neighbor in S, and that we use exactly $2w(e) + 1$ vertices from an edge gadget of e.

Now, if we would use a degree-one neighbor of $x \in V$, we can swap this vertex with x, and also have a solution. Thus $V \subseteq S$, and each vertex $x \in V \cap S$ has its incident degree one edge associated to it. Consider an edge $e = \{x, y\}$. If $b \in S$, then orient the edge as xy; otherwise, $a \in S$ and we orient the edge as yx. If $b \in S$, $a \notin S$, and thus all neighbors of a are in S. This means that the edges from x to the gadget must be covered by x, and thus x have $w(e)$ edges from the gadget assigned to it. The number of gadget edges assigned to x thus equals its weighted outdegree, which is at most $t(x)$. The lemma now follows. ■

As the transformation maintains planarity, and increases the outerplanarity of a graph by at most 1, the following result is obtained.

Theorem 5.11. Capacitated Vertex Cover *is XNLP-hard for planar graphs with outerplanarity as parameter.*

5.6.5 f-Domination and k-Domination

In contrast to capacitated versions of dominating set where vertices can dominate only a limited number of neighbours, in the f-Dominating Set problem vertices not in the solution must be dominated multiple times.

f-Dominating Set
Input: Graph $G = (V, E)$, demand function $f : V \to \mathbb{N}$, integer k
Task: Is there a set $S \subseteq V$ such that $|S| \geqslant k$ and for each $v \in V$, we have $v \in S$ or $|N(v) \cap S| \leqslant f(v)$?

A special case of f-Dominating Set is the k-Dominating Set problem, where we $f(v)$ is equal for all vertices $v \in V$. Note that if k is fixed, standard (dynamic programming) techniques give an FPT algorithm with treewidth as parameter.

k-Dominating Set
Input: Graph $G = (V, E)$, integers ℓ, k
Task: Is there a set $S \subseteq V$ such that $|S| \leqslant \ell$ and for each $v \in V$ we have $v \in S$ or $|N(v) \cap S| \geqslant k$.

The notion of a k-dominating set was introduced in 1985 by Fink and Jacobson [60] and was mostly studied from a graph theoretic perspective. If k is fixed, then the k-Dominating Set problem can be solved in $O(k^{O(tw)}n)$ time, using dynamic programming on tree decompositions [116]. The generalization of f-domination was introduced by Zhou, see e.g. [38]. Again, the f-Dominating Set can be easily seen to be fixed-parameter tractable when we take a combined parameter of treewidth and the maximum value of f, $k = \max_{v \in V} f(v)$, as parameter, also by using dynamic programming on tree decompositions.

The hardness proof is quite similar. An edge of weight γ is replaced by the following structure. We take a triangle with three vertices, say a, b, and c. Take γ copies of a path of three vertices, and make the first vertices on these paths adjacent to x, and the last (third) vertices of these paths adjacent to a. Also, take γ copies of a path of three vertices, and make the first vertices on these paths adjacent to b, and the last (third) vertices of these paths adjacent to y. See Figure 5.12 (left side) for an example. For all vertices in the new edge gadgets, we set their f-values equal to 1. For an original vertex $v \in V$, we set $f(v) = t(v)$.

Let H be the resulting graph, with domination demand function f.

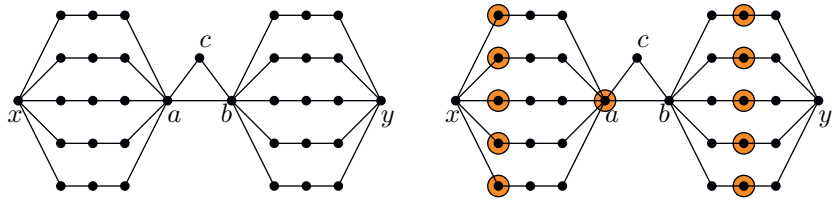


Figure 5.12: Example transformation of an edge for the f-Dominating Set problem. The right side corresponds to an orientation of the edge from x to y. All new vertices have $f(v) = 1$; $f(x) = t(x)$; $f(y) = t(y)$

Lemma 5.12. *H has an f-dominating set of size $\sum_{e \in E}(2w(e)+1)$, if and only if G has an orientation with each vertex outdegree at least $t(v)$.*

Proof. Consider an orientation of G such that each vertex has outdegree at least $t(v)$. We will construct an f-dominating set S in H as follows. For each edge $xy \in E(G)$ that is oriented from x to y, we add to S the first vertex of each path from x to a, the vertex a, and the middle vertex of each path from b to y. Note that for each such edge we added $2w(e)+1$ vertices to S, so $|S| = \sum_{e \in E(G)}(2w(e)+1)$. It is easy to see that all the new vertices in H have a neighbour in S. Consider a vertex $x \in V(G)$ and an edge $xy \in E(G)$ oriented from x to y. This edge contributes $w(e)$ to the outdegree of x in G, and it contributes $w(e)$ to the number of neighbours in the set S in H. If an edge $xy \in E(G)$ is oriented from y to x, it contributes neither to the outdegree of x in G nor to the number of neighbours of v in S in H. The other direction (i.e. constructing an orientation of G from an f-dominating set in H) is analogous. ■

Theorem 5.13. *The f-Dominating Set problem is:*

1. *XNLP-hard for planar graphs with outerplanarity as parameter.*

2. *XNLP-complete with pathwidth as parameter.*

3. *XALP-complete with treewidth as parameter.*

Proof. We reduce from Target Outdegree Orientation. We construct the graph H as in the above description. Using Lemma 5.12, we obtain the desired result. ■

Corollary 5.4. *The* k-Dominating Set *problem is:*

1. *XNLP-hard for planar graphs with outerplanarity as parameter.*

2. *XNLP-complete with pathwidth as parameter.*

3. *XALP-complete with treewidth as parameter.*

Proof. The reduction for the hardness proofs is the following.

Suppose we are given an instance of f-Dominating Set, with $G = (V, E)$ a graph, f a demand function, and ℓ the maximum solution size. Let $k = \max_{v \in V} f(v)$. We may assume $k > 1$, otherwise the problem is fixed-parameter tractable (this can be shown by standard techniques).

Now, to each vertex $v \in V$, add $k - f(v)$ new vertices, only adjacent to v. Increase ℓ by the total number of added new vertices. Correctness follows by observing that all new vertices must be an element of the dominating set (they have only one neighbour in G), and now each vertex in V must be in the dominating set, or has at least $f(v)$ neighbours in V in the dominating set (as it has $k - f(v)$ neighbours that are new vertices in the dominating set.) ■

5.6.6 Target Set Selection

The Target Set Selection problem models the viral marketing process, i.e. the style of promotion relying on consumers recommending the product to their social network. Informally, the problem setup is as follows. We are given a graph corresponding to a social network, and we want to advertise our product by giving it away to at most k "influencers" in this network. They will spread the word about our product and convince others to buy it — each person will buy the product if it was recommended by at least a certain number (i.e. threshold) of their friends.

Formally, given a graph $G = (V, E)$, a threshold function $t : V \to \mathbb{N}$ and $S \subseteq V$, the *activation process in* G *starting with* S is a sequence of subsets $Active[0] \subseteq Active[1] \subseteq \ldots$ such that $Active[0] = S$ and a vertex v belongs to $Active[i]$ if it belongs to $Active[i-1]$ or it has at least $t(v)$ neighbours in $Active[i-1]$. We repeat this process until we get $Active[j] = V$ or $Active[j] = Active[j-1]$ for some $j \in \mathbb{N}$, and we define $Active(S) = Active[j]$.

Target Set Selection
Input: Graph $G = (V, E)$, a threshold $t : V \to \mathbb{N}$, integer k
Task: Is there a set $S \subseteq V$ such that $Active(S) = V$?

Ben-Zwi et al. [10] gave an XP algorithm with treewidth as parameter, and an $n^{\Omega(\sqrt{tw})}$ lower bound. Their proof also implies W[1]-hardness.

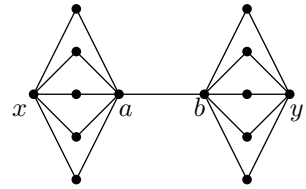


Figure 5.13: Transformation for Target Set Selection. The example transforms an edge of weight $\gamma = 5$. a and b have threshold $\gamma + 1 = 6$; x has a threshold equal to $t(x)$; y has a threshold equal to $t(y)$; the other (degree 2) vertices have threshold 1.

For our hardness proofs, we use an edge gadget that is again similar to previous edge gadgets; it is illustrated in Figure 5.13. Here, we reduce from Minimum Outdegree Orientation. Original vertices have a threshold which is equal to their target outdegree; the vertices a and b have a threshold equal to their degree, i.e., the weight of the edge plus one, and the vertices of degree two in the gadget have a threshold of 1. We will call the obtained graph H.

Lemma 5.14. *We can activate H by initially activating $|E(G)|$ vertices, if and only if G has an orientation with each vertex $v \in V$ total weighted outdegree at least $t(v)$.*

Proof. Suppose we can activate H by initially activating a set S such that $|S| = |E(G)|$.

Firstly note that in each edge gadget, we must initially activate either a or b: otherwise, at least one of them cannot reach the threshold. Now, in each edge gadget, exactly one of the vertices a and b is in S, as otherwise we would have too many vertices in S. Consider an edge gadget where $b \in S$. In order to activate a, we must activate all of its degree 2 neighbours, which means we must activate x before that. In other words, in order to activate x, we cannot use any of its neighbours in this gadget. Thus, to activate x, we must use its degree 2 neighbours from edge gadgets where $a \in S$ (note that in these gadgets all the degree 2 vertices between a and x are activated in the same round).

This gives us an orientation of edges in G: for each edge $xy \in E(G)$, orient the edge from x to y if the corresponding vertex a is in S and from y to x otherwise. By the above arguments, there are at least $t(x)$ neighbours of x activated before x, which means that the sum of weights of all outgoing edges from x is $t(x)$. The reverse direction is analogous. ■

Theorem 5.15. *The* Target Set Selection *problem is:*

1. *XNLP-hard for planar graphs with outerplanarity as parameter.*
2. *XNLP-complete with pathwidth as parameter.*
3. *XALP-complete with treewidth as parameter.*

Proof. As the transformation described above can be done in logarithmic space, hardness follows from the hardness for Minimum Outdegree Orientation.

Membership follows from a modification of the XP algorithm in [10]. Instead of building the entire DP table for a path decomposition, we guess the "next element". In a tree decomposition, we traverse the tree in post-order, guessing elements instead of building the entire DP table; in a (join) node with two children, after handling the left branch, we store its result on a stack, then handle the right branch, and then combine the results of left and right branch. ■

5.7 Conclusion

In this chapter, we showed XALP-completeness or XNLP-hardness for several problems on planar graphs parameterized by outerplanarity. In a number of cases, these were problems not yet established to be XALP-complete for treewidth, and thus for these, we also obtained new results for the general class of graphs. For several "flow-like" problems, we could only show XNLP-hardness; we conjecture that these are also XALP-complete parameterized by outerplanarity. It would be sufficient to show that All-or-Nothing Flow is XALP-hard for outerplanarity; the follow up transformations can be used to show several other problems including Capacitated Dominating Set, Target Set Selection are XALP-complete.

The results are also interesting in the light of a conjecture by Pilipczuk and Wrochna [107], which claims that an XNLP-hard problem has no XP algorithm that uses $O(f(k)n^{O(1)})$ space. Our results are negative in the sense that it appears that for the problems studied in this setting, going from graphs of bounded treewidth to graphs of bounded outerplanarity, the complexity does not drop.

We expect that more problems that are in XP with outerplanarity as parameter are complete for XALP, and leave finding more examples as an interesting open problem. One candidate is Multicommodity Flow: in [25], it was shown that it is XALP-complete for treewidth and XNLP-complete for pathwidth; capacities are given in unary. The result already holds for 2 commodities. The gadgets used in that proof are non-planar, so establishing the complexity of Multicommodity Flow with outerplanarity seems to need new techniques.

Another interesting direction for further research is the following. Problems with an algorithm with running time of the form $2^{O(\ell)}n^{O(1)}$ with ℓ the treewidth or outerplanarity are in XP when we take "logarithmic treewidth" or "logarithmic outerplanarity" as parameter, i.e., the parameter is $\frac{\ell}{\log n}$. In [23], it was shown that Dominating Set and Independent Set are XNLP-complete with logarithmic pathwidth as parameter; this translates to XALP-completeness for these problems with logarithmic treewidth as parameter [22]. As these proofs produce non-planar graphs, it is open whether Dominating Set and Independent Set are XALP-complete when we take logarithmic outerplanarity as parameter.

Chapter 6

On the Parameterized Complexity of the Connected Flow and Many Visits TSP Problem

6.1 Introduction

In the Connected Flow problem we are given a directed graph $G = (V, E)$ with costs and capacities on the edges (cost and cap respectively) and a set $D \subseteq V$ such that each $v \in D$ has a fixed demand, described by the function dem. We then ask for a minimum cost *connected flow* on the edges that satisfies the demand for each $v \in D$, i.e. we look for a minimum cost *flow conserving* function $f : E \to \mathbb{N}$, such that the set of edges with strictly positive flow f is connected and the total flow coming into $v \in D$ is equal to its demand (see Section 6.2 for the formal problem definition).

One arrives at the Connected Flow problem by adding a natural connectivity constraint to the well known Flow problem. Unfortunately, Connected Flow has the same fate as many other generalizations of Flow: The additional requirement changes the complexity of the problem from being solvable in polynomial time to being NP-complete (see [65, Section A2.4] for more such NP-complete generalizations).

The problem generalizes a number of problems, including the Many Visits TSP (MVTSP)[1]. This problem has a variety of potential applications in scheduling and computational geometry (see e.g the discussion by Berger et al. [13]), and its study from the exponential time perspective recently witnessed several exciting results. In particular, Berger et al. [13] improved an old $n^{O(n)}$ time algorithm by Cosmadakis and Papadimitriou [41] to $O^\star(5^n)$ time and polynomial space, and recently the analysis

[1] In this problem a minimum length tour is sought that satisfies each vertex a given number of times. The generalization is by setting the demand of a vertex to the number of times the tour is required to visit that vertex and using infinite capacities.

of that algorithm was further improved by Kowalik et al. [85] to $O^\star(4^n)$ time.

The Connected Flow problem also generalizes other problems studied in parameterized complexity, such as the Eulerian Steiner Subgraph problem, that was used in an algorithm for Hamiltonian Index by Philip et al. [106], or the problem of finding two short edge disjoint paths in undirected graphs (whose parameterized complexity was for example studied by Cai and Ye [34]).

Based on these connections with existing literature on in particular the MVTSP, its appealing formulation, and it being a direct extension of the well-studied Flow problem, we initiate the study of the parameterized complexity of Connected Flow in this chapter.

Our Contributions. We first study the (arguably) most natural parameterization: the number of demand vertices for which we require a certain amount of flow. We show that the problem is NP-complete even in a very special case:

Theorem 6.1. Connected Flow *with* 2 *demand vertices is NP-complete.*

The reduction heavily relies on the capacities and we show that this is indeed what makes the problem hard. Namely, using the algorithm for MVTSP from [85], we get an algorithm that can solve instances of Connected Flow if all capacities are infinite:

Theorem 6.2. *Any instance* $(G, D, \mathsf{dem}, \mathsf{cost}, \mathsf{cap})$ *of* Connected Flow *where* $\mathsf{cap}(e) = \infty$ *for all* $e \in E$ *can be solved in time* $O^\star(4^{|D|})$.

Next we study a typically much larger parameterization, the size k of a vertex cover of G. One of our main technical contributions is that Connected Flow is FPT, parameterized by k:

Theorem 6.3. *There is an algorithm solving a given instance* $(G, D, \mathsf{dem}, \mathsf{cost}, \mathsf{cap})$ *of* Connected Flow *such that* G *has a vertex cover of size* k *in time* $O^\star(k^{O(k)})$.

Theorem 6.3 is interesting even for the special case of MVTSP as it generalizes the $O^\star(n^n)$ time algorithm from Cosmadakis and Papadimitriou [41], though it is a bit slower than the more recent algorithms from [13, 85]. For this special case, we even find a polynomial kernel:

Theorem 6.4. MVTSP *admits a kernel polynomial in the size* k *of the vertex cover of* G.

The starting point of the proofs of both Theorem 6.3 and Theorem 6.4 is a strengthening of a non-trivial lemma from Kowalik et al. [85] which proves the existence of a solution s' that is "close" to a solution r of the Flow problem instance obtained by relaxing the connectivity requirement. Since such an r can be found in polynomial time, it can be used to determine how the optimal solution roughly looks.

This is subsequently used by a dynamic programming algorithm that aims to find such a solution close to r to establish Theorem 6.3; the restriction to solutions being close to r crucially allows us to evaluate only $O^\star(k^{O(k)})$ table entries. Additionally, this is used in the kernelization algorithm of Theorem 6.4 to locate a set of $O(k^5)$

vertices such that only edges incident to vertices in this set will have a different flow in r and s'.

The last parameter we consider is the *treewidth*. We present a Dynamic Programming algorithm for Connected Flow:

Theorem 6.5. *Let M be an upper bound on the demands in the input graph G, and suppose a tree decomposition of width tw of G is given. Then a* Connected Flow *instance with G can be solved in time $|V(G)|^{O(tw)}$ and an* MVTSP *instance with G can be solved in time $\min\{|V(G)|, M\}^{O(tw)}|V(G)|^{O(1)}$.*

We also give a matching lower bound for MVTSP. This lower bound heavily builds on previous approaches, and in particular, some gadgets from Cygan et al. [46].

Theorem 6.6. *Assuming the Exponential Time Hypothesis,* MVTSP *cannot be solved in time $f(tw)|V(G)|^{o(tw)}$ for any computable function $f(\cdot)$.*

Note that since MVTSP is a special case of Connected Flow this lower bound extends to Connected Flow.

Organization. The remainder of this chapter is organized as follows: in Section 6.3 we study the parameterization by the number of demand vertices. We show NP-completeness and discuss the reduction of the infinite capacities case of Connected Flow to MVTSP.

In Section 6.4 we first introduce an extension of a lemma from Kowalik et al. [85] that shows that we can transform an optimal solution to the Flow relaxation to include a specific edge set from an optimal solution of the original Connected Flow instance, without changing too many edges. This lemma is subsequently used in Section 6.4.2 to prove Theorem 6.3 and in Section 6.4.3 to prove Theorem 6.4.

In Section 6.5 we discuss the parameterization by treewidth and pathwidth, giving a Dynamic Programming algorithm for Connected Flow and a matching lower bound for MVTSP.

We conclude the chapter with a discussion on further research opportunities.

6.2 Preliminaries

We assume that all integers are represented in binary, so in this chapter the input size will be polynomial in the number of vertices of the input graph and the logarithm of the maximum input integer. All graphs in this chapter are directed unless stated otherwise.

Recall that a multiset is an ordered pair (A, m_A) consisting of a set A and a multiplicity function $m_A : A \to \mathbb{Z}^+$. We slightly abuse notation and let $m_A(e) = 0$ if $e \notin A$. We can see flow f as a multiset of directed edges, where each edge appears $f(e)$ number of times. We then say that $f(e)$ is the *multiplicity* of e. Given a function $f : E \to \mathbb{N}$, we define $G_f = (V', E')$ as the multigraph where $e \in E'$ has multiplicity $f(e)$ and V' is the set of vertices incident to at least one $e \in E'$. We let $E(G_f)$ be equal to the multiset E'. We also define $\text{supp}(f) = \{e \in E : f(e) > 0\}$ as the *support* of f.

Recall the definition of Demand Flow, a problem equivalent to the standard Min Cost Flow:[2]

Demand Flow
Input: Directed graph $G = (V, E)$, $D \subseteq V$, $\mathsf{dem} : D \to \mathbb{N}$, $\mathsf{cost} : E \to \mathbb{N}$, $\mathsf{cap} : E \to \mathbb{N} \cup \{\infty\}$
Task: Find a function $f : E \to \mathbb{N}$ such that

- for every $v \in V$ we have $\sum_{u \in V} f(u, v) = \sum_{u \in V} f(v, u)$,
- for every $v \in D$ we have $\sum_{u \in V} f(u, v) = \mathsf{dem}(v)$,
- for every $e \in E : f(e) \leqslant \mathsf{cap}(e)$,

and the value $\mathsf{cost}(f) = \sum_{e \in E} \mathsf{cost}(e) f(e)$ is minimized.

Connected Flow is defined as Demand Flow with an additional connectivity requirement:

Connected Flow
Input: $G = (V, E)$, $D \subseteq V$, $\mathsf{dem} : D \to \mathbb{N}$, $\mathsf{cost} : E \to \mathbb{N}$, $\mathsf{cap} : E \to \mathbb{N} \cup \{\infty\}$
Task: Find a function $f : E \to \mathbb{N}$ such that

- G_f is connected,
- for every $v \in V$ we have $\sum_{(u,v) \in E} f(u, v) = \sum_{(v,u) \in E} f(v, u)$,
- for every $v \in D$ we have $\sum_{(u,v) \in E} f(u, v) = \mathsf{dem}(v)$,
- for every $e \in E$ we have $f(e) \leqslant \mathsf{cap}(e)$,

and the value $\mathsf{cost}(f) = \sum_{e \in E} \mathsf{cost}(e) f(e)$ is minimized.

Given an instance of Connected Flow, we call a flow r a *relaxed solution* if it is a solution of the Demand Flow instance obtained by removing the connectivity requirement.

Note that G_f in the above definition is Eulerian (every vertex has the same in and out degree), so it is strongly connected if and only if it is weakly connected.

In Kowalik et al. [85], the Many Visit TSP (MVTSP) is defined as follows.

Many Visits TSP (MVTSP)
Input: Directed graph $G = (V, E)$, $\mathsf{dem} : V \to \mathbb{N}$, $\mathsf{cost} : V^2 \to \mathbb{N}$
Task: Find a minimal cost tour c, such that each $v \in V$ is visited exactly $\mathsf{dem}(v)$ times.

Note that MVTSP is a special case of Connected Flow, where $D = V$ and the capacities of all edges are infinite.

[2] See Section 2.2 for more details and proof of equivalence. In particular, this implies that Demand Flow is polynomial-time solvable.

6.3 Parameterization by Number of Demand Vertices

In this section we study the parameterized complexity of Connected Flow with parameter $|D|$, the number of vertices with a demand. We first prove that the problem is NP-hard, even for $|D| = 2$, by a reduction from the problem of finding two vertex disjoint paths in a directed graph. Next we show that, if $\mathsf{cap}(e) = \infty$ for all $e \in E$, the problem can be reduced to an instance of MVTSP, and hence solved in time $O^\star(4^{|D|})$.

Theorem 6.1. Connected Flow *with* 2 *demand vertices is NP-complete.*

Proof. We give a reduction from the problem of finding two vertex-disjoint paths in a directed graph to Connected Flow with demand set D of size 2. The directed vertex-disjoint paths problem has been shown to be NP-hard for fixed $k = 2$ by Fortune et al. [63], so this reduction will prove our theorem for $|D| = 2$. Note that the case of $|D| > 2$ is at least as hard, since we can view $|D| = 2$ as a special case, by adding isolated vertices with demand 0.

Given a graph G and pairs (s_1, t_1) and (s_2, t_2), we construct an instance $(G', D, \mathsf{dem}, \mathsf{cost}, \mathsf{cap})$ of Connected Flow. Let $V_0 = V \setminus \{s_1, s_2, t_1, t_2\}$, we define

$$V(G') = \{s_1, s_2, t_1, t_2\} \cup \{v_{\mathsf{in}} : v \in V_0\} \cup \{v_{\mathsf{out}} : v \in V_0\}$$

We let $D = \{s_1, s_2\}$ and set $\mathsf{dem}(s_1) = \mathsf{dem}(s_2) = 1$. We also define

$$\begin{aligned} E(G') = &\{(v_{\mathsf{in}}, v_{\mathsf{out}}) : v \in V_0\} \\ &\cup \{(s_i, v_{\mathsf{in}}) : (s_i, v) \in E(G), i = 1, 2\} \\ &\cup \{(v_{\mathsf{out}}, t_i) : (v, t_i) \in E(G), i = 1, 2\} \\ &\cup \{(u_{\mathsf{out}}, v_{\mathsf{in}}) : u, v \in V_0, (u, v) \in E(G)\} \\ &\cup \{(t_1, s_2), (t_2, s_1)\}. \end{aligned}$$

We now set $\mathsf{cost}(u, v) = 0$ and $\mathsf{cap}(u, v) = 1$ for every $(u, v) \in E(G')$. We prove that G has two vertex-disjoint paths (from s_1 to t_1 and from s_2 to t_2) if and only if $(G', D, \mathsf{dem}, \mathsf{cost}, \mathsf{cap})$ has a connected flow of cost 0.

Let P_1 and P_2 be two vertex disjoint paths in G, from s_1 to t_1 and from s_2 to t_2 respectively. Intuitively we will simply walk through the same two paths in G' and then connect the end of one to the start of the other. More formally, we construct a flow f in G' as follows. Let $P_1 = s_1, v^1, \ldots, v^\ell, t_1$, we set $f(s_1, v^1_{\mathsf{in}}) = f(v^\ell_{\mathsf{out}}, t_1) = 1$ as well as $f(v^i_{\mathsf{in}}, v^i_{\mathsf{out}}) = f(v^i_{\mathsf{out}}, v^{i+1}_{\mathsf{in}}) = 1$ for all $i \in [1, \ell]$. We do the same for P_2. Finally we set $f(t_1, s_2) = f(t_2, s_1) = 1$ and set f to 0 for all other edges. We note that all capacities have been respected and all demands have been met. The resulting flow is connected, since the paths were connected and $f(t_1, s_2) = 1$.

For the other direction, let f be a connected flow for $(G', D, \mathsf{dem}, \mathsf{cost}, \mathsf{cap})$. Since $\mathsf{dem}(s_1) = \mathsf{dem}(s_2) = 1$ and s_1 and s_2 only have one incoming edge, we have that $f(t_1, s_2) = f(t_2, s_1) = 1$. We argue that $G_f - \{(t_1, s_2), (t_2, s_1)\}$ consists of two vertex disjoint paths in G', one from s_1 to t_1 and the other from s_2 to t_2. First we note that for every vertex in G', it has in-degree 1 or out-degree 1 (or possibly both).

This means that since we have $\mathsf{cap}(u,v) = 1$ for every $(u,v) \in E(G')$, every vertex in $V(G_f)$ has in- and out-degree 1 in G_f. Since G_f is connected we find that G_f is a single cycle and thus $G_f - \{(t_1,s_2),(t_2,s_1)\}$ is the union of two vertex-disjoint paths. We now find two vertex-disjoint paths in G by contracting the edges $(v_{\mathsf{in}}, v_{\mathsf{out}})$ in $G_f - \{(t_1,s_2),(t_2,s_1)\}$. ■

Lemma 6.7. *Given an instance* $(G, D, \mathsf{dem}, \mathsf{cost}, \mathsf{cap})$ *of* Connected Flow *where* $\mathsf{cap}(e) = \infty$ *for all* $e \in E$*, we can construct an equivalent instance of* MVTSP *on* $|D|$ *vertices.*

Proof. We construct an equivalent instance $(G', \mathsf{dem}, \mathsf{cost}')$ of MVTSP as follows. First we let $V(G') = D$ and for $u, v \in D$ we let $(u,v) \in E(G')$ if and only if there is a $u - v$ path in G, disjoint from other vertices in D. We then set $\mathsf{cost}(u,v)$ to be the total cost of the shortest such path. We keep $\mathsf{dem}(v)$ the same.

We now show equivalence of the two instances. Let $s' : E(G') \to \mathbb{N}$ be a valid tour on $(G', \mathsf{dem}, \mathsf{cost}')$. We construct a connected flow f on $(G, D, \mathsf{dem}, \mathsf{cost}, \mathsf{cap})$ as follows: For each $(u,v) \in E(G')$, add $s'(u,v)$ copies of the shortest D-disjoint u-v-path in G to the flow. Note that the demands are met, since the demands in both instances are the same. Also note that by definition the total cost of $s'(u,v)$ copies of the shortest D-disjoint $u - v$ path is equal to $s'(u,v) \cdot \mathsf{cost}'(u,v)$ and thus the total cost of f is equal to that of s'. Finally, we note that the capacity constraints are trivially met.

For the other direction, let $f : E(G) \to \mathbb{N}$ be an optimal connected flow on $(G, D, \mathsf{dem}, \mathsf{cost}, \mathsf{cap})$. Note that G_f is connected and that every vertex in this multigraph has equal in- and out-degrees. This means we can find an Eulerian tour on G_f. We now construct an MVTSP tour s' on G' by adding the edge (u,v) every time v is the first vertex with demand to appear after an appearance of u in the Eulerian tour. Again it is easy to see that s' is connected and that the demands are met. We claim that the total cost of s' is the same as f. Indeed, if it were larger, then there would be a pair $u, v \in D$ such that the cost of some path in the Eulerian tour from u to v is less than $\mathsf{cost}'(u,v)$, which contradicts the definition of cost'. If it were smaller, then there would be a D-disjoint path in the Eulerian tour from some u to some v which is longer than $\mathsf{cost}'(u,v)$. We can then find a cheaper flow by replacing this path with the shortest path, contradicting the optimality of f. ■

Since MVTSP can be solved in $O^\star(4^n)$ time by Kowalik et al. [85], we get as a direct consequence:

Theorem 6.2. *Any instance* $(G, D, \mathsf{dem}, \mathsf{cost}, \mathsf{cap})$ *of* Connected Flow *where* $\mathsf{cap}(e) = \infty$ *for all* $e \in E$ *can be solved in time* $O^\star(4^{|D|})$.

6.4 Parameterization by Vertex Cover

In this section, we consider Connected Flow and MVTSP, parameterized by the cardinality k of a vertex cover of the input graph. We first extend a lemma from Kowalik et al. [85] to instances of Connected Flow. Then we use this lemma to obtain an FPT algorithm for Connected Flow and a polynomial-sized kernel for MVTSP.

6.4.1 Enforcing edges in flow relaxation

Let s be an optimal solution of Connected Flow and let $T \subseteq \text{supp}(s)$. We prove that, given any optimal solution r for Demand Flow, there is always a flow f that is "close" to r and $T \subseteq \text{supp}(f)$. Furthermore, it has cost $\mathsf{cost}(f) \leqslant \mathsf{cost}(s)$. Note that if T connects all demand vertices to each other, this implies that f is connected and thus an optimal solution of Connected Flow.

The basic idea and arguments are from Kowalik et al. [85], where a similar theorem for MVTSP was proved. We adjust their proof to the case with capacities and where not all vertices have a demand. Furthermore, we note that we can restrict the tours $C \in \mathcal{C}$ in the proof to be inclusion-wise minimal, which allows us to conclude a stronger inequality.

Lemma 6.8. *Let $(G, D, \mathsf{dem}, \mathsf{cost}, \mathsf{cap})$ be an instance of* Connected Flow*, where $G = (V, E)$. Let s be an optimal solution of* Connected Flow *and let $T \subseteq \text{supp}(s)$. For every optimal solution r of* Demand Flow *for the same instance, there is a flow f with $\mathsf{cost}(f) \leqslant \mathsf{cost}(s)$, with $f(e) > 0$ for all $e \in T$ and such that for every $v \in V$:*

$$\sum_{u \in V} |r(u,v) - f(u,v)| \leqslant 2|T|, \qquad \text{and} \qquad \sum_{u \in V} |r(v,u) - f(v,u)| \leqslant 2|T|.$$

Proof. We follow the structure of the proof of Lemma 3.2 from Kowalik et al. [85]. We build a flow f (not necessarily optimal for Demand Flow), containing T and with multiplicities close to r. Recall that m_B denotes the multiplicity function of the multiset B. We define the multisets of edges A_s, A_r and A such that for all $e \in E$:

- $m_{A_s}(e) = \max\{s(e) - r(e), 0\}$,
- $m_{A_r}(e) = \max\{r(e) - s(e), 0\}$, and
- $m_A(e) = \max\{m_{A_r}(e), m_{A_s}(e)\} = \max\{s(e) - r(e), r(e) - s(e)\}$.

Note that A is the symmetric difference of s and r, and therefore A is equal to the disjoint union of A_r and A_s.

Let H be a tour (i.e. a closed walk) of undirected edges. We then say that $\overrightarrow{H}$ is a *cyclic orientation* of H if it is an orientation of the edges in H such that $\overrightarrow{H}$ forms a directed tour. A directed edge e that overlaps with H is in *positive orientation* with respect to $\overrightarrow{H}$ if it has the same orientation, and negative otherwise. We now define $(s - r)$ directed tours (for an example, see Figure 6.1).

Definition 6.9. *Let $C = (e_0, \ldots, e_\ell) \subseteq A$ be a set of edges such that its underlying undirected edge set H is a tour. We then say that C is an* $(s - r)$ directed tour *if there is an orientation $\overrightarrow{H}$ of H such that:*

- *if $e \in C$ is in positive orientation with respect to $\overrightarrow{H}$, then $e \in A_s$,*
- *if $e \in C$ is in negative orientation with respect to $\overrightarrow{H}$, then $e \in A_r$,*
- *if two subsequent edges e_i, e_{i+1} of C have the same orientation, then their shared vertex, v, is not in D. This also holds for the edge pair (e_ℓ, e_0).*

We will partition A into a multiset of $(s - r)$ directed tours. We take $(u, v) \in A$ arbitrarily as our first edge of our walk and iteratively add edges until we find an $(s - r)$ directed tour. We assume the current edge (u, v) is in A_s (if the edge is in A_r,

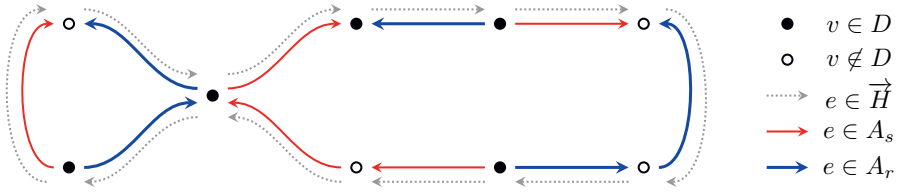


Figure 6.1: Example of an $(s-r)$ directed tour. Note that every time the tour visits a vertex $v \in D$, the orientation changes. However if the tour visits a vertex $v \notin D$, the orientation might not change.

the proof is analogous). If $v \in D$, then there exists $(v,w) \in A_r$, because v is visited $\mathsf{dem}(v)$ times by both r and s. If $v \notin D$, there exists either $(v,w) \in A_r$ or $(w,v) \in A_s$ because A is the symmetric difference of the flows r and s. We take this edge as the next edge in our $(s-r)$ directed tour. This way we can keep finding the next edges, until we can take our first edge (u,v) as our next edge and we find an $(s-r)$ directed tour. We then remove this tour and inductively find the next until A is empty.

It follows that A can be partitioned into a multiset $\mathcal{C}$ of $(s-r)$ directed tours, i.e.

$$m_A = \sum_{C \in \mathcal{C}} m_C,$$

where m_C is the multiplicity of $(s-r)$ directed tour C.

We may assume that these $(s-r)$ directed tours are inclusion-wise minimal, i.e. for each $(s-r)$ directed tour $C \in \mathcal{C}$, no subset $C' \subset C$ is an $(s-r)$ directed tour. Otherwise, C can be split into two disjoint $(s-r)$ directed tours C' and $C \setminus C'$.

Claim 6.1. For any $v \in V$ and any inclusion-wise minimal $C \in \mathcal{C}$ we have

$$\sum_{u \in V} [(u,v) \in C] \leqslant 2 \qquad \text{and} \qquad \sum_{u \in V} [(v,u) \in C] \leqslant 2. \tag{6.1}$$

Proof of Claim: We only prove the first inequality, as the second one is analogous. Assume for contradiction that there exists $C \in \mathcal{C}$ and $v \in V$ such that there exist $x_1, x_2, x_3 \in V$ with $(x_i, v) \in C$ for $i = 1,2,3$. Each of these edges must be either in A_s or A_r. Without loss of generality we may assume that $(x_1,v),(x_2,v) \in A_s$. Both (x_1,v) and (x_2,v) can be paired with the edge it traverses v with, i.e. its subsequent edge in the tour, as $(x_1,v),(x_2,v) \in A_s$ are positively oriented. Let e_1, e_2 be these subsequent edges. Then C can be split into two smaller $(s-r)$ directed tours C_1 and C_2, with C_1 starting with edge e_1 and ending with (x_2,v), and C_2 starting with edge e_2 and ending with (x_1,v). This contradicts the assumption that C was inclusion-wise minimal. □

We denote with $T^+ = E(T) \setminus \mathrm{supp}(r)$ the set of edges of T that are not yet covered by r. Hence, if $e \in T^+$, then $e \in A_s$ and there is at least one $C \in \mathcal{C}$ that contains e. We choose for each $e \in T^+$ such an $(s-r)$ directed tour $C_e \in \mathcal{C}$ arbitrarily. Let $\mathcal{C}^+ = \{C_e : e \in T^+\}$ be the set of chosen $(s-r)$ directed tours. We define f as follows:

for each $u, v \in V$ we set

$$f(u,v) = r(u,v) + (-1)^{[(u,v)\in A_r]} \sum_{C\in\mathcal{C}^+} [(u,v) \in C]. \tag{6.2}$$

In other words, f is obtained from r by removing one copy of edges in $C \cap A_r$ and adding one copy of edges in $C \cap A_s$ for all $C \in \mathcal{C}^+$.

Notice that $|\mathcal{C}^+| \leqslant |T^+| \leqslant |T|$. By using (6.2) and subsequently (6.1), we get for all $v \in V$:

$$\begin{aligned}\sum_{u\in V} |r(u,v) - f(u,v)| &\leqslant \sum_{u\in V}\sum_{C\in\mathcal{C}^+} [(u,v) \in C] \\ &= \sum_{C\in\mathcal{C}^+}\sum_{u\in V} [(u,v) \in C] \\ &\leqslant \sum_{C\in\mathcal{C}^+} 2 \leqslant 2|T|.\end{aligned}$$

Similarly, we can conclude that for all $v \in V$, $\sum_{u\in V} |r(v,u) - f(v,u)| \leqslant 2|T|$.

Claim 6.2. For all $e \in T$, $f(e) > 0$ and f satisfies the flow constraints.

Proof of Claim: We first show that for all $e \in E$,

$$\min\{r(e), s(e)\} \leqslant f(e) \leqslant \max\{r(e), s(e)\}.$$

If $e \notin A$, Equation (6.2) implies that $r(e) = f(e) = s(e)$. If $e \in A_s$, we have $r(e) < s(e)$ and we can see from (6.2) that if the multiplicity of e changes, it is because copies of e are added to r to form f (and none are removed). Since $m_A(e) \leqslant s(e) - r(e)$, at most this many copies of e can be added to r to form f. Hence $r(e) \leqslant f(e) \leqslant r(e) + (s(e) - r(e)) = s(e)$. Similarly, if $e \in A_r$, $r(e) > s(e)$ and at most $r(e) - s(e)$ copies of e are removed from r to form f.

Next we prove that f is a valid solution to the given Demand Flow instance. Let $C \in \mathcal{C}^+$ and let e, e' be two subsequent edges from C with common vertex v. If $e \in A_s$ and $e' \in A_r$, then the in- and out-degrees of v do not change while adding a copy of e and removing a copy of e' (in Equation (6.2)), because the orientation of e and e' is different. This is also true if $e \in A_r$ and $e' \in A_s$. If $e, e' \in A_r$, then both e and e' have a copy removed in Equation (6.2). Since the orientation of e and e' is the same, both the in- and out-degree of v go down by one. We remark that this case only occurs if $v \notin D$ by definition of $(s - r)$ directed tours. Similarly, if $e, e' \in A_s$, the in- and out-degree of v increases by one. Since r was a valid solution, this implies that the number of incoming- and outgoing edges of v in f are equal, in other words, the flow is preserved. Since for $v \in D$, the total incoming (and total outgoing) edges do not change, the demands are satisfied by f. Furthermore, the capacity constraints are satisfied since $f(e) \leqslant \max\{r(e), s(e)\} \leqslant \mathsf{cap}(e)$.

Finally, we need to show that $T \subseteq \mathrm{supp}(f)$. For $e \in T^+$, we have $e \in A_s$ so copies of e are added to r to form f in Equation (6.2). Since $e \in T^+$, at least one tour $C \in \mathcal{C}^+$ contains e, so $f(e) > 0$. If $e \in T \setminus T^+$, then $r(e) > 0$ since $T^+ = T \setminus \mathrm{supp}(r)$. We also see that $s(e) > 0$ because $T \subseteq \mathrm{supp}(s)$ by assumption. Using our earlier

result that $\min\{r(e), s(e)\} \leqslant f(e)$, we conclude that $f(e) > 0$. □

It remains to prove that $\mathsf{cost}(f) \leqslant \mathsf{cost}(s)$. For any $C \in \mathcal{C}$, define $\delta(C) = \mathsf{cost}(A_s \cap C) - \mathsf{cost}(A_r \cap C)$ as the cost of adding all edges in $A_s \cap C$ and removing all edges in $A_r \cap C$. Notice that $\delta(C) \geqslant 0$ for all tours $C \in \mathcal{C}$, as otherwise r would not have been optimal since we could improve it by augmenting along C. We note that $\sum_{C \in \mathcal{C}^+} \delta(C) \leqslant \sum_{C \in \mathcal{C}} \delta(C)$ as $\mathcal{C}^+ \subseteq \mathcal{C}$. Therefore,

$$\mathsf{cost}(f) = \mathsf{cost}(r) + \sum_{C \in \mathcal{C}^+} \delta(C) \leqslant \mathsf{cost}(r) + \sum_{C \in \mathcal{C}} \delta(C) = \mathsf{cost}(s).$$ ■

6.4.2 FPT algorithm

Now we use Lemma 6.8 to show that Connected Flow is FPT parameterized by the size of vertex cover of G:

Theorem 6.3. *There is an algorithm solving a given instance* $(G, D, \mathsf{dem}, \mathsf{cost}, \mathsf{cap})$ *of* Connected Flow *such that G has a vertex cover of size k in time* $O^\star(k^{O(k)})$.

Proof. Let X be a vertex cover of size k of $G = (V, E)$, let s be an arbitrary optimal solution of Connected Flow and let $X' \subseteq X$ be the set of vertices of X that are visited at least once by s. We will guess this set X' as part of our algorithm, i.e. go through all possible sets. Hence we do the following algorithm for all X' such that $D \cap X \subseteq X' \subseteq X$, which is at most 2^k times.

For any X', we modify G such that the vertex cover is an independent set and all $x \in X'$ are visited at least once in any solution as follows. We remove any edge $(x_i, x_j) \in E$ for $x_i, x_j \in X'$ and add a new vertex y to V. The vertex y has no demand and has edges (x_i, y) and (y, x_j), with capacities equal to the old capacity $\mathsf{cap}(x_i, x_j)$ and $\mathsf{cost}(x_i, y) = \mathsf{cost}(x_i, x_j)$ and $\mathsf{cost}(y, x_j) = 0$. This removes any edges between vertices in the set X', making it an independent set. We note that X is still a vertex cover of size k.

The vertices $x \in X' \cap D$ are visited at least once because of their demand. For all $x \in X' \setminus D$ we add a vertex b_x to V, with $\mathsf{dem}(b_x) = 1$ and we add edges (x, b_x) and (b_x, x), both with 0 cost and a capacity of 1. As b_x has a demand of 1 and has only x as its neighbor, this ensures that x is visited at least once.

We remove all $x \in X \setminus X'$ from V and denote the resulting graph with $G' = (V', E')$. Note that if X' is guessed correctly, the optimal solution s of the original instance, is also optimal for this newly created instance (by adding flow over the newly created edges between x and b_x, and replacing any edge (x_i, x_j) by the edges (x_i, y) and (y, x_j)).

We use dynamic programming to construct the solution f. Namely, we iteratively add vertices from the independent set $B = V' \setminus X'$ and keep track of the connectedness of our vertex cover X' with a partition π. We will later use Lemma 6.8 to reduce the number of table entries we need to compute.

Denote $X' = \{x_1, \ldots, x_{k'}\}$ and $B = \{b_1, \ldots, b_n\}$. For $j \in [0, n]$ let B_j be the set of the first j vertices of B, i.e. $B_j = \{b_1, \ldots, b_j\}$ and define $V_j = X' \cup B_j$. For any

$f : (V_j)^2 \to \mathbb{N}$ and $v \in V_j$ define

$$f^{\mathsf{out}}(v) = \sum_{u \in V_j} f(v,u) \quad \text{and} \quad f^{\mathsf{in}}(v) = \sum_{u \in V_j} f(u,v).$$

Let $\mathbf{c}^{\mathsf{in}} = (c_1^{\mathsf{in}}, \ldots, c_{k'}^{\mathsf{in}}) \in \mathbb{N}^{k'}$ and $\mathbf{c}^{\mathsf{out}} = (c_1^{\mathsf{out}}, \ldots, c_{k'}^{\mathsf{out}}) \in \mathbb{N}^{k'}$ be two vectors of integers and let π be a partition of the vertices of X'.

For $j \in [0,n]$ we define the dynamic programming table entry $T_j(\pi, \mathbf{c}^{\mathsf{in}}, \mathbf{c}^{\mathsf{out}})$ to be equal to the minimal cost of any partial solution $f : (V_j)^2 \to \mathbb{N}$ having the specified in and out degrees ($\mathbf{c}^{\mathsf{in}}$ and $\mathbf{c}^{\mathsf{out}}$) for vertices in X' and connecting all vertices $x \in S$ for each $S \in \pi$. More formally, $T_j(\pi, \mathbf{c}^{\mathsf{in}}, \mathbf{c}^{\mathsf{out}})$ is equal to $\min_f \mathsf{cost}(f)$ over all $f : (V_j)^2 \to \mathbb{N}$ such that the following conditions hold:

1. for all blocks S of the partition π, S is weakly connected in G'_f,
2. for all $x_i \in X'$: $f^{\mathsf{out}}(x_i) = c_i^{\mathsf{out}}$, $f^{\mathsf{in}}(x_i) = c_i^{\mathsf{in}}$,
3. for all $v \in B_j$: $f^{\mathsf{out}}(v) = f^{\mathsf{in}}(v)$, and if $v \in B_j \cap D$, then $f^{\mathsf{in}}(v) = \mathsf{dem}(v)$,
4. for all $u, v \in V_j : f(u,v) \leqslant \mathsf{cap}(u,v)$.

We set $T_j(\pi, \mathbf{c}^{\mathsf{in}}, \mathbf{c}^{\mathsf{out}}) = \infty$ if no such f exists.

Claim 6.3. Each table entry $T_j(\pi, \mathbf{c}^{\mathsf{in}}, \mathbf{c}^{\mathsf{out}})$ can be computed from all table entries T_{j-1}.

Proof of Claim: Compute table entries for $j = 0$ as follows. Set $T_0(\{\{x_1,\}, \ldots, \{x_{k'}\}\}, \mathbf{0}, \mathbf{0})$ to 0, and all other entries of T_0 to ∞, as $V_0 = X'$ is an independent set and so a flow of zero on every edge is the only possible flow.

Now assume $j > 0$. Informally, in order to compute $T_j(\pi, \mathbf{c}^{\mathsf{in}}, \mathbf{c}^{\mathsf{out}})$ we consider all possible multiplicities of edges (b_j, x_i) and (x_i, b_j) (h_i^{in} and h_i^{out} respectively), and partitions π' of X', such that each block in π either belongs to π', or consists of several blocks of π' that are connected to b_j. Formally, we have

$$T_j(\pi, \mathbf{c}^{\mathsf{in}}, \mathbf{c}^{\mathsf{out}}) = \min_{(*)} \{T_{j-1}(\pi', \mathbf{c}^{\mathsf{in}} - \mathbf{h}^{\mathsf{in}}, \mathbf{c}^{\mathsf{out}} - \mathbf{h}^{\mathsf{out}})$$

$$+ \sum_{i=1}^{k'} \left(h_i^{\mathsf{in}} \cdot \mathsf{cost}(b_j, x_i) + h_i^{\mathsf{out}} \cdot \mathsf{cost}(x_i, b_j) \right) \},$$

where the minimization $(*)$ goes over all $\mathbf{h}^{\mathsf{in}} = (h_1^{\mathsf{in}}, \ldots, h_{k'}^{\mathsf{in}}) \in \mathbb{N}^{k'}$, $\mathbf{h}^{\mathsf{out}} = (h_1^{\mathsf{out}}, \ldots, h_{k'}^{\mathsf{out}}) \in \mathbb{N}^{k'}$ and partitions π' of X' that satisfy the following conditions:

(i) $h_i^{\mathsf{in}} \leqslant \mathsf{cap}(b_j, x_i)$ for all $i = 1, \ldots, k'$

(ii) $h_i^{\mathsf{out}} \leqslant \mathsf{cap}(x_i, b_j)$ for all $i = 1, \ldots, k'$

(iii) $\sum_{i=1}^{k'} h_i^{\mathsf{in}} = \sum_{i=1}^{k'} h_i^{\mathsf{out}}$ and if $b_j \in D$, $\sum_{i=1}^{k'} h_i^{\mathsf{in}} = \mathsf{dem}(b_j)$

(iv) For all $S \in \pi$, we have $S \in \pi'$ or there exist $S'_1, \ldots, S'_\ell \in \pi'$ such that $S'_1 \cup \cdots \cup S'_\ell = S$ and for each $t \in [\ell]$, there exists $x_i \in S'_t$ such that $h_i^{\mathsf{in}} + h_i^{\mathsf{out}} > 0$.

Conditions (i) and (ii) correspond to capacity constraints. Condition (iii) ensures that flow preservation holds for b_j and possibly that the demand of b_j is met. Condition (iv) states that the partition π can be obtained from π' by merging blocks that are connected to b_j.

Notice that with this recurrence, the table entries are computed correctly as only the vertex b_j was added compared to the table entries T_{j-1}. Therefore we may assume that only the edges incident to b_j were added to the solution for some table entry in T_{j-1}. □

Let us now show how to bound the number of table entries we need to compute. Let s be a solution of Connected Flow on the input graph G such that $X' \subseteq X$ is the set of vertices in X that are visited at least once by s. We compute a *relaxed* solution r for the instance G', which can be done in polynomial time. We claim that there exists a directed tree T of size at most $2k$ such that $T \subseteq \mathrm{supp}(s)$ and all $x \in X'$ are incident to at least one edge $e \in T$.

Indeed, since s is connected and visits all $x \in X'$, we can find a tree $T \subseteq \mathrm{supp}(s)$ that contains all $x \in X'$. If $|T| > 2k$, we remove all the leaves from T not in X'. Since $V' \setminus X'$ is an independent set (as X' is a vertex cover), this means that the size of T is bounded by $2k$. Note that all $x \in X'$ are still incident to an edge $e \in T$, which proves the claim.

Applying Lemma 6.8 to s and T, we conclude that there exists a flow f such that $\mathsf{cost}(f) \leqslant \mathsf{cost}(s)$ and for every $v \in V$,

$$\sum_{u \in V'} |r(u,v) - f(u,v)| \leqslant 4k, \qquad \text{and} \qquad \sum_{u \in V'} |r(v,u) - f(v,u)| \leqslant 4k. \tag{6.3}$$

Since $T \subseteq \mathrm{supp}(f)$, f visits all the vertices in X' at least once. As X' is a vertex cover, this means that f is a connected flow and hence an optimal solution of the instance of Connected Flow.

We restrict our dynamic program using Equation (6.3). As X' is an independent set, there are only edges between $x \in X'$ and $b \in B_j$. Therefore, there exists a solution f such that for every $x \in X'$ and $j \in [0, n]$:

$$\sum_{b \in B_j} |r(b,x) - f(b,x)| \leqslant 4k, \qquad \text{and} \qquad \sum_{b \in B_j} |r(x,b) - f(x,b)| \leqslant 4k \tag{6.4}$$

We only compute table entries T_j respecting Equation (6.4), by requiring that for all $i \in [1, k']$:

$$\begin{aligned} c_i^{\mathsf{out}} &\in \left[\sum_{b \in B_j} r(x_i, b) - 4k, \sum_{b \in B_j} r(x_i, b) + 4k \right], \text{ and} \\ c_i^{\mathsf{in}} &\in \left[\sum_{b \in B_j} r(b, x_i) - 4k, \sum_{b \in B_j} r(b, x_i) + 4k \right]. \end{aligned} \tag{6.5}$$

Note that the dynamic program is still correct with this added restriction, as $\sum_{b \in B_{j-1}} |r(b,x) - f(b,x)| \leqslant \sum_{b \in B_j} |r(b,x) - f(b,x)| \leqslant 4k$, so any table entry T_j

respecting Equation (6.5) can be computed from all table entries T_{j-1} respecting Equation (6.5).

The dynamic program returns the minimum value of $T_n(\{X'\}, \mathbf{c}, \mathbf{c})$ for all $\mathbf{c}$ such that $c_i = \mathsf{dem}(x_i)$ for all $x_i \in D \cap X'$. This returns the value of a minimum cost solution f for G', respecting Equation (6.4), if one exists. Let $f_{X'}$ be solution the dynamic program found in the iteration using X'. Then $\min\{f_{X'} : (D \cap X) \subseteq X' \subseteq X\}$ is equal to the minimum cost connected flow.

We count the number of different table entries T_j computed by the dynamic program for fixed j. There are at most $(8k)^k$ possible values for both $\mathbf{c}^{\mathsf{in}}$ and $\mathbf{c}^{\mathsf{out}}$ and at most k^k different partitions π of X', so a total of $k^k \cdot (8k)^{2k}$ different entries. To compute one table entry of T_j, we only need table entries of T_{j-1}. Note that we compute this dynamic programming table for each X' such that $(D \cap X) \subseteq X' \subseteq X$, that is at most 2^k different X'. Hence the algorithm runs in time $O^\star(k^{O(k)})$. ■

6.4.3 Polynomial Kernel for MVTSP

We now present how to find a kernel with $O(k^5)$ vertices for any instance of MVTSP, where k is the size of a vertex cover of G. We do this by first finding an optimal solution r to the relaxed Flow problem and then fixing the amount of flow on some edges based on this r. We prove that there is an optimal solution s of MVTSP such that for all except $O(k^5)$ vertices, all edges incident to these vertices have exactly the same flow in r and s, as a consequence of Lemma 6.8.

Theorem 6.4. MVTSP *admits a kernel polynomial in the size k of the vertex cover of G.*

Proof. Fix an input instance on MVTSP. Let k be the number of vertices in the vertex cover $X = \{x_1, \ldots, x_k\}$ of G and let n be the size of the independent set $B = V \setminus X$. Let r be an optimal solution of the instance of Flow obtained by relaxing the connectivity constraint in the given instance of MVTSP.

Define multisets $\overrightarrow{F} = (X \times B) \cap r$ (i.e. all edges in r going from vertices in X to vertices in B) and $\overleftarrow{F} = (B \times X) \cap r$.

Claim 6.4. We may assume that for both $\overrightarrow{F}$ and $\overleftarrow{F}$, their underlying undirected edge sets do not contain cycles.

Proof of Claim: We modify r such that for both $\overrightarrow{F}$ and $\overleftarrow{F}$, their underlying undirected edge sets do not contain cycles. Assume that there is an *alternating* cycle $C \subseteq \overleftarrow{F}$, meaning that its underlying edge set is a cycle and (hence) the edges alternate between being in positive and negative orientation. We can then create solutions r' and r'' of Flow by alternatingly adding and removing edges from C. Note that we can start by either adding or removing, giving us these two different solutions r' and r''. Since the edges added to r to form r' are exactly the edges that were removed from r to form r'', and vice versa, it holds that $\mathsf{cost}(r) - \mathsf{cost}(r') = -(\mathsf{cost}(r) - \mathsf{cost}(r''))$. Since r is an optimal solution, we conclude $\mathsf{cost}(r) = \mathsf{cost}(r') = \mathsf{cost}(r'')$. We can therefore choose either r' or r'' to replace r, such that $\overleftarrow{F}$ now has one alternating cycle less without changing any of the edges of r outside C. Hence we can iteratively remove the cycles from $\overleftarrow{F}$ and $\overrightarrow{F}$ and obtain an optimal solution r to the Flow

instance in which both $\overleftarrow{F}$ and $\overrightarrow{F}$ are forests in polynomial time. $\square$

We partition B as follows: $B = Y \cup \left(\bigcup_{i,j\in[1,k]} B_{ij}\right)$, where for each $b \in B_{ij}$: $r(x_i, b) > 0$, $r(b, x_j) > 0$, and

$$r(x_a, b) = 0 \text{ for all } a \neq i \qquad \text{and} \qquad r(b, x_a) = 0 \text{ for all } a \neq j,$$

and $Y = B \setminus \left(\bigcup_{i,j\in[1,k]} B_{ij}\right)$.

We claim that $|Y| \leqslant k$. Recall that m_B denotes the multiplicity function of a multiset B. Let $F = \mathrm{supp}(m_{\overleftarrow{F}}) \cup \mathrm{supp}(m_{\overrightarrow{F}})$ (note that F is a set and not a multiset). Then $|F| \geqslant \left(\sum_{i,j\in[1,k]} 2|B_{ij}|\right) + 3|Y| = 2n + |Y|$, as any vertex $v \in B_{ij}$ corresponds to exactly 2 edges in F and each vertex in Y corresponds to at least 3 edges to F. Here we use the fact that each vertex has a demand and therefore must have at least one incoming and outgoing edge from r. As F is a union of two forests on $n + k$ vertices, we see that $|F| \leqslant 2(n + k - 1)$. We conclude that $2(n + |Y|) \leqslant 2(n + k - 1)$, i.e. $|Y| \leqslant k$.

Let s be an optimal solution of the MVTSP instance. By definition, s visits every vertex at least once. Using arguments similar to those in the proof of Theorem 6.3, we conclude that there exists a directed tree $T \subseteq \mathrm{supp}(s)$ such that it covers all vertices of X and has size at most $2k$. We apply Lemma 6.8 to s and T, to find that there exists an optimal solution f to the given MVTSP instance such that

$$\sum_{v\in V} \left(|r(x_i, v) - f(x_i, v)| + |r(v, x_i) - f(v, x_i)|\right) \leqslant 8k \qquad \forall i \in [1, k]. \tag{6.6}$$

We note that G_f is connected because $T \subseteq \mathrm{supp}(f)$ and T connects all the vertices of the vertex cover. Equation (6.6) implies that there is an optimal solution f that differs from $\overleftarrow{F}$ and $\overrightarrow{F}$ by at most $8k^2$ edges.

For every $i, j, \ell \in [1, k]$, we define $\overrightarrow{A_{ij}}(\ell)$ as the set of $8k^2 + 2$ vertices $v \in B_{ij}$ with the smallest values of $\mathsf{cost}(x_\ell, v) - \mathsf{cost}(x_i, v)$ (arbitrarily breaking ties if needed). If $|B_{ij}| \leqslant 8k^2 + 2$, we let $\overrightarrow{A_{ij}}(\ell) = B_{ij}$. Intuitively, the vertices in $\overrightarrow{A_{ij}}(\ell)$ are the vertices for which re-routing the flow sent from x_i to v to go from x_ℓ to v is the least expensive. Similarly we define $\overleftarrow{A_{ij}}(\ell)$ as a set of size $8k^2 + 2$ containing vertices $v \in B_{ij}$ with the smallest values of $\mathsf{cost}(v, x_\ell) - \mathsf{cost}(v, x_j)$.

We also define a set R_{ij} of "remainder vertices" as follows:

$$R_{ij} = B_{ij} \setminus \left(\left(\bigcup_{\ell\in[1,k]} \overleftarrow{A_{ij}}(\ell)\right) \cup \left(\bigcup_{\ell\in[1,k]} \overrightarrow{A_{ij}}(\ell)\right)\right) \text{ for all } i, j \in [1, k].$$

Claim 6.5. There exists an optimal solution f' of the MVTSP instance such that for all $i, j \in [1, k]$, $b \in R_{ij}$ and $x_\ell \in X$ it holds that $r(x_\ell, b) = f'(x_\ell, b)$ and $r(b, x_\ell) = f'(b, x_\ell)$.

Proof of Claim: We build f' iteratively from f, by removing any edges $(x_{i'}, b)$ and $(b, x_{j'})$ for $i' \neq i$ and $j' \neq j$ for each $b \in R_{ij}$. In particular, this implies that

$r(x_i, b) = f'(x_i, b)$ and $r(b, x_j) = f'(b, x_j)$, as b then only has edges coming from x_i and to x_j and since b has a fixed demand.

Note that we retain optimality and connectivity for f'. Furthermore, after each step, the solutions r and f' differ by at most $8k^2$ edges. We start by setting $f' = f$.

Consider $b \in R_{ij}$ and suppose that $f'(x_\ell, b) > 0$ for some $\ell \neq i$ (the case where $f'(b, x_\ell) > 0$ for some $\ell \neq j$ is analogous). We remark that $|\overrightarrow{A_{ij}}(\ell)| = 8k^2 + 2$ as $R_{ij} \neq \emptyset$. As at most $8k^2$ edges are different between r and f', there are vertices $v, w \in \overrightarrow{A_{ij}}(\ell)$ such that all of the edges adjacent to v and w have the same multiplicities in r and f', i.e. $f'(x, v) = r(x, v)$ and $f'(x, w) = r(x, w)$ for all $x \in X$.

Define flow f'' by removing one copy of the edges (x_ℓ, b) and (x_i, v) from f' and adding one copy of the edges (x_i, b) and (x_ℓ, v) (see Figure 6.2). As $b \notin \overrightarrow{A_{ij}}(\ell)$ and $v \in \overrightarrow{A_{ij}}(\ell)$, the cost of f'' is at most the cost of f' by definition of the set $\overrightarrow{A_{ij}}(\ell)$.

We now argue that f'' is connected. As f' is a solution to MVTSP, it must be connected. Since we removed (x_ℓ, b) and (x_i, v) from f' to form f'', proving that the pairs x_ℓ, b and x_i, v are connected in f'' suffices. The edges (x_i, w), (w, x_j) and (v, x_j) in f'' connect x_i and v. As a consequence, x_ℓ and b are also connected by edges (x_ℓ, v) and (x_i, b).

We remark that the number of edges that differ between f'' and r has not changed. Hence, we continue with setting $f' = f''$ and repeating until f' has the required properties. □

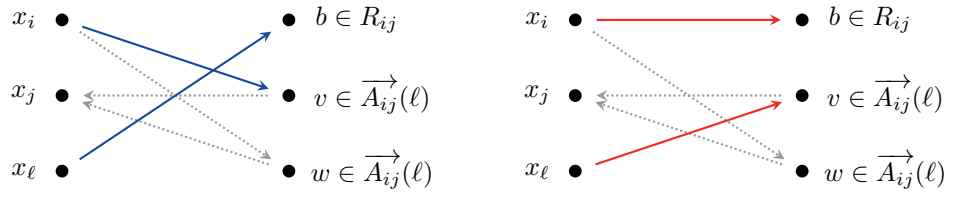


Figure 6.2: Adjusting flow f', depicted on the left, to get flow f'', depicted on the right. The blue edges are replaced by the red edges, the rest of the solutions are equal. The vertex w assures the new solution remains connected

Therefore, we may assume that in $G_{f'}$ the vertices in R_{ij} are adjacent only to x_i and x_j for all $i, j \in [1, k]$. This proves that the following reduction rule is correct: contract all vertices in R_{ij} into one vertex r_{ij} with edges only (x_i, r_{ij}) and (r_{ij}, x_j) of cost zero and let the demand $\mathsf{dem}(r_{ij}) = \sum_{v \in R_{ij}} \mathsf{dem}(v)$. Hence, we require any solution to use the vertices in r_{ij} exactly the number of times that we would traverse all the vertices of R_{ij}. By applying this rule, we get a kernel with the vertices from the sets X, Y, $\overleftarrow{A_{ij}}(\ell)$, $\overrightarrow{A_{ij}}(\ell)$, and r_{ij}, which is of size

$$|X| + |Y| + \sum_{i,j,\ell \in [1,k]} \left(\left|\overleftarrow{A_{ij}}(\ell)\right| + \left|\overrightarrow{A_{ij}}(\ell)\right| \right) + k^2 \leqslant k + k + k^3 \cdot (8k^2 + 2) + k^2 = O(k^5).$$

To subsequently reduce all costs to be at most $2^{k^{O(1)}}$ we can use a method from Etscheid et al. [59] in a standard manner.

We show that one can construct this kernel in polynomial time. First, compute a relaxed solution r and remove any cycles in $\overrightarrow{F}$ and $\overleftarrow{F}$ in polynomial time. Next for each $i, j, \ell \in [1, k]$, compute in polynomial time the sets $\overrightarrow{A_{ij}}(\ell)$ and $\overleftarrow{A_{ij}}(\ell)$, by computing the values of $\mathsf{cost}(x_\ell, v) - \mathsf{cost}(x_i, v)$ and sorting. Finally, we can contract all vertices in R_{ij} into a vertex r_{ij} polynomial time for all $i, j \in [1, k]$. ■

6.5 Parameterization by Treewidth

In this section we consider the complexity of Connected Flow when parameterized by the treewidth tw of G. We first give a $|V(G)|^{O(tw)}$ time dynamic programming algorithm for Connected Flow. Subsequently, we give a matching conditional lower bound on the complexity of MVTSP parameterized by the pathwidth of G. Since MVTSP is a special case of Connected Flow, this shows that our dynamic programming algorithm is in some sense optimal.

6.5.1 XP algorithm for Connected Flow

In this subsection we show the following:

Theorem 6.5. *Let M be an upper bound on the demands in the input graph G, and suppose a tree decomposition of width tw of G is given. Then a* Connected Flow *instance with G can be solved in time $|V(G)|^{O(tw)}$ and an* MVTSP *instance with G can be solved in time $\min\{|V(G)|, M\}^{O(tw)}|V(G)|^{O(1)}$.*

Proof. The algorithm is based on a standard dynamic programming approach; we only describe the table entries and omit the recurrence to compute table entries since it is standard. We assume we are given a tree decomposition $\mathcal{T} = (\{X_i\}, R)$ on the given graph. Let Y_i be the set of vertices in the subtree rooted at X_i. For a given bag X_i, let π be a partition on X_i. Furthermore, let $\mathbf{d}^{\mathsf{in}} = (d_v^{\mathsf{in}})_{v \in X_i} \in \mathbb{N}^{X_i}$ and $\mathbf{d}^{\mathsf{out}} = (d_v^{\mathsf{out}})_{v \in X_i} \in \mathbb{N}^{X_i}$ be two vectors of integers, indexed by X_i.

We define the dynamic programming table entry $T(X_i, \pi, \mathbf{d}^{\mathsf{in}}, \mathbf{d}^{\mathsf{out}})$ to be the cost of the cheapest partial solution on the graph "below" the bag X_i, among solutions whose connected components agree with the partition π and whose in and out degrees agree with the vectors $\mathbf{d}^{\mathsf{in}}$ and $\mathbf{d}^{\mathsf{out}}$. For each bag X_i, a partition π of X_i and sequences $\mathbf{d}^{\mathsf{in}}$ and $\mathbf{d}^{\mathsf{out}}$ satisfying (i) $0 \leqslant d_v^{\mathsf{in}}, d_v^{\mathsf{out}} \leqslant \mathsf{dem}(v)$ for each $v \in D$ and (ii) $0 \leqslant d^{\mathsf{in}}(v), d^{\mathsf{out}}(v) \leqslant M|V(G)|$ for each $v \notin D$, define $T(X_i, \pi, \mathbf{d}^{\mathsf{in}}, \mathbf{d}^{\mathsf{out}}) = \min_s \mathsf{cost}(s)$ over all $s : Y_i^2 \to \mathbb{N}$ such that the following conditions hold:

1. $\sum_{u \in Y_i} s(u, v) = \sum_{u \in Y_i} s(v, u) = \mathsf{dem}(v)$ for all $v \in D \cap (Y_i \setminus X_i)$,
2. $\sum_{u \in Y_i} s(u, v) = d_v^{\mathsf{in}}$ for all $v \in X_i$,
3. $\sum_{u \in Y_i} s(v, u) = d_v^{\mathsf{out}}$ for all $v \in X_i$,
4. each block of the partition π is weakly connected in G_s,
5. $s(u, v) \leqslant \mathsf{cap}(u, v)$ for all $(u, v) \in E(G[Y_i])$.

We can compute the table starting at the leaves of R and work our way towards the root.

Let us examine the size of this dynamic programming table. First we note that there are at most $|V(G)|^{O(1)}$ bags in the tree decomposition. Next we consider the values d_v^{in} and d_v^{out}. Note that we can assume that an optimal solution only visits any vertex without demand at most $M|V(G)|$ times: Any solution can be decomposed into a collection of paths between vertices with demand. Each such path can be assumed to not visit any vertex more than once (except possibly in the end points of the path) since the solution is of minimum weight and all costs are non-negative. We find that each vertex gets visited at most $M|V(G)|$ times and thus we only need to consider $M|V(G)|$ many values of d_v^{in} and d_v^{out}. Thus the degree values of the partial solutions contribute a factor of $(M|V(G)|)^{O(tw)}$ to the overall running time of the algorithm if the given instance is a Connected Flow instance, and only $M^{O(tw)}$ if the given instance is an MVTSP instance (in which all vertices are demand vertices).

We will show that we may assume that $M = |V(G)|^{O(1)}$. Together with the fact that the number of possibilities for π is $tw^{O(tw)} \leqslant |V(G)|^{O(tw)}$, the claimed result for Connected Flow follows.

Claim 6.6. Given an instance of Connected Flow, there is an equivalent instance such that $M = |V(G)|^{O(1)}$.

Proof of Claim: We will use a variation of the proof of Theorem 3.4 from Kowalik et al. [85]. Let r be an optimal solution to Flow. By applying Lemma 6.8 with T being a subtree of G_s spanning all demand vertices, we find that there is an optimal solution s of Connected Flow such that $|r(u,v) - s(u,v)| \leqslant 2n$ for every edge (u,v).

We now construct a flow f from r by subtracting simple directed cycles from r. Note that each time that we subtract such a cycle, the result is again a flow. We start with $f = r$ on all edges. Now if there is an edge $(u,v) \in E$ for which $f(u,v) > \max\{r(u,v) - 2n - 1, 0\}$, we can find a simple directed cycle $C \in G_f$ containing (u,v) as f is a flow and thus G_f is Eulerian. Define $f'(u,v) = f(u,v) - [(u,v) \in C]$. Note that f' is again a flow. Set $f = f'$. We repeat this process of subtracting simple directed cycles from f until $f(u,v) \leqslant \max\{r(u,v) - 2n - 1, 0\}$ for every edge (u,v).

Note that $0 \leqslant s(u,v) - f(u,v)$ for all $(u,v) \in E$. Define a new instance with $\mathsf{dem}'(v) = \mathsf{dem}(v) - \sum_{u \in V} f(u,v)$ and $\mathsf{cap}'(u,v) = \mathsf{cap}(u,v) - f(u,v)$. Note that $s(u,v) - f(u,v)$ is an optimal connected flow for this instance. If $\mathsf{dem}'(v) \leqslant 2n^2 + n$ we are done. Otherwise let r' be a relaxed solution for the new instance. Note that there is an edge (u',v') for which $r'(u',v') > 2n+1$ and thus we can repeat the previous argument to find a non-zero flow f' such that $f'(u,v) \leqslant \max\{r'(u,v) - 2n - 1, 0\}$ on every edge and define a corresponding new instance. Since each time we subtract a non-zero flow, after some number of repetitions we find $\mathsf{dem}'(v) \leqslant 2n^2 + n$. □

For the result for MVTSP, the above approach would give a running time of $\min\{|V(G)|, M\}^{O(tw)} tw^{O(tw)} |V(G)|^{O(1)}$. However, the factor $tw^{O(tw)}$ in the running time needed to keep track of all partitions π can be reduced to $2^{O(tw)}$ via a standard application of the rank based approach (see e.g. [45, Section 11.2.2] or [19, 46]). ■

6.5.2 Lower bound

In order to obtain a lower bound for MVTSP, we will modify a lower bound from Cygan et al. [46], which reduces 3-CNF-SAT to Hamiltonian Cycle.

We will create an instance of MVTSP that is symmetric in the sense that the graph G is undirected, hence we denote edges as unordered pairs of vertices (i.e. $\{u, v\} = \{v, u\}$). As a consequence, when c is a tour on G, then we say $c(u, v) = c(v, u)$. The general proof strategy is as follows. For a given 3-CNF-SAT formula ϕ on n variables[3] we will construct an equivalent MVTSP instance (G, d). This graph will consist of n/s paths, for some value of s, with each path propagating some information encoding the value of s variables of ϕ. For each clause of ϕ we will add a gadget which checks if the assignment satisfies the clause. We then bound the size and the pathwidth of the constructed graph G, obtaining the required lower bound.

Gadgets

We start describing the following gadget from Cygan et al. [46], called a *2-label gadget*.

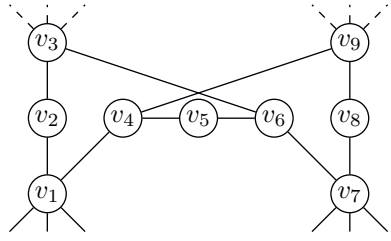


Figure 6.3: A 2-label gadget.

The key feature of this gadget is that if all vertices in the gadget have demand equal to 1, then if a solution tour enters the gadget at v_3, it has to leave the gadget at v_9 and vice versa. A similar relation holds for v_1 and v_7. We will refer to any edge connected to either v_1 or v_7 as having label 1 and any edge connected to v_3 or v_9 as having label 2. We will use this gadget to construct a gadget that can detect certain multisets of edges in a part of a graph. In this construction we will chain 2-label gadgets together using label 1 edges. Whenever we do this, we always connect the vertex v_7 of one gadget to the vertex v_1 in the next. To facilitate the construction, in the rest of this section we will refer to a 2-label gadget as if it were a single vertex.

The next gadget is also inspired by a construction from Cygan et al. [46].

Definition 6.10. *A* scanner gadget *in an unweighted* MVTSP *instance (G, d) is described by a tuple $(X, a, b, \mathcal{F})$, where $X \subseteq V$, $a, b \in V \setminus X$ with $\mathsf{dem}(a) = \mathsf{dem}(b) = 1$, $\mathcal{F}$ is a family of multisets of edges in[4] $E(X, X)$ and $\emptyset \notin \mathcal{F}$. We say that a tour c of G is* consistent with *$(X, a, b, \mathcal{F})$ if its restriction $c_{E(X,X)}$ is in $\mathcal{F}$ and if $c(a, b) > 0$.*

[3] In this section, we will only use n to refer to the number of variables of a 3-CNF-SAT instance.

[4] Here $E(X, X)$ are all edges with both endpoints in X and the restriction c_Y are all edges in c in Y (keeping multiplicities).

When refering to the gadget as a subgraph, we will use $G_{\mathcal{F}}$. We implement the scanner gadget using the following construction, obtaining an instance (G', dem') of MVTSP.

- Remove the edges in $E(X, X)$.
- Add an independent set $I = \{s_1, \ldots, s_\ell\}$ and edges $\{a, s_1\}$ and $\{s_\ell, b\}$, for $\ell = |\mathcal{F}|$.
- Let $\mathcal{F} = \{F_1, \ldots, F_\ell\}$. For $i = 1, \ldots, \ell$, let $F_i = \{e_1^{q_1}, \ldots, e_z^{q_z}\}$ (i.e. F_i contains q_i copies of e_i). For each $i \in [\ell]$, do the following:
  - Add a path $P_i = \{p_i^1, \ldots, p_i^{t_i}\}$ of 2-label gadgets, where $t_i = |F_i| = \sum_{i=j}^{z} q_j$. We connect the gadgets in a chain using label 1 edges.
  - Connect p_i^1 to s_{i-1} and s_i using label 1 edges (green edges in Figure 6.4) and connect $p_i^{|F_i|}$ to s_i and s_{i+1} using label 1 edges (blue edges in Figure 6.4).
  - For all $j = 1, \ldots, z$ add label 2 edges from x to $p_i^{j'}$ and from y to $p_i^{j'}$ for $e_j = \{x, y\}$ and for q_j different, previously unused values of j' (red edges in Figure 6.4).
- We set the demand of all added vertices to 1.

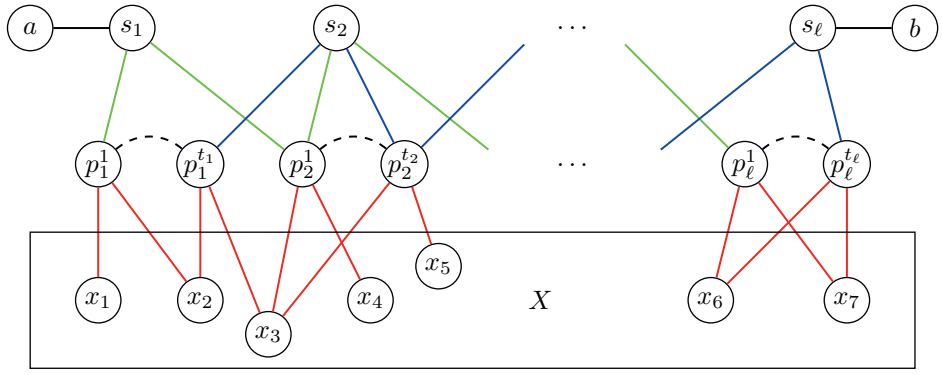


Figure 6.4: Example of the scanner gadget.

The function of the gadget is captured by the following lemma.

Lemma 6.11. *There exists an tour on (G, dem) that is consistent with $(X, a, b, \mathcal{F})$ if and only if there exists a tour on (G', dem').*

Proof. Our proof will closely follow that in Cygan et al. [46]. Suppose we have a tour on (G, dem) which is consistent with a gadget $(X, a, b, \mathcal{F})$. Let $F_i \in \mathcal{F}$ be the restriction of the tour on $E(X, X)$. Then the tour on (G, dem) can be extended to a tour on (G', dem') by replacing the q_j instances of an edge $\{u, v\} \in F_i$ with two edges

$\{u, p_i^{j'}\}$ and $\{v, p_i^{j'}\}$ for q_j different values of j'. We also replace the edge $\{a, b\}$ by the path

$$a, s_1, P_1, \ldots, P_{i-1}, s_i, P_{i+1}, s_{i+1}, \ldots, P_\ell, s_\ell, b.$$

Since the obtained tour visits all vertices in the gadget exactly once and since the restriction of the adjusted tour connects the same pairs of vertices in X as the restriction of the original tour, the obtained tour is a solution for the instance (G', dem').

For the other direction, suppose we have a tour c' on (G', dem'). Note that by the properties of the 2-label gadgets no tour can cross from some s_i into X through one of the 2-label gadgets in one of the paths P_i. Thus the tour can only travel from outside the gadget to s_i, by going through a or b. Therefore the tour must include the edges $\{a, s_1\}$ and $\{s_\ell, b\}$. Furthermore, s_1 and s_ℓ must be connected by some path P' in the tour. Since I is an independent set, P' has to alternate between the P_i's and the s_i's and has to include every s_i, since this is the only way to reach a vertex s_i.

This means that there is exactly one path P_{i_0} which is not covered by P'. We can now obtain a tour c of (G, dem) by first setting $c(u, v) = c'(u, v)$ for $\{u, v\} \neq \{a, b\}$ for u or v not in X. We then include any edge in X a number of times according to its multiplicity in F_{i_0} i.e. we set $c(u, v) = F_{i_0}(u, v)$. Finally, we set $c(a, b) = 1$. Note that since $c(a, b) > 0$ and $F_{i_0} \in \mathcal{F}$, we conclude that c is consistent with $(X, a, b, \mathcal{F})$. ■

The following lemma will allow us to implement the gadget without increasing the pathwidth of the graph too much.

Lemma 6.12. *The scanner gadget has pathwidth at most* $|X| + 21$.

Proof. We define the bags of the decomposition as follows

$$\begin{aligned} B_a &:= X \cup \{a, s_1\} \\ B_{i,j} &:= X \cup \{s_{i-1}, s_i, s_{i+1}, p_i^j, p_i^{j+1}\} \\ B_b &:= X \cup \{b, s_\ell\}. \end{aligned}$$

It is easy to see that the following bags form a path decomposition of $G_{\mathcal{F}}$: $B_a, B_{1,1}, B_{1,2}, \ldots B_{1,t_1}, B_{2,1}, \ldots B_{\ell,t_\ell}, B_b$. ■

Construction

Suppose we are given a 3-CNF-SAT formula $\phi = C_1 \wedge \cdots \wedge C_m$. We will construct an equivalent unweighted MVTSP instance Γ_ϕ using scanner gadgets. We will interpret a tuple $(q, j) \in \{1, \ldots, 2^s\} \times \{1, \ldots, n/s\}$ as an assignment of $x_{(j-1)s+1}, \ldots, x_{js}$ by first decomposing

$$q - 1 = \sum_{i=1}^{s} c_i 2^{i-1}$$

and setting $x_{(j-1)s+i}$ to true if $c_i = 1$ and false if $c_i = 0$. We say a clause C is satisfied by a set Q of such tuples if $j \neq j'$ for all $(q, j), (q, j') \in Q$, and if the partial assignment given by the tuples satisfies C. Let s be a constant that will be determined later.

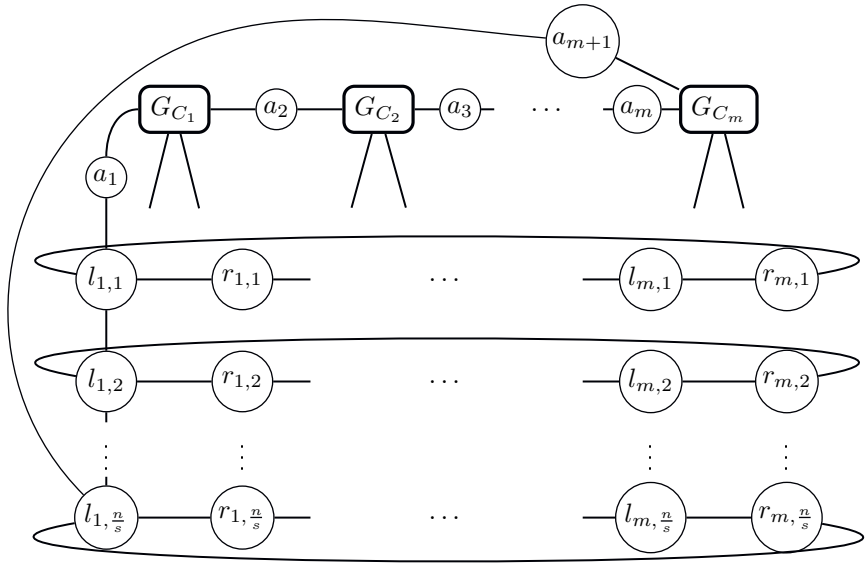


Figure 6.5: Construction of the graph Γ_ϕ.

- We start by creating vertices $l_{i,1}, \ldots, l_{i,n/s}$ and $r_{i,1}, \ldots, r_{i,n/s}$ for $i = 1, \ldots, m$. [5]

- We set the demand of $l_{1,j}$ to $2^s + 1$ for $j = 1, \ldots, n/s$ and add edges $\{l_{1,j}, l_{1,j+1}\}$ for $j = 1, \ldots, n/s - 1$.

- We set the demand of every other $l_{i,j}$ and every $r_{i,j}$ to 2^s and add edges $\{l_{i,j}, r_{i,j}\}$, $\{r_{i,j}, l_{i+1,j}\}$ and $\{r_{m,j}, l_{1,j}\}$ for $i = 1, \ldots, m-1$ and $j = 1, \ldots, n/s$.

- We connect $l_{1,1}$ to $l_{1,n/s}$ using a path $a_1, \ldots, a_{m+1}$.

- For $i = 1, \ldots, m$ let x_a, x_b, x_c be the variables appearing in C_i. We set $j_1 = \lceil a/s \rceil, j_2 = \lceil b/s \rceil, j_3 = \lceil c/s \rceil$. Let

$$X = \{l_{i,j_1}, l_{i,j_2}, l_{i,j_3}, r_{i,j_1}, r_{i,j_2}, r_{i,j_3}\}$$

  and let $\mathcal{F}_{C_i}$ be the set of all

$$F = \{\{l_{i,j_1}, r_{i,j_1}\}^{q_1}, \{l_{i,j_2}, r_{i,j_2}\}^{q_2}, \{l_{i,j_3}, r_{i,j_3}\}^{q_3}, \}$$

  such that $Q = \{(q_1, j_1), (q_2, j_2), (q_3, j_3)\}$ satisfies C_i.

- For $i = 1, \ldots, m$ we implement a scanner gadget G_{C_i} using the tuple $(X_i, a_i, a_{i+1}, \mathcal{F}_{C_i})$

[5] If n is not divisible by s, we may either add dummy variables until it is, or lower the demand of $l_{i,n/s}$ and $r_{i,n/s}$.

We prove the following useful facts about this graph.

Lemma 6.13. *Γ_ϕ is a yes instance of* MVTSP *if and only if ϕ has a satisfying assignment.*

Proof. Let Γ'_ϕ be the graph obtained by the above construction before implementing the scanner gadgets. Let $x_1, \ldots, x_n$ be the variables used in the formula ϕ. Let $\chi_1, \ldots, \chi_n$ be a satisfying assignment. We first define the tour on Γ'_ϕ and then use Lemma 6.11 to find the desired tour on Γ_ϕ. Set $c(l_{i,j}, l_{i+1,j}) = c(l_{1,1}, a_1) = c(a_{m+1}, l_{1,n/s}) = c(a_i, a_{i+1}) = 1$. We set

$$c'(l_{i,j}, r_{i,j}) = 1 + \sum_{k=1}^{s} 2^{k-1} \chi_{(j-1)s+k}$$

for $i = 1, \ldots, m$ and $j = 1, \ldots, n/s$. Due to the chosen demands we need to set

$$c'(r_{i,j}, l_{i+1,j}) = 2^{s+1} - c'(l_{i,j}, r_{i,j})$$

for $i = 1, \ldots, m$ and $j = 1, \ldots, n/s$, where we interpret i modulo m, i.e. $m + 1 \equiv 1$. Note that c' is connected and satisfies the demands on Γ'_ϕ. Also note that since χ is a satisfying assignment, c' is consistent with all the scanner gadgets G_{C_i} and thus by Lemma 6.11 we there is some valid tour c on Γ_ϕ.

Now suppose there exists a valid tour c on Γ_ϕ. Then by Lemma 6.11 there exists a tour c' on Γ'_ϕ consistent with each gadget G_{C_i}. By definition of G_{C_i}, the values of $c'(l_{i,j}, r_{i,j})$ encode an assignment satisfying C_i for $i = 1, \ldots, m$. Since for $i \geqslant 2$ the demands of $l_{i,j}$ and $r_{i,j}$ equal 2^s, we have that $c'(l_{i,j}, r_{i,j}) = 2^{s+1} - c'(r_{i,j}, l_{i+1,j}) = c'(l_{i+1,j}, r_{i+1,j})$ and therefore the values of $c'(l_{1,j}, r_{1,j})$ encode an assignment satisfying all clauses $C_1, \ldots, C_m$, which means we obtain an assignment which satisfies ϕ. ∎

Let us now bound the pathwidth of Γ_ϕ.

Lemma 6.14. *Γ_ϕ has pathwidth at most $3n/s + 21$.*

Proof. We define the bags of the decomposition as follows. First we add

$$A = \{l_{1,1}, \ldots, l_{1,n/s}\}$$

to every bag. Let $W_1, \ldots, W_{l_i}$ be a path decomposition of G_{C_i}. We define the bag $X_{i,j}$ as follows:

$$X_{i,j} = A \cup \{l_{i,k} : k \in [n/s]\} \cup \{r_{i,k} : k \in [n/s]\} \cup W_j.$$

We then define Y_i as $\{l_{i+1,k} : k \in [n/s]\} \cup \{r_{i,k} : k \in [n/s]\}$. The final path decomposition then becomes

$$X_{1,1}, \ldots, X_{1,l_1}, Y_1, X_{2,1}, \ldots, X_{i,l_i}, Y_i, X_{i+1,1}, \ldots, X_{m,l_m}.$$

It is easy to see that the above bags indeed form a path decomposition.

By Lemma 6.12 (noting that $X \subseteq \{l_{i,k} : k \in [n/s]\} \cup \{r_{i,k} : k \in [n/s]\}$) the width of this path decomposition is at most $3 \cdot \frac{n}{s} + 21$. ∎

Now we use our reduction to prove the following lower bound:

Theorem 6.15. *Let M be an upper bound on the demands in a graph G. Then* MVTSP *cannot be solved in time $f(pw)\min\{|V(G)|, M\}^{o(pw)}|V(G)|^{O(1)}$, unless ETH fails.*

Proof. We start by proving the following claim.

Claim 6.7. $|V(G_{C_i})| = O(2^{3s})$ for $i = 1, \ldots, m$.

Proof of Claim: Note that $\mathcal{F}_{C_i}$ is defined on at most three unique edges with each edge being chosen at most 2^s times. Due to the way we interpret the multiplicities as truth assignments, we know each edge gets chosen at least once. Therefore, we can represent $\mathcal{F}_{C_i}$ by tuples $(z_1, z_2, z_3) \in [2^s]^3$. Since each tuple contributes a path of $z_1 + z_2 + z_3$ vertices, we find that

$$\begin{aligned}
|V(G_{C_i})| &= 8 + |\mathcal{F}_{C_i}| + \sum_{(z_1,z_2,z_3)\in\mathcal{F}_{C_i}} (z_1 + z_2 + z_3) \\
&\leqslant 2^{3s+1} + \sum_{z_1,z_2=1}^{2^s} \left(2^s(z_1 + z_2) + \sum_{z_3=1}^{2^s} z_3\right) \\
&\leqslant 2^{3s+1} + \sum_{z_1,z_2=1}^{2^s} \left(2^s(z_1 + z_2) + 2^{s+1}\right) \\
&\leqslant 2^{3s+1} + \sum_{z_1=1}^{2^s} \left(2^{2s} z_1 + 2^{2s+1} + 2^{2s+1}\right) \\
&\leqslant 2^{3s+3}.
\end{aligned}$$

□

Note that by Lemma 6.13, solving a 3-CNF-SAT instance ϕ reduces to solving MVTSP on Γ_ϕ for some choice of s. We will show that there is no $O(f(pw)M^{o(pw)}|V(G)|^{O(1)})$ time algorithm for MVTSP, unless ETH fails.

Suppose we have a $O\left(f(pw)M^{o(pw)}|V(G)|^{O(1)}\right)$ time algorithm for MVTSP. Let $s = 4n/g(n)$ for some strictly increasing function $g(n) = 2^{o(n)}$ such that $f(g(n)) = 2^{o(n)}$. Note that $s = o(n)$ and $pw \leqslant g(n)$ for large enough n. We construct the instance Γ_ϕ as previously described. We first note that by claim 6.7

$$|V(\Gamma_\phi)| = 2m\frac{n}{s} + \sum_{i=1}^{m} |V(G_{C_i})| = O\left(m\left(\frac{n}{s} + 2^{3s}\right)\right)$$

and by Lemma 6.14 we have that for any choice of s and large enough n, Γ_ϕ has pathwidth at most $4n/s$. By our reduction, we now find an algorithm for 3-CNF-SAT running in time

$$\begin{aligned}
O\left(f(pw)M^{o(pw)}|V(\Gamma_\phi)|^{O(1)}\right) &= O\left(f(4n/s)(2^s)^{o(n/s)}\left(m\left(\frac{n}{s} + 2^{3s}\right)\right)^{O(1)}\right) \\
&= O\left(f(g(n))\cdot 2^{o(n)}\cdot\left(m\left(g(n)/4 + 2^{o(n)}\right)\right)^{O(1)}\right).
\end{aligned}$$

We may assume that $m = 2^{o(n)}$ by the sparsification lemma [75]. Using this and the fact that $g(n) = 2^{o(n)}$ we obtain an algorithm for 3-CNF-SAT running in time $O\left(2^{o(n)} \cdot \left(2^{o(n)}\right)^{O(1)}\right) = O\left(2^{o(n)}\right)$. This contradicts ETH, completing our proof. ■

6.6 Conclusion

We initiated the study of the parameterized complexity of the Connected Flow problem and showed that the problem behaves very differently when parameterized by the number of demand vertices, the size of the vertex cover, or treewidth of the input graph.

While we essentially settled the complexity of the problem parameterized by the number of demands or by the treewidth, we still leave the following questions open for the vertex cover parameterization:

Can Connected Flow be solved in $O^\star(c^{O(k)})$ time, with c a constant and k the size of the vertex cover of the input graph? Such an algorithm would be a strong generalization of the algorithms from [13, 85]. While we believe our approach from Theorem 6.3 makes significant progress towards solving this question affirmatively, it seems that non-trivial ideas are required.

Does Connected Flow admit a kernel polynomial in k where k is the size of the vertex cover if the input graph? It would be interesting to see if our arguments for Theorem 6.4 can be extended to kernelize this more general problem as well.

Chapter 7

Conclusion

In this thesis, we studied several problems from the parameterized complexity point of view. We showcased classical algorithmic paradigms such as branching and dynamic programming, combined with various problem specific insights to obtain efficient algorithms. In some cases, we also presented matching (conditional) lower bounds, thus proving that the corresponding algorithms are in some sense optimal. The problems studied in this thesis are of both theoretical and practical interest, thus we hope our results inspire further research in this direction. In the remainder of this chapter, we discuss several open problems.

Open problems related to Chapter 3. In Chapter 3, we studied the Subgraph Isomorphism problem on unit disk graphs. A natural question to ask is whether our results can be adapted to other geometric intersection graphs, e.g. polygons of the same size. More generally, can our techniques be used for solving the Subgraph Isomorphism problem on a larger graph class, such as intersection graphs of fat objects? Our approach relies on several properties of unit disk graphs. Firstly, we rely on a result from [27] bounding the number of unlabeled unit disk graphs, thus the first step would be obtaining a similar bound for other graph classes.

In order to bound the number of non-isomorphic separations, we use the fact that each vertex has a small number of neighbours which are not adjacent to each other, which also holds for some other geometric graph classes, e.g. intersections of unit squares. We also exploit the fact that we are working with objects of the same size, which allows us to ensure that each of these objects has a bounded number of grid lines intersecting it. This remains true if we consider intersection graphs of objects where the ratio of the smallest and largest one is bounded. Finally, we remark that an alternative way to ensure that the separators corresponding to vertical lines are small is to consider λ-precision disk graphs, where the distance between centres of two disks is at least λ.

Since our algorithms heavily rely on the embedding, one important question to ask is whether we can efficiently construct an embedding of a given unit disk graph. This problem is known to be NP-hard [30]. Note that it is not even clear whether the problem is in NP: the natural certificate, consisting of coordinates of disk centres,

might contain irrational points. It is known that every unit disk graph can be realized as the intersection graph of disks with the same integer radius and whose centres have integer coordinates. However, it was shown in [97] that these coordinates can be doubly exponentially large. In particular, we cannot obtain a polynomially sized certificate in this way.

Open problems related to Chapter 4. Chapter 4 concerns a special case of Set Cover and Exact Cover problems, where the universe is a set of integers, and the sets correspond to arithmetic progressions. The question of strong NP-hardness remains open. Another natural question to ask is whether our algorithms are optimal. In particular, can we formalize the intuition that the Exact Cover by Arithmetic Progressions is harder than the Cover by Arithmetic Progressions problem?

Arithmetic progressions can be defined in any group, so it is natural to ask whether we can extend our algorithms to groups other than $\mathbb{Z}$. We note that our algorithms use the result of [42] (that any collection of k infinite APs that cover the set $\{1, 2, \ldots, 2^k\}$ also cover the whole set of positive integers) applied to *indices* rather than elements of the input set. Hence, the two structural lemmas hold for any group. However, in our algorithms we use the fact that if we know two elements a and b of an AP, and we know how many elements the AP has between a and b, we immediately conclude the difference of the AP. In other words, our algorithms rely on division, and the XCAP algorithm additionally relies on computation of greatest common divisors.

Another question concerns the parameterization by the number of APs that cover a set. Intuitively, if a set can be covered by a small number of APs, that gives us some insight into its structure. A possible direction for further research is whether this structure can be exploited for solving problems like Subset Sum. Since number theoretic problems have not been well studied from the parameterized point of view, we hope our results inspire further research in this direction.

Open problems related to Chapter 5. In Chapter 5, we prove XALP-completeness and XNLP-hardness for several problems on planar graphs. For some of the problems we show XNLP-hardness, and it is not clear whether they belong to the class XNLP. We conjecture that these problems are in fact XALP-complete. Since the classes XNLP and XALP have been introduced only recently, not many complete problems on planar graphs are known, so a natural question to ask is which other problems are XNLP-hard and XALP-hard on planar graphs.

Some of the gadgets used in our hardness reductions can be easily transformed into unit disk graphs by replacing edges by paths composed of unit disks. Roughly speaking, any gadget that does not have a high degree vertex (more precisely, a vertex with at least 6 nonadjacent neighbours) can be embedded as a unit disk graph, and this procedure does not change the treewidth. The challenge lies in replacing the high degree vertices (e.g. the sink in the All-or-Nothing Flow proof or the edge gadgets in the reductions from All-or-Nothing Flow).

Another direction for further research concerns the computation of tree decomposition. Namely, tree decomposition can be computed in FPT time and polynomial space (see e.g. [84]), and in XP time and logarithmic space [54]. However, it is not known whether there is an algorithm that computes the tree decomposition in FPT

time and logarithmic space. Such an algorithm would allow us to remove the assumption that we are given the tree decomposition as a part of the input.

Open problems related to Chapter 6. Chapter 6 studies the MANY VISITS TSP and CONNECTED FLOW problems. Since we showed that MVTSP has a polynomial kernel parameterized by vertex cover, it is natural to ask whether this result extends to CONNECTED FLOW. Another possible direction for further research would be a faster algorithm parameterized by vertex cover (or a lower bound proving that such algorithm does not exist).

The natural parameterization by the number of demand vertices $|D|$ could be studied further. Since CONNECTED FLOW is NP-complete even when there are only two demand vertices, we do not expect it to be FPT parameterized by $|D|$. Thus it is natural to ask whether it is hard for some class in the W hierarchy (or even XNLP-hard). Obtaining efficient algorithms parameterized by $|D|$ (in case when the capacities are finite) is another possible direction for further research.

In [12], the many visits *multiple* TSP was introduced. In this problem, we have m salesmen visiting n cities, and each city needs to be visited the required number of times by the salesmen. In [12], several variants of this problem have been considered and several approximation algorithms have been described. A possible direction for further research would be to study the analogous generalization of CONNECTED FLOW, which asks for m connected flows satisfying the vertex demands.

Bibliography

[1] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Annals of Mathematics*, pages 781–793, 2004.

[2] Jochen Alber, Michael R. Fellows, and Rolf Niedermeier. Polynomial-time data reduction for dominating set. *Journal of the ACM*, 51(3):363–384, 2004.

[3] Per Alexandersson. NP-complete variants of some classical graph problems. *arXiv*, abs/2001.04120, 2020.

[4] László Babai. Graph isomorphism in quasipolynomial time (extended abstract). In *Proceedings of the 48th Annual ACM Symposium on Theory of Computing (STOC 2016)*, pages 684–697. ACM, 2016.

[5] Brenda S. Baker. Approximation algorithms for NP-complete problems on planar graphs. *Journal of the ACM*, 41(1):153–180, 1994.

[6] Manu Basavaraju, Mathew C. Francis, M.S. Ramanujan, and Saket Saurabh. Partially polynomial kernels for set cover and test cover. *SIAM Journal on Discrete Mathematics*, 30(3):1401–1423, 2016.

[7] Hannah Bast and Sabine Storandt. Frequency data compression for public transportation network algorithms. In *Proceedings of the 6th International Symposium on Combinatorial Search (SOCS 2013)*, volume 4, pages 205–206, 2013.

[8] Hannah Bast and Sabine Storandt. Frequency-based search for public transit. In *Proceedings of the 22nd ACM International Conference on Advances in Geographic Information Systems (SIGSPATIAL 2014)*, pages 13–22, 2014.

[9] Richard Bellman. Dynamic programming treatment of the Travelling Salesman Problem. *Journal of the ACM*, 9(1):61–63, 1962.

[10] Oren Ben-Zwi, Danny Hermelin, Daniel Lokshtanov, and Ilan Newman. Treewidth governs the complexity of target set selection. *Discrete Optimization*, 8(1):87–96, 2011.

[11] Kristóf Bérczi, Matthias Mnich, and Roland Vincze. A 3/2-approximation for the metric many-visits path TSP. *SIAM Journal on Discrete Mathematics*, 36(4):2995–3030, 2022.

[12] Kristóf Bérczi, Matthias Mnich, and Roland Vincze. Approximations for many-visits multiple Traveling Salesman Problems. *Omega*, 116:102816, 2023.

[13] André Berger, László Kozma, Matthias Mnich, and Roland Vincze. Time- and space-optimal algorithm for the many-visits TSP. *ACM Transactions on Algorithms*, 16(3):35:1–22, 2020.

[14] Sujoy Bhore, Paz Carmi, Sudeshna Kolay, and Meirav Zehavi. Parameterized study of Steiner tree on unit disk graphs. *Algorithmica*, 85(1):133–152, 2023.

[15] Andreas Björklund, Thore Husfeldt, and Mikko Koivisto. Set partitioning via inclusion-exclusion. *SIAM Journal on Computing*, 39(2):546–563, 2009.

[16] Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996.

[17] Hans L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209(1-2):1–45, 1998.

[18] Hans L. Bodlaender, Gunther Cornelissen, and Marieke van der Wegen. Problems hard for treewidth but easy for stable gonality. In *Proceedings of the 48th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2022)*, volume 13453 of *LNCS*, pages 84–97. Springer, 2022.

[19] Hans L. Bodlaender, Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Deterministic single exponential time algorithms for connectivity problems parameterized by treewidth. *Information and Computation*, 243:86–111, 2015.

[20] Hans L. Bodlaender, Carla Groenland, and Hugo Jacob. List colouring trees in logarithmic space. In *Proceedings of the 30th Annual European Symposium on Algorithms (ESA 2022)*, volume 244 of *LIPIcs*, pages 24:1–15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

[21] Hans L. Bodlaender, Carla Groenland, Hugo Jacob, Lars Jaffke, and Paloma T. Lima. XNLP-completeness for parameterized problems on graphs with a linear structure. In *Proceedings of the 17th International Symposium on Parameterized and Exact Computation, (IPEC 2022)*, volume 249 of *LIPIcs*, pages 8:1–18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[22] Hans L. Bodlaender, Carla Groenland, Hugo Jacob, Marcin Pilipczuk, and Michal Pilipczuk. On the complexity of problems on tree-structured graphs. In *Proceedings of the 17th International Symposium on Parameterized and Exact Computation (IPEC 2022)*, volume 249 of *LIPIcs*, pages 6:1–17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[23] Hans L. Bodlaender, Carla Groenland, Jesper Nederlof, and Céline M. F. Swennenhuis. Parameterized problems complete for nondeterministic FPT time and logarithmic space. In *Proceedings of the 62nd IEEE Annual Symposium on Foundations of Computer Science (FOCS 2021)*, pages 193–204. IEEE Computer Society, 2022.

[24] Hans L. Bodlaender, Daniel Lokshtanov, and Eelko Penninkx. Planar capacitated dominating set is $W[1]$-hard. In *Proceedings of the 4th International Workshop on Parameterized and Exact Computation (IWPEC 2009)*, volume 5917 of *LNCS*, pages 50–60. Springer, 2009.

[25] Hans L. Bodlaender, Isja Mannens, Jelle J. Oostveen, Sukanya Pandey, and Erik Jan van Leeuwen. The parameterised complexity of integer multicommodity flow. In *Proceedings of the 18th International Symposium on Parameterized and Exact Computation (IPEC 2023)*, volume 285 of *LIPIcs*, pages 6:1–6:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[26] Hans L. Bodlaender, Jesper Nederlof, and Tom C. van der Zanden. Subexponential time algorithms for embedding H-minor free graphs. In *Proceedings of the 43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, volume 55 of *LIPIcs*, pages 9:1–14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[27] Christian Borgs, Jennifer Chayes, Jeff Kahn, and László Lovász. Left and right convergence of graphs with bounded degree. *Random Structures and Algorithms*, 42:1–28, 1 2013.

[28] Guillem Brasó and Laura Leal-Taixé. Learning a neural solver for multiple object tracking. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR 2020)*, pages 6247–6257, 2020.

[29] Marco Bressan, Flavio Chierichetti, Ravi Kumar, Stefano Leucci, and Alessandro Panconesi. Motif counting beyond five nodes. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 12(4):1–25, 2018.

[30] Heinz Breu and David G. Kirkpatrick. Unit disk graph recognition is NP-hard. *Computational Geometry*, 9(1-2):3–24, 1998.

[31] Karl Bringmann, László Kozma, Shay Moran, and N.S. Narayanaswamy. Hitting set for hypergraphs of low VC-dimension. In *Proceedings of the 24th Annual European Symposium on Algorithms (ESA 2016)*, volume 57, page 23. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2016.

[32] Karl Bringmann and Vasileios Nakos. Top-k-convolution and the quest for near-linear output-sensitive subset sum. In *Proceedings of the 52nd Annual ACM Symposium on Theory of Computing (STOC 2020)*, pages 982–995. ACM, 2020.

[33] Rowland Leonard Brooks. On colouring the nodes of a network. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 37, pages 194–197. Cambridge University Press, 1941.

[34] Leizhen Cai and Junjie Ye. Finding two edge-disjoint paths with length constraints. In *42nd International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2016), Revised Selected Papers*, volume 9941 of *LNCS*, pages 62–73, 2016.

[35] Alberto Caprara, Paolo Toth, and Matteo Fischetti. Algorithms for the set covering problem. *Annals of Operations Research*, 98(1):353–371, 2000.

[36] Timothy M. Chan. Finding triangles and other small subgraphs in geometric intersection graphs. In *Proceedings of the 34th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2023)*, pages 1777–1805. SIAM, 2023.

[37] Timothy M. Chan and Moshe Lewenstein. Clustered integer 3SUM via additive combinatorics. In *Proceedings of the 47th ACM Symposium on Theory of Computing (STOC 2015)*, pages 31–40. ACM, 2015.

[38] Beifang Chen and Sanming Zhou. Upper bounds for f-domination number of graphs. *Discrete Mathematics*, 185(1-3):239–243, 1998.

[39] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *Proceedings of the 63rd Annual IEEE Symposium on Foundations of Computer Science (FOCS 2022)*, pages 612–623. IEEE Computer Society, 2022.

[40] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC 1971)*, page 151–158. ACM, 1971.

[41] Stavros S. Cosmadakis and Christos H. Papadimitriou. The Traveling Salesman Problem with many visits to few cities. *SIAM Journal on Computing*, 13(1):99–108, 1984.

[42] R. B. Crittenden and C. L. Vanden Eynden. Any n arithmetic progressions covering the first 2^n integers cover all integers. *Proceedings of the American Mathematical Society*, 24(3):475–481, 1970.

[43] Robert Crowston, Gregory Gutin, Mark Jones, Venkatesh Raman, and Saket Saurabh. Parameterized complexity of MaxSAT above average. *Theoretical Computer Science*, 511:77–84, 2013.

[44] Marek Cygan. Deterministic parameterized connected vertex cover. In *Proceedings of the 13th Scandinavian Symposium on Algorithm Theory (SWAT 2012)*, pages 95–106. Springer, 2012.

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

[46] Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Fast Hamiltonicity checking via bases of perfect matchings. *Journal of ACM*, 65(3):12:1–12:46, 2018.

[47] David P. Dailey. Uniqueness of colorability and colorability of planar 4-regular graphs are NP-complete. *Discrete Mathematics*, 30(3):289–293, 1980.

[48] Samuel I. Daitch and Daniel A. Spielman. Faster approximate lossy generalized flow via interior point algorithms. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing (STOC 2008)*, pages 451–460. ACM, 2008.

[49] George B. Dantzig. Application of the simplex method to a transportation problem. *Activity analysis and production and allocation*, pages 359–373, 1951.

[50] Mark de Berg, Hans L. Bodlaender, Sándor Kisfaludi-Bak, Dániel Marx, and Tom C. van der Zanden. A framework for Exponential-Time-Hypothesis-tight algorithms and lower bounds in geometric intersection graphs. *SIAM Journal on Computing*, 49(6):1291–1331, 2020.

[51] Irit Dinur and David Steurer. Analytical approach to parallel repetition. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing (STOC 2014)*, pages 624–633. ACM, 2014.

[52] Michael Dom, Daniel Lokshtanov, Saket Saurabh, and Yngve Villanger. Capacitated domination and covering: a parameterized perspective. In *Proceedings of the 3rd International Workshop on Parameterized and Exact Computation (IWPEC 2008)*, volume 5018 of *LNCS*, pages 78–90. Springer, 2008.

[53] Rodney G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness II: On completeness for $W[1]$. *Theoretical Computer Science*, 141(1&2):109–131, 1995.

[54] Michael Elberfeld, Andreas Jakoby, and Till Tantau. Logspace versions of the theorems of Bodlaender and Courcelle. In *Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science (FOCS 2010)*, pages 143–152. IEEE Computer Society, 2010.

[55] Michael Elberfeld, Christoph Stockhusen, and Till Tantau. On the space and circuit complexity of parameterized problems: classes and completeness. *Algorithmica*, 71(3):661–701, 2015.

[56] Pál Erdős. Problems and results in number theory. In *Proceedings of the 9th Manitoba Conference on Numerical Mathematics and Computing*, pages 3–21, 1979.

[57] Pál Erdős and Pál Turán. On a problem of Sidon in additive number theory, and on some related problems. *Journal of the London Mathematical Society*, s1-16(4):212–215, 1941.

[58] Hiroshi Eto, Fengrui Guo, and Eiji Miyano. Distance-independent set problems for bipartite and chordal graphs. *Journal of Combinatorial Optimization*, 27(1):88–99, 2014.

[59] Michael Etscheid, Stefan Kratsch, Matthias Mnich, and Heiko Röglin. Polynomial kernels for weighted problems. *Journal of Computer and System Sciences*, 84:1–10, 2017.

[60] John F. Fink and Michael S. Jacobson. n-Domination in graphs. In *Graph Theory with Application to Algorithms and Computer Science*, pages 282–300. John Wiley and Sons, 1985.

[61] Fedor V. Fomin, Petr A. Golovach, Daniel Lokshtanov, and Saket Saurabh. Almost optimal lower bounds for problems parameterized by clique-width. *SIAM Journal on Computing*, 43(5):1541–1563, 2014.

[62] Fedor V. Fomin, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. Subexponential parameterized algorithms for planar and apex-minor-free graphs via low treewidth pattern covering. *SIAM Journal on Computing*, 51(6):1866–1930, 2022.

[63] Steven Fortune, John E. Hopcroft, and James Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10:111–121, 1980.

[64] András Frank and Éva Tardos. An application of simultaneous diophantine approximation in combinatorial optimization. *Combinatorica*, 7(1):49–65, 1987.

[65] Michael R. Garey and David S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. W. H. Freeman, 1979.

[66] Andrew Goldberg and Robert Tarjan. Solving minimum-cost flow problems by successive approximation. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC 1987)*, pages 7–18. ACM, 1987.

[67] Andrew V. Goldberg and Satish Rao. Beyond the flow decomposition barrier. *Journal of the ACM*, 45(5):783–797, 1998.

[68] W. D. Grobman and T. W. Studwell. Data compaction and vector scan e-beam system performance improvement using a novel algorithm for recognition of pattern step and repeats. *Journal of Vacuum Science and Technology*, 16(6):1803–1808, 1979.

[69] Qian-Ping Gu and Hisao Tamaki. Improved bounds on the planar branchwidth with respect to the largest grid minor size. *Algorithmica*, 64(3):416–453, 2012.

[70] Sariel Har-Peled, Haim Kaplan, Wolfgang Mulzer, Liam Roditty, Paul Seiferth, Micha Sharir, and Max Willert. Stabbing pairwise intersecting disks by five points. *Discrete Mathematics*, 344(7):112403, 2021.

[71] Lenwood S. Heath. Covering a set with arithmetic progressions is NP-complete. *Information Processing Letters*, 34(6):293–298, 1990.

[72] Michael Held and Richard M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.

[73] Dorit S. Hochbaum and Wolfgang Maass. Approximation schemes for covering and packing problems in image processing and VLSI. *Journal of the ACM*, 32(1):130–136, 1985.

[74] Bob Hough. Solution of the minimum modulus problem for covering systems. *Annals of Mathematics*, pages 361–382, 2015.

[75] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001.

[76] Klaus Jansen and Petra Scheffler. Generalized coloring for tree-like graphs. *Discrete Applied Mathematics*, 75(2):135–155, 1997.

[77] Frank Kammer. Determining the smallest k such that G is k-outerplanar. In *Proceedings of the 15th Annual European Symposium on Algorithms (ESA 2007)*, volume 4698 of *LNCS*, pages 359–370. Springer, 2007.

[78] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing (STOC 1984)*, pages 302–311. ACM, 1984.

[79] Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of the Symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[80] Ioannis Katsikarelis, Michael Lampis, and Vangelis Th. Paschos. Structurally parameterized d-scattered set. *Discrete Applied Mathematics*, 308:168–186, 2022.

[81] Ton Kloks. *Treewidth, Computations and Approximations*, volume 842 of *LNCS*. Springer, 1994.

[82] Donald E. Knuth. Dancing links. *arXiv*, abs/cs/0011047, 2000.

[83] Tuukka Korhonen. A single-exponential time 2-approximation algorithm for treewidth. In *Proceedings of the 62nd Annual IEEE Symposium on Foundations of Computer Science (FOCS 2021)*, pages 184–192. IEEE Computer Society, 2021.

[84] Tuukka Korhonen and Daniel Lokshtanov. An improved parameterized algorithm for treewidth. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing (STOC 2023)*, pages 528–541. ACM, 2023.

[85] Łukasz Kowalik, Shaohua Li, Wojciech Nadara, Marcin Smulewicz, and Magnus Wahlström. Many visits TSP revisited. In *Proceedings of the 28th Annual European Symposium on Algorithms (ESA 2020)*, volume 173 of *LIPIcs*, pages 66:1–22. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

[86] Richard E. Ladner. On the structure of polynomial time reducibility. *Journal of the ACM*, 22(1):155–171, 1975.

[87] Leonid A. Levin. Universal problems of full search. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973.

[88] Shen Lin and Brian W. Kernighan. An effective heuristic algorithm for the Traveling-Salesman Problem. *Operations Research*, 21:498–516, 1973.

[89] Xin Liu, Haojie Pan, Mutian He, Yangqiu Song, Xin Jiang, and Lifeng Shang. Neural subgraph isomorphism counting. In *Proceedings of the 26th ACM International Conference on Knowledge Discovery & Data Mining (KDD 2020)*, pages 1959–1969, 2020.

[90] Daniel Lokshtanov, Fahad Panolan, Saket Saurabh, Jie Xue, and Meirav Zehavi. Subexponential parameterized algorithms on disk graphs (extended abstract). In *Proceedings of the 33rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2022)*, pages 2005–2031. SIAM, 2022.

[91] Dániel Marx. On the optimality of planar and geometric approximation schemes. In *Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007)*, pages 338–348. IEEE Computer Society, 2007.

[92] Dániel Marx and Marcin Pilipczuk. Subexponential parameterized algorithms for graphs of polynomial growth. In *Proceedings of the 25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *LIPIcs*, pages 59:1–15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[93] Dániel Marx and Michal Pilipczuk. Everything you always wanted to know about the parameterized complexity of subgraph isomorphism (but were afraid to ask). In *Proceedings of the 31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*, volume 25 of *LIPIcs*, pages 542–553. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014.

[94] Dániel Marx and Ildikó Schlotter. Cleaning interval graphs. *Algorithmica*, 65:275–316, 2013.

[95] David W. Matula. Subtree isomorphism in $O(n^{5/2})$. In *Annals of Discrete Mathematics*, volume 2, pages 91–106. Elsevier, 1978.

[96] Ciaran McCreesh, Patrick Prosser, and James Trimble. The Glasgow subgraph solver: using constraint programming to tackle hard subgraph isomorphism problem variants. In *Proceedings of the 13th International Conference on Graph Transformation (ICGT 2020)*, pages 316–324. Springer, 2020.

[97] Colin McDiarmid and Tobias Müller. Integer realizations of disk and segment graphs. *Journal of Combinatorial Theory, Series B*, 103(1):114–143, 2013.

[98] Marta Mesquita and Ana Paias. Set partitioning/covering-based approaches for the integrated vehicle and crew scheduling problem. *Computers & Operations Research*, 35(5):1562–1575, 2008.

[99] Moni Naor, Leonard J. Schulman, and Aravind Srinivasan. Splitters and near-optimal derandomization. In *Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1995)*, pages 182–191. IEEE Computer Society, 1995.

[100] Jesper Nederlof. Detecting and counting small patterns in planar graphs in subexponential parameterized time. In *Proceedings of the 52nd Annual ACM Symposium on Theory of Computing (STOC 2020)*, pages 1293–1306. ACM, 2020.

[101] Jaroslav Nešetřil and Patrice Ossona De Mendez. *Sparsity: graphs, structures, and algorithms*, volume 28. Springer, 2012.

[102] Fahad Panolan, Saket Saurabh, and Meirav Zehavi. Contraction decomposition in unit disk graphs and algorithmic applications in parameterized complexity. In *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2019)*, pages 1035–1054. SIAM, 2019.

[103] Pedro Paredes and Pedro Ribeiro. Rand-FaSE: fast approximate subgraph census. *Social Network Analysis and Mining*, 5(1):17, 2015.

[104] Ha-Myung Park, Francesco Silvestri, U Kang, and Rasmus Pagh. MapReduce triangle enumeration with guarantees. In *Proceedings of the 23rd ACM International Conference on Information and Knowledge Management (CIKM 2014)*, pages 1739–1748, 2014.

[105] Erwin Pesch and Fred Glover. TSP ejection chains. *Discrete Applied Mathematics*, 76:165–181, 1997.

[106] Geevarghese Philip, M. R. Rani, and R. Subashini. On computing the Hamiltonian index of graphs. *Theoretical Computer Science*, 940(Part):149–179, 2023.

[107] Michał Pilipczuk and Marcin Wrochna. On space efficiency of algorithms working on structural decompositions of graphs. *ACM Transactions on Computation Theory*, 9(4):18:1–18:36, 2018.

[108] Nataša Pržulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 23(2):177–183, 2007.

[109] Pedro Ribeiro, Pedro Paredes, Miguel E.P. Silva, David Aparicio, and Fernando Silva. A survey on subgraph counting: concepts, algorithms, and applications to network motifs and graphlets. *ACM Computing Surveys*, 54(2):1–36, 2021.

[110] Neil Robertson and Paul D. Seymour. Graph minors XIII. The disjoint paths problem. *Journal of Combinatorial Theory, Series B*, 63(1):65–110, 1995.

[111] Barna Saha and Lise Getoor. On maximum coverage in the streaming model & application to multi-topic blog-watch. In *Proceedings of the 9th SIAM International Conference on Data Mining (SDM 2009)*, pages 697–708. SIAM, 2009.

[112] Pascal Schweitzer. *Problems of unknown complexity: graph isomorphism and Ramsey theoretic numbers*. PhD thesis, Universität des Saarlandes, 2009.

[113] Daniel A Spielman and Shang-Hua Teng. Nearly linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems. *SIAM Journal on Matrix Analysis and Applications*, 35(3):835–885, 2014.

[114] Stergios Stergiou and Kostas Tsioutsiouliklis. Set cover at web scale. In *Proceedings of the 21th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD 2015)*, pages 1125–1133, 2015.

[115] Céline Maria Francisca Swennenhuis. *Fine-Grained Parameterized Complexity of Scheduling and Sequencing Problems*. PhD thesis, Eindhoven University of Technology, 2022.

[116] Jan Arne Telle. Complexity of domination-type problems in graphs. *Nordic Journal of Computing*, 1(1):157–171, 1994.

[117] Alan M. Turing. Systems of logic based on ordinals. *Proceedings of the London Mathematical Society, Series 2*, 45:161–228, 1939.

[118] Anthony Wren and Barbara M. Smith. Experiences with a crew scheduling system based on set covering. In *Proceedings of the 4th International Workshop on Computer-Aided Scheduling of Public Transport (CASPT 1988)*, pages 104–118. Springer, 1988.

[119] Mingyu Xiao and Hiroshi Nagamochi. Exact algorithms for maximum independent set. *Information and Computation*, 255:126–146, 2017.

[120] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. HUGE: an efficient and scalable subgraph enumeration system. In *Proceedings of the International Conference on Management of Data (SIGMOD 2021)*, pages 2049–2062, 2021.

[121] Meirav Zehavi, Fedor V. Fomin, Daniel Lokshtanov, Fahad Panolan, and Saket Saurabh. ETH-tight algorithms for long path and cycle on unit disk graphs. *Journal of Computational Geometry*, 12(2):126–148, 2021.

Samenvatting

Zoals de titel van dit proefschrift suggereert, bestuderen we hoe de complexiteit van problemen verandert als we er enkele beperkingen aan toevoegen. Om dit te bereiken, gebruiken we het raamwerk van de geparameteriseerde complexiteit. In tegenstelling tot klassieke complexiteit, waar we het aantal stappen dat een algoritme neemt uitdrukken in termen van de invoergrootte, drukken we bij geparameteriseerde complexiteit het aantal stappen uit in termen van de invoergrootte en een extra parameter. Deze parameter beschrijft meestal een eigenschap van de invoer: bijvoorbeeld, als we een graaf als deel van de invoer hebben, kunnen we verschillende graafparameters beschouwen, zoals de grootte van een vertex cover of de treewidth (die meet hoe "dicht" de graaf bij een boom staat). Geparameteriseerde complexiteit geeft ons een nauwkeuriger classificatie van NP-moeilijke problemen. De klasse FPT (fixed-parameter tractable) bevat problemen die we als eenvoudig beschouwen. De W-hiërarchie beschrijft moeilijkere geparameteriseerde problemen.

In het eerste deel van dit proefschrift bestuderen we hoe het toevoegen van structuur aan de invoer problemen eenvoudiger maakt. In Hoofdstuk 3 bestuderen we het (Counting) Subgraph Isomorphism probleem. Bij Subgraph Isomorphism probleem krijgen we twee grafen, P en G, en wordt ons gevraagd of G een subgraaf heeft die isomorf is met P. Dit probleem staat bekend als $W[1]$-moeilijk, zelfs wanneer P een clique is. In de telversie van dit probleem (Counting Subgraph Isomorphism) moeten we ook het aantal subgrafen van G berekenen die isomorf zijn met P. Om dit probleem eenvoudiger te maken, voegen we geometrische structuur toe aan de invoer: namelijk, we beperken de invoer tot unit disk grafen (intersectiegrafen van eenheidsdisks getekend in het vlak). We geven een algoritme voor het Counting Subgraph Isomorphism op unit disk grafen, samen met een ondergrens, die impliceert dat ons algoritme in zekere zin optimaal is. Op unit disk grafen met begrensde ply draait het algoritme in FPT-tijd (geparameteriseerd door het aantal knopen van P). Ons algoritme gebruikt dynamisch programmeren, waarbij we de geometrische structuur van de invoergrafen gebruiken om het aantal tabelgegevens te verminderen.

In Hoofdstuk 4 beschouwen we het Set Cover probleem, waarin we een verzameling U van gehele getallen krijgen, samen met een verzameling $\mathcal{S}$ van deelverzamelingen daarvan. Ons wordt gevraagd het kleinste aantal verzamelingen in $\mathcal{S}$ te vinden waarvan de vereniging gelijk is aan U. Het Set Cover probleem staat bekend als $W[2]$-compleet. We bestuderen ook een andere versie van het Set Cover probleem, genaamde Exact Cover, waarin we bovendien vereisen dat de verzamelingen in

de oplossing disjunct zijn. De beperking die we opleggen aan de invoer is dat elke verzameling in $\mathcal{S}$ een rekenkundige rij moet vormen. We verkrijgen FPT-algoritmen (geparameteriseerd door de oplossingsgrootte) voor deze beperkte versies van Set Cover en Exact Cover, met behulp van een recursieve aanpak, gecombineerd met getaltheoretische resultaten over rekenkundige rijen.

In het tweede deel van dit proefschrift is ons uitgangspunt het Flow (Stroming) probleem. Flow kan in polynomiale tijd worden opgelost met behulp van het welbekende Edmonds-Karp algoritme. Het toevoegen van twee natuurlijke beperkingen aan de oplossing maakt dit probleem echter aanzienlijk moeilijker. In Hoofdstuk 5 beschouwen we het All-or-Nothing Flow probleem, waarbij we bovendien vereisen dat elke pijl ofwel nul stroom of stroom gelijk aan zijn capaciteit heeft. We kijken naar dit probleem op vlakke (planaire) grafen, d.w.z., grafen die getekend kunnen worden op een vlak zonder kruisende kanten. Als parameter kijken we naar het aantal lagen van een tekening; dit wordt de outerplanariteit genoemd. Zelfs op planaire grafen blijkt All-or-Nothing Flow XNLP-moeilijk te zijn met de outerplanariteit als parameter. Dit impliceert dat het probleem $W[t]$-moeilijk is voor elke t. We gebruiken dit probleem als uitgangspunt om XNLP-moeilijkheid te bewijzen voor verschillende andere problemen op planaire grafen.

In Hoofdstuk 6 beginnen we bij het Min Cost Flow (Minimum Kosten Stroming) probleem en voegen we een eis van samenhang toe aan de oplossing. Namelijk, we beschouwen het Connected Flow probleem, waarin we bovendien vereisen dat de onderliggende graaf van de stroming verbonden wordt. We tonen aan dat deze extra beperking het probleem NP-moeilijk maakt. We presenteren een FPT-algoritme voor Connected Flow (geparameteriseerd door de vertex cover), evenals een XP-algoritme voor de parameterisatie door treewidth.

Curriculum vitae

Education

2020-2024	**Utrecht University** *Computer Science PhD*
2018-2020	**Freie Universität Berlin** *Mathematics MSc*
2015-2018	**University of Cambridge** *Mathematics BSc*

Selected Awards

2024	SOFSEM Best Paper Award *Joint paper with Jesper Nederlof*
2018	Berlin Mathematical School Scholarship *Full scholarship for Master studies*
2018	Rouse Ball Travelling Studentship in Mathematics *Support for mobility of students*
2015	Trinity College Overseas Bursary Scholarship *Full scholarship for Bachelor studies*