Parameterized Graph Problems: Counting, the Tutte Polynomial and Logarithmic Space

Geparameteriseerde Graafproblemen: Tellen, het Tutte Polynoom en Logaritmisch Geheugen
(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

maandag 21 oktober 2024 des ochtends te 10.15 uur

door

Isja Maria Emiel Mannens

geboren op 15 april 1996
te Almere

Promotor:
Prof. dr. H.L. Bodlaender

Copromotor:
Dr. J. Nederlof

Beoordelingscommissie:
Prof. dr. G.L.M. Cornelissen
Prof. dr. H. Dell
Prof. dr. J.A. Ellis-Monaghan
Dr. P. Kaski
Prof. dr. J.H.P. Kwisthout

Dit proefschrift werd (mede) mogelijk gemaakt met financiële steun van het project CRACKNP dat financiering heeft ontvangen van de European Research Council (ERC) als deel van het European Union's Horizon 2020 research and innovation programme (grant agreement No 853234) en van de Nederlandse Organisatie voor Wetenschappelijk Onderzoek als deel van project no. 613.009.031b.

If I must die,
you must live
to tell my story
to sell my things.

---

dr. Refaat Alareer

To those we've lost, may they stay with us
Tom
Sjoerd
Miel
Lian

Contents

Part I

Prologue

1 Introduction

My idea of a perfect school, Miss Honey, is one that has no children in it at all.

Miss Trunchbull, in Roald Dahl's Matilda

Imagine you find yourself in the following unenviable situation. You are a teacher at a school tasked with leading a school trip. Don't freak out, this is only a hypothetical! The last thing you want to happen is to leave a child behind somewhere, so you decide to periodically count the children.

When you first load the children on the bus, you decide to count them all one-by-one. This takes quite some time, especially as you have to start over a number of times as a result of children asking when the bus is leaving and you losing count as a result. Since the bus leaves five minutes late as a result, you decide to think of a better way to count the children.

At the first bathroom break you decide to count the number of rows of children, rather than the individual kids. Children are social creatures and as such they have filled up the various rows of the bus nicely. This method has sped up the counting process by a factor of 4, good enough for this trip.

At the next stop the children have caught on to your counting method and decide to mess with you by intentionally leaving varying numbers of empty seats in each row. Not to be outsmarted by these little twerps, you grab a notepad and keep track of the number of rows with 1 child, the number of rows with 2 children, etc. At the end you add them up, weighted by the number of children in the row, to find the total number of children. Turns out one of the kinds was still in the bathroom... Good thing you counted!

Upon your return you are commended on you great performance. Not a single child left behind, a first for the school! Perhaps you have done a bit too well, as you are asked to coordinate the next school trip, which is a joint trip with three other schools. Not wanting to turn down this 'honor' bestowed upon you by the principle, you begrudgingly accept the task.

This time the children are encouraged to mingle with the other schools and as a result they keep switching between the various buses. As the schedule is quite tight and the children cannot be expected to stand still for more than 5 seconds, counting the whole group outside of the buses is not an option. You decide to have each teacher count the number of children in their bus and then step outside to report

said number. You then add them up and check if all children are accounted for. Again you return home with all children accounted for.

What we have seen here is an example of how more sophisticated counting techniques allow us to both count faster and count in more complex situations. Of course, this example is still fairly simple and throughout this thesis we will see ways to count much more complicated structures using much more sophisticated techniques. We will spend the rest of this chapter introducing the major themes in this thesis on an intuitive level. While we will use some technical terms in this chapter, we do not expect the reader to know their (exact) meaning. Precise definitions of terms relevant for the rest of the thesis can be found in Chapter 2.

1.1 Algorithms and Complexity

Algorithms While the counting in the example above can still be done by a person, as the complexity and size of problems increases this eventually stops being feasible. It is at this point that we turn to computers to help lighten the load using specialized algorithms.

The term *algorithm* is derived from the name Muhammad ibn Musa al-Khwarizmi, a Persian polymath and a vastly underrated contender for the title of most influential mathematician of all time. Translations of his book on Indian arithmetic [103] introduced the Indian numerical system, which we still use today, to medieval Europe in the 12th century [143]. His book *Al-Jabr* [104] is arguably the founding text of the modern field of Algebra, being the first to teach Algebra for its own sake [88]. Hence it is not surprising that the field was named after this book, translations of which were used as the primary mathematics textbook from the 12th until the sixteenth century [85]. One could argue that the systematic solutions to linear and quadratic equations in the Al-Jabr were an example of ancient algorithms and that the study of algorithms dates back even further to at least the ancient Greeks, but the modern study of algorithms is considered to have started with the birth of computer science in the early 20th century. Many of the topics studied and concepts introduced during this time are still relevant today, for example Turing machines [147], Fast Fourier Transforms [60], the Simplex algorithm [47] or the Ford-Fulkerson algorithm [82], just to name a few.

An algorithm always has an input and an output. The input consists of some data describing a problem we want to solve. In the earlier example this would have been the group of children that needed to be counted. The output then consists of a/the solution to said problem, i.e. the number of children. When studying algorithms we want to determine how 'good' they are according to some useful metric. The most common way to measure the quality of an algorithm is to determine the amount of resources required to run the algorithm. Of course, the most valuable resource we have is time and it is not surprising that this is typically the metric we consider. Other common metrics include the amount of required memory, accuracy (in the case of approximation algorithms) and success probability (in the case of randomized algorithms).

The running time (and memory usage) of an algorithm almost always depend on the data it receives and the hardware it is run on. It therefore does not make sense to

measure running time in concrete numbers and computer scientists use other means instead. Practically oriented researchers will often use benchmarks, where the algorithm is run on some large collection of instances of the problem[1] and running time is measured for each instance. This produces a large set of statistics, which can be compared to the benchmarks of other algorithms. Here special care needs to be taken that the different algorithms are run the same hardware under the same circumstances. Theoretical computer scientists are often not interested in the observed running time and put more emphasis on obtaining worst-case guarantees. To this end, their goal is typically to determine how the running time grows as a function of the amount of data the algorithm receives. Constant factors are often ignored, i.e. they do not care whether the algorithm runs in time n or $2 \cdot n$. This function is referred to as the *complexity* or *time complexity* of the problem. As this thesis approaches algorithms from a theoretical angle, we will use the latter perspective. It is worth mentioning that this approach is not without its flaws. It is often the case that the best algorithm in a theoretical setting does not perform well or is difficult to implement in practice. In the other direction, algorithms that perform well in practice may appear to be incredibly slow from a theoretical perspective, due to its poor performance on some far fetched edge cases. Unfortunately this compromise is necessary for the majority of theoretical results, as formalizing what it means for an instance to be practical often proves to be quite difficult. There are some ways that researchers have attempted to bridge the gap, like average-case analysis or parameterized complexity, the latter of which is the subject of this thesis.

Theoretical computer scientists often group problems together, depending on their complexity, into so called *complexity* classes. The exact definition of such a class is beyond the scope of this informal introduction, but loosely speaking a class is often defined by a statement of the form 'all problems that can be solved in ... time (and using ... space), by a computer of type ...'. Notable examples include the class P of problems that can be solved in polynomial time on a *deterministic Turing machine*[2] and the class NP of problems that can be solved in polynomial time on a *non-deterministic Turing machine*. Problems can then be said to be $\mathbb{K}$-*hard* for a class $\mathbb{K}$, if they are at least as hard to solve as any problem in the class $\mathbb{K}$. If a problem is also a member of $\mathbb{K}$, in addition to being $\mathbb{K}$-hard, we say it is $\mathbb{K}$-*complete*. Showing hardness for a difficult class is generally considered a good indication that a problem cannot be solved efficiently by any algorithm. In the field of algorithmic complexity, the goal is usually to either find a fast algorithm or show hardness for a difficult class. In the ideal case we find an algorithm and a hardness result that in some sense match. For example we might find an algorithm that runs in exponential time and also show hardness for the class of problems that can be solved in exponential time.

Parameterized complexity It turns out that considering only the input size when analyzing an algorithm is quite limiting. There are often other properties of the data that can be exploited to produce significantly faster algorithms. This is the

[1] If we stretch our example to fit this notion, one can think of this as making a teacher count the number of children in a large number of buses and keeping track of the time needed for each bus.

[2] One can think of a Turing machine as just a computer and a non-deterministic Turing machine as a computer that can make choices that are not predetermined. See Chapter 7 for a technical definition.

main motivation behind the field of *parameterized complexity*. In this setting, we consider a problem in conjunction with one or more parameters of that problem. For example, we saw earlier that it is easier to count children on a bus, if they do not leave empty seats in any row of seats. We could consider the maximum number of empty seats per row as a parameter of the problem. Another example is found in the navigation of the bus. If the road network that the bus is navigating has a few central intersections that all roads radiate out of, then it is very easy to determine the fastest route from A to B, you have to go via the central hub. The number of such intersections[3] could be a parameter of the route finding problem.

When analyzing the complexity of an algorithm, we typically assume that the parameters are small relative to the input size. This means that we prioritize a good dependency on the input size over the dependency on the parameter. For example, for a parameter k and input size n a running time $2^k n^2$ is generally considered good, while $k^2 2^n$ is not.

In a sense, parameterized complexity forms somewhat of a middle ground between theory and practice, as most practical situations concern highly structured data. Parameterized complexity still provides the same theoretical guarantees, while also taking this extra structure into account.

While examples of parameterized algorithms date back to the 1980's [102], the systematic exploration of parameterized complexity only started in 1990's, after it was proposed by Downey and Fellows [63]. It has since turned into a rapidly evolving field with interest in the field showing no sign of slowing down. In the now standard textbook for the field *Parameterized Algorithms* [56] (colloquially referred to as the 'blue book'), the authors mention that at the time of writing Google Scholar produces more than 4.000 papers containing the term 'fixed-parameter tractable'. In the 9 years since this number has grown to over 21.000!

Parameterized complexity has received interest from all kinds of fields within the study of algorithms, including but not limited to approximation algorithms [77], streaming algorithms [44], distributed computing [142], parallel algorithms [1], databases [100], computational biology [16] and robotics [70]. Clearly, parameterized complexity has great potential and this is increasingly being recognized by a diverse subset of the scientific community.

A brief history of counting complexity The study of counting complexity arguably began with the introduction of the class #P by Valiant [153]. In this paper Valiant describes a computer that counts things as "*magically* printing the number of solutions on a *special* tape"[4]. In a follow up paper [151], Valiant gives a list of 14 #P-complete problems. In this paper he notes that some of these problems, like counting perfect matchings or directed trees, have easy decision versions. Indeed, it is clearly a trivial task to find **any** directed tree in a given graph, but counting them all is quite difficult. This divergence between easy decision problems and difficult counting versions has proven to be a popular theme in counting complexity to this day. Recent examples

[3] Commonly referred to as the *vertex cover number* of the network.

[4] The exact phrasing is as follows: "A counting Turing machine is a standard non-deterministic TM with an auxiliary output device that (magically) prints in binary notation on a special tape the number of accepting computations induced by the input."

of this include the parameterized complexity of counting perfect matchings [55] and counting Hamiltonian cycles parameterized by pathwidth [53, 57].

In the early days of counting complexity, a big question was how difficult counting problems are, compared to decision problems. Clearly counting the number of solutions to a problem is more difficult than deciding whether that same problem even has any solutions. One might wonder though, could the counting classes like #P also capture more difficult decision classes than the ones they are based[5] on? This question was answered in the affirmative by Toda [146], about a decade after Valiant first introduced #P. In particular, Toda showed that #P is at least as hard as the entire *polynomial hierarchy*, another generalization of the class NP.

Flum and Grohe [80] later initiated the study of the hierarchy #W[1], #W[2], ... of classes of parameterized counting problems. These classes play a similar role as #P, in the sense that they form the counting counterparts to typically difficult classes, but in this case for parameterized problems. Flum an Grohe showed that in the parameterized setting there are problems, like counting cycles or paths of length k, whose counting version is #W[1]-complete, while the decision version is easy to solve. While in the classical setting many natural counting problems are either easy or #P-complete, in the parameterized setting there are a lot of problems that are #W[t]-hard for some t, but are not known to be contained in any such class. In Chapter 7 we introduce the new classes #XLP and #XALP, for which a number of these problems are complete.

Various different frameworks, i.e. abstract problems that generalize many natural problems, for counting problems have been developed over the years. Three of the most well-studied frameworks are *Constraint Satisfaction Problems* [38, 31, 32, 48, 39, 66, 69, 34], *Graph Homomorphism Problems* [68, 30, 67, 94, 35, 33, 98] and *Holant Problems* [37, 38, 40, 36, 152, 50]. We will encounter all three frameworks at some point in this thesis. One reason why such frameworks are interesting is because they allow for a systematic approach to the complexity of problems that fall under the frameworks. The ultimate goal of such a framework is to find a full classification of the complexity of its problems, however even finding a classification for a restricted version of the framework can be a major result (see [49] for a number of examples). The fact that this is even possible is noteworthy, as this is not the case in general, for example if P $\neq$ NP, then there are problems in NP $\setminus$ P that are not NP-hard [122].

The graph coloring problem we study in Chapter 3, as well as many evaluations of the Tutte polynomial (see Chapter 5 or Section 2.6) can be seen as special cases of counting graph homomorphisms [92]. The complexity bounds we find in Chapter 3 use a version of #CSP as the starting point of the reduction. In Chapter 7 we show #XLP- and #XALP-hardness[6] for two parameterizations of a problem that falls under the Holant framework, using various other problems from the framework in the reduction.

ETH Bounds vs W[1] vs XNLP In this thesis we give different types of difficulty guarantees for various problems. Here will discuss how these notions relate to one

[5] In this case #P is based on the class NP.

[6] see Section 2.5 for definitions of these classes.

another. For the purpose of this discussion we will assume the reader has some familiarity with the various complexity classes an hypotheses mentioned. For more technical details, we refer the reader to Sections 2.2, 2.3 and 2.5.

Our two main ways to give an indication of the difficulty of a problem are *hardness results* based on complexity classes and *fine-grained bounds* based on difficulty hypotheses. When using the term *fine-grained*, we refer to explicit running time bounds and will do so throughout the thesis. We generally prefer fine-grained bounds over hardness results as they give a tangible lower bound on the potential running time of algorithms, rather than a broad statement of difficulty. However, in some cases it can be difficult to obtain a fine-grained bound. In these cases it will still prove very insightful to show hardness for some complexity class.

When showing $\mathbb{K}$-hardness for some complexity class $\mathbb{K}$, the rule of thumb is "the bigger the class, the more difficult the problem". The (parameterized) classes we consider can be ordered as follows

$$\text{FPT} \subseteq \text{W}[1] \subseteq \cdots \subseteq \text{W}[t] \subseteq \cdots \subseteq \text{XNLP} \subseteq \text{XALP} \subseteq \text{XP} \subseteq \text{para-NP}$$

with the same inclusions holding for the counting versions. We also consider the class of NP, which is not comparable to these parameterized classes, as it consists of classical (i.e. non-parameterized) problems.

One might wonder why so many classes are necessary. After all, if a problem is unlikely to be solvable in a reasonable time, why should we care **how** unreasonable the running time is? In other words, is there really a meaningful difference between W[1] and W[2], or even W[1] and XNLP? Unfortunately, we are unable to conclusively answer these questions. While some of these inclusions can be shown to be strict, for example FPT $\subsetneq$ XP (see also [62, Proposition 27.1.1]), for most of these inclusions it is unclear whether they are strict or not, although most assume that they are.

One can however use these classes to show that some problems are meaningfully more difficult than others, assuming that the relevant inclusions are in fact strict. This is why the notion of $\mathbb{K}$-*completeness* is so important, as it indicates that a problem is as hard **and** no harder than the problems in $\mathbb{K}$. This is actually one of the main motivations behind the class XNLP, as it provides completeness for many problems, that previously where only known to be W[1]- or W[2]-hard, but did not have completeness results yet.

We will often use fine-grained bounds to refine the complexity of problems in one of these classes. In particular we will often give lower bounds for problems in FPT, bounding dependency on the parameter in the running time. As FPT is the smallest class in this chain, showing hardness for FPT would not give much information about a problem and thus fine-grained bounds are a big improvement. In contrast, for problems outside of FPT we will often settle for W[1]- or XNLP-hardness, as these are bigger classes and thus give more valuable hardness results. When giving fine-grained bounds, we use the *(Strong) Exponential Time Hypothesis* (ETH/SETH), which postulates a fine-grained bound on a problem called the *Satisfiability problem*. Lokshtanov, Marx and Saurabh [124, 123] first used this method to give fine-grained bounds for parameterized problems. They gave lower bounds for a number of fundamental problems, most of which were parameterized by treewidth, that match

the running time of existing algorithms. ETH (and SETH) bounds represent a shift in perspective, as they give concrete running time bounds, rather than the abstract notion of hardness found in the study of complexity classes. Hardness results are useful to get a general idea of the difficulty of a problem and to develop a deeper understanding of the broader landscape by grouping similarly difficult problems together in classes. In practice, however, fine-grained results are often preferred as they don't just tell us whether a *feasible* algorithm exists, but also whether existing algorithms are (broadly) optimal or can potentially be further improved.

1.2 Graphs and Graph Polynomials

Graphs So far we have only discussed algorithmic problems in the abstract. While there are many different types of algorithmic problem, most of the problems we consider in this thesis share one common trait, they are defined on *graphs*.

Graphs are one of the most studied objects in both Discrete Mathematics and Theoretical Computer Science. Simply put, a graph is a collection of points (often called *vertices* or *nodes*) and lines (often called *edges* or *arcs*). This simplicity makes graphs very versatile, especially when you consider the many variations there are on the concept of graphs. One could decide to make the edges directed, assign weights to edges or vertices, give certain edges or vertices special labels and so on.

Graph theory is generally considered to have started in the 18th century, with the paper on the seven bridges of Königsberg by Leonard Euler [75]. A century later the mathematician Francis Guthrie posed the *four color problem* [157], now known as the four color **theorem**, as it was proven in 1976 by Appel and Haken [6, 7] using the first ever computer assisted proof. A cursory glance at the bibliography of this thesis will provide many more examples of works combining the fields of graph theory and computer science. Indeed graphs have proven to be a popular model for problems in theoretical computer science.

Clearly any problem relating to a network of some kind can (typically) be phrased as a graph problem, i.e. the vertices represent the objects in the network and the edges represent the relations between them. There are however a lot of other types of problems that can be seen as a graph problem is less obvious ways. For example, the problem of having to assign classes to time slots can be modeled as a graph-coloring problem, the same problem that appeared in Guthrie's four color problem. In this problem we are given a graph and a set of colors, with the goal of assigning colors to vertices in such a way that adjacent vertices are assigned different colors. If we create a vertex for each class and add edges between vertices (classes) that should not be scheduled simultaneously, we can then 'color' the vertices with time slots to find a valid schedule.

When analyzing algorithms for graphs, the number of vertices is often used as a proxy for the size of the graph. However, since a graph with n vertices can have n^2 edges, a more accurate measure would be the number of vertices AND edges. For some algorithms this distinction makes a big difference and as such one might see the running time expressed in terms of the number of edges. In many cases, however, it suffices to use the number of vertices. There are also a lot of structural parameters of graphs that get considered in parameterized complexity analyses. The

structure of the graph can have major implications for the efficiency of algorithms, especially for problems that are effected by local structures, like the coloring problem mentioned earlier. Some structural parameters measure how similar the graph is to some canonically easy example (like a path- or tree-shaped graph), others measure the size of some substructure in the graph that might form an obstruction for the algorithm (for example, a really dense cluster of edges). In Section 2.4 we will discuss the specific parameters used in this thesis, in more detail.

Graph Parameters When studying algorithmic problems on graphs from the perspective of parameterized complexity, one is often concerned with graph parameters. A graph parameter is any numerical property of the graph, like the number of points, the number of connected components, or more complex properties like the treewidth (see Definition 2.4.1). These kinds of parameters are one of the most used used types in parameterized complexity and will be the involved in most of the results in this thesis. For an overview of the graph parameters we use, see Section 2.4.

Over the years many different types of graph parameters have been developed. To get an idea of breadth of parameters and their relations we refer the reader to the Graph Parameter Hierarchy project[7]. For now we will discuss some of the broader categories of parameters[8].

Some graph parameters are defined by the optimal solution to some problem. This could be some substructure in the graph, like the smallest vertex cover [24], smallest dominating set [130] or the largest independent set [97], or it could be some external object like a proper graph coloring with a minimal number of colors [99]. The graph parameter then consists of the value by which optimality is measured, i.e. the size of the independent set or the number of colors.

Another category is that defined by the minimal number of vertices one needs to remove to obtain a graph of some form [101]. Examples of target forms include planar graphs [129], bipartite graphs [108] and cliques [2].

Finally a very broad category is that of width parameters. Most graph parameters we consider will fall in this category. Very generally a width parameter is typically defined by some notion of decomposition of the graph. A decomposition, in this context, can be seen as some additional structure placed on top of the graph. Some notion of width is defined for a decomposition and the width of the graph is then defined as the minimum width over all decompositions. A decomposition might consist of a partition of the vertices (tree partition width [141][9]), a collection of vertex subsets that cover the graph (treewidth [12, 139] [10]), a sequence of contractions that turn the graph into a single vertex (twinwidth [27]), an ordering of the vertices (cutwidth [45]), or a number of other types of structures. Width measures can vary from the size of the sets used to cover the graph (treewidth, pathwidth) to the number of edges crossing any cut of the ordering (cutwidth).

[7] https://gitlab.com/gruenwald/parameter-hierarchy

[8] Note that this is by no means an exhaustive list.

[9] The parameter was originally introduced under the name 'strong tree-width'.

[10] The parameter was originally introduced under the name 'dimension'.

The Tutte Polynomial One of the central objects in this thesis and the subject of Part II is the Tutte polynomial ($T(G;x,y)$). It was named after the English/Canadian mathematician and codebreaker W.T. Tutte, who first introduced it as the *dichromate*[149, 148, 150]. It is sometimes referred to as the Tutte-Whitney polynomial as, in Tutte's words, "[Hassler Whitney] knew and used analogous coefficients without bothering to affix them to two variables". The Tutte polynomial of a graph is a polynomial in two variables x and y. For a formal introduction to the object, see Section 2.6.

The Tutte polynomial has generated interest in a wide variety of fields, due to the fact that it generalizes many other properties of graphs (see for example the discussion in [106]). Along the line $y = 0$, the Tutte polynomial specializes to the chromatic polynomial of the graph. Along $x = 0$ we find the flow polynomial, which for planar graphs is equivalent to the chromatic polynomial of the dual graph. Continuing with planar graphs, the Tutte polynomial of a planar graph at $xy = 1$ specializes to the Jones polynomial of the associated alternating knot. On the line $x = 1$ we find the reliability polynomial, which captures the resilience of a network against failures in connections in the network (i.e. edges being deleted independently, uniformly at random). It also finds applications in statistical physics by specializing to the partition function of the q-state Potts model [138, 158, 159] (itself a generalization of the Ising model) at $(x-1)(y-1) = q$.

Finally there are a number of individual points that capture interesting graph properties. For example $T(G;2,1)$ gives the number of forests in G, $T(G;2,0)$ gives the number of acyclic orientations of G and $T(G;1,2)$ gives the number of edge subsets with the same number of connected components as G. One can even compute the number of ways to tile a $4m$ by $4n$ grid with T-shape tetrominoes, as $2T(G;3,3)$.

In this thesis we will focus on the computational complexity of computing the Tutte polynomial at various points on a two-dimensional plane. The first, full hardness dichotomy for this problem was given by Jaeger, Vertigan and Welsh [106], building on a series of previous results. In this paper, the authors show that, with the exception of a small number of points, for almost all points (x,y), $T(G;x,y)$ is #P-hard. Many of the techniques that appear in the paper have since been used to prove further refinements, such as the result by Brand, Dell and Roth [28], who show that any #P-hard point cannot even be computed in subexponential time, assuming the #ETH. Even for planar graphs, the Tutte polynomial is #P-hard to compute at almost all points [154]. Only the curve corresponding to the Ising model (called 'H_2') becomes easy in this setting, using techniques developed by Kasteleyn [115].

Approximating the Tutte polynomial has also generated considerable interest. As was the case for the planar setting, the curve H_2 allows for some positive results here. In particular Jerrum and Sinclair [112] have given an FPRAS[11] for this curve. For most other points such results are very unlikely, as Goldberg and Jerrum [95] have showed that for almost all points (x,y) with $x,y < -1$ no such scheme can exist.

In terms of parameterized complexity there has been little study on the Tutte polynomial on graphs. Aside from the work featured in this thesis (see Chapter 5),

[11] A *Fully Polynomial-time Randomized Approximation Scheme*, which is a collections of randomized algorithms that approximate the value to some ratio ε in time polynomial in both the input size n and ε^{-1}.

to our knowledge there have only been a few other paper combining parameterized complexity with the Tutte polynomial [140, 4, 134, 8, 125, 93]. In one of them Roth et al. [140] consider a version of the problem where the polynomial itself is parameterized. In the others [4] give algorithms computing the Tutte polynomial for graphs of bounded treewidth or cliquewidth. Seeing as there is clearly interest in computing the Tutte polynomial for restricted graph classes, it makes perfect sense to ask for complexity lower bounds and more broadly for a full classification of the parameterized complexity. We think this is currently a major blind spot of the field and warrants additional research.

Some research has also been done towards determining the complexity of computing the Tutte polynomial modulo some prime. In his thesis Annan [5] discusses some cases for which hardness can be shown. Goodall [96] later showed that the complex valued easy points can be translated to the modular setting. A natural conjecture then becomes that all other points are hard, but to our knowledge no further progress towards this goal has been made.

While it is beyond the scope of this thesis, it is worth mentioning that the Tutte polynomial can also be extended to matroids. In this setting there has also been some fine-grained analysis of the parameterized complexity of the Tutte polynomial [15].

For a more extensive overview of research on the Tutte polynomial, we refer the reader to this book [72] or this survey [73].

1.3 Dynamic Programming and the Rank-Based Approach

Dynamic Programming *Dynamic programming* (DP) is one of the most used techniques in algorithm design and the basis for many of the results in this thesis. The technique was first developed by Richard Bellman [11] while working at the RAND corporation. A firsthand account of its discovery can be read in a publication by Dreyfus [65], containing excerpts from Bellman's autobiography. The name dynamic programming was largely chosen to fool a politician responsible for Bellman's funding, who was not particularly fond of fundamental research. In his own words, Bellman chose the name *"to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation."*

Since its invention, dynamic programming has become one of the most ubiquitous techniques in algorithm design (see [145] for a recent survey). The basic idea behind dynamic programming is to solve small parts of the problem and save the solutions in a table. The algorithm can then use these saved solutions to compute solutions for larger parts of the problem, until finally a solution for the whole problem is found.

A simple example that often gets used to illustrate this technique is the computation of Fibonacci numbers. The n-th Fibonacci number F_n is defined recursively as $F_n = F_{n-1} + F_{n-2}$, with $F_0 = 0$ and $F_1 = 1$. If we were to feed this definition to a computer with no further instructions, it try to compute it recursively. This would mean that to compute one value, it tries to compute two previous values, each of which requires two more values, etc. The number of recursive calls then grows exponentially and we end up taking 2^n time to compute F_n. With dynamic programming

we essentially compute the number in the same way a human would, by starting at $F_2 = F_1 + F_0 = 1$ and then computing the numbers $F_3, \ldots, F_n$ in order, while remembering each number to be used in subsequent computations. This method only takes n time, a dramatic improvement over the recursive method.

Especially in a parameterized setting, DP algorithms often turn out to be very efficient and even optimal for many problems [99, 123]. One reason for this, which we will see in some of our results, is that small structural parameters often allow us to partition our problem into small parts. This then allows us to solve the problem one part at a time.

One might expect dynamic programming to require a lot of memory, but in the parameterized setting this is often not an issue. A lot of DP tables can be divided up into layers, where the computation of a cell in one layer only requires knowledge of the cells in the previous layer. For many parameterized DP algorithms the size of these layers is bounded by some function of the parameter. In practice, this means we only actually need to keep the most recent layer of the DP table in storage and can delete the rest.

The Rank-Based Approach One of the most important techniques used in this thesis is the *rank-based approach*, first introduced in [18]. A formal description can be found in Section 2.8, but broadly speaking, the rank-based approach uses some problem specific low-rank matrix M to compress the size of a dynamic programming table. This then allows for the corresponding dynamic programming algorithm to be performed much faster.

While the technique existed before this work, there are two new, significant insights that were gained during the course of the research featured in this thesis.

The first insight is that it can often be beneficial to phrase the approach entirely in terms of linear algebra. When first introduced for decision problems, the approach was defined using representative sets. This perspective is helpful because it gives a good intuition for why the compressed information still produces the correct outcome, but it can be a bit involved to define and argue about. When we replace these representative sets with vectors, the connection with the rank of M becomes a lot clearer and we can make much cleaner, succinct statements, using existing concepts and notation from linear algebra. Both perspectives will be formally introduced in Section 2.8.

This linear algebra perspective enabled a second insight, namely that the approach can be subtly adapted to work for counting problems. One essentially replaces the $0,1$-vectors of the decision version with arbitrary real-valued vectors[12]. The connection to the rank of M becomes even clearer here, as the notion of representation can be expressed as a very simple equality involving only matrix-vector multiplications:

$$Ma = Mb.$$

Representation for decision problems can actually be phrased in the same manner, but requires operations to be performed over the Boolean semiring. Again, for more information and formal definitions, see Section 2.8.

[12] However, the entries will typically be integers.

The rank-based approach has proven to be a powerful tool and a fruitful source of new results. For many problems, the algorithms obtained from this approach prove to be optimal, with the matrix M also playing a role in the accompanying lower bound. This can often be explained by the fact that M, in some sense, encodes the transferal of information at certain information bottlenecks in the problem instance. The rank of M then precisely captures the amount of information that can pass through the bottleneck, but also the amount of information you need to distinguish two meaningfully different states on one side of the bottleneck. In terms of information complexity, the matrix M therefore often captures the difficulty of the problem perfectly and it only remains to show that one can match this in terms of runtime complexity.

1.4 Our Goals and Contributions

The overarching goal of this thesis is to further our understanding of parameterized counting complexity. Almost all of our results concern the parameterized complexity of counting problems or problems closely related to counting, like the Tutte polynomial. One of the main questions we will ask is how the counting versions of problems relate to the decision versions. We will find that for some problems, like counting colorings, our results show that there is a difference in complexity with the decision version [109]. In other cases (for example in Chapter 7) we adapt hardness proofs for decision problems to show that the counting version is hard for the analogous counting class.

Of the results we present that are not related to counting problems, some relate to the classes XNLP and XALP. One of our goals is to build on recent developments around these classes [71, 17, 21, 23]. We do this in two ways. The first way is by providing more problems that are complete for these classes. The second way is by initiating the study of the counting versions of these classes.

Another goal of this thesis is to initiate the study of the parameterized complexity of the Tutte polynomial, when parameterizing by graph parameters. We note that the complexity of a version of the Tutte polynomial where the polynomial itself is parameterized has already been studied of [140]. Such results build on the seminal #P-hardness dichotomy by Jaeger, Vertigan and Welsh [106], as well as subsequent fine-grained classifications [61, 28].

Finally we aim to showcase the potential of the rank-based approach for the counting paradigm. It was initially introduced [18] for decision problems. The phrasing used in the original definition does not lend itself well to counting problems. By rephrasing the method using more linear algebra flavored language, we aim to both deepen the understanding of the original method, as well as further developing it to work for counting problems.

Outline of this Thesis This thesis consists of three parts. **Part I** (which you are currently reading), is an introductory part containing (technical) background information and philosophical discussions about the topics of the other two parts.

Part II works towards a parameterized complexity classification for computing the Tutte polynomial. **Chapter 3** gives SETH tight bounds on the complexity

of counting (list) colorings modulo a prime, parameterized by cutwidth. These bounds then get extended to counting connected edgesets. **Chapter 4** gives ETH tight bounds on the complexity of counting forests, parameterized by cutwidth. **Chapter 5** uses the results of the other two chapters in **Part II** to give ETH/SETH bounds on the complexity of computing the Tutte polynomial on integer points, parameterized by cutwidth, pathwidth and treewidth.

Part III is centered around the complexity classes XNLP and XALP. **Chapter 6** gives hardness results, including XNLP/XALP-hardness, for a number of variations on the Multicommodity Flow problem. In **Chapter 7** we introduce #XLP and #XALP, which can be seen as counting versions of XNLP and XALP. We show #XLP/#XALP-hardness for a number of problems.

List of Publications Chapters 3 to 7 are based on various papers, from which they differ only slightly. The main changes will be in the introduction and conclusion sections. Chapters 4 and 5 were published as one paper, but were split into two chapters to create a clearer structure for this thesis. The following papers were adapted to produce this thesis.

- (Chapter 3) *Tight bounds for counting colorings and connected edge sets parameterized by cutwidth*, by Carla Groenland, Isja Mannens, Jesper Nederlof, and Krisztina Szilágyi. In Proceedings of the 39th International Symposium on Theoretical Aspects of Computer Science, STACS 2022, pages 36:1–36:20, 2022. ([99])
- (Chapters 4 and 5) *A fine-grained classification of the complexity of evaluating the Tutte polynomial on integer points parameterized by treewidth and cutwidth*, by Isja Mannens and Jesper Nederlof. In Proceedings of the 31st Annual European Symposium on Algorithms, ESA 2023, volume 274 of Leibniz International Proceedings in Informatics (LIPIcs), pages 82:1–82:17, 2023. ([126])
- (Chapter 6) *The parameterised complexity of integer multicommodity flow*, by Hans L. Bodlaender, Isja Mannens, Jelle J. Oostveen, Sukanya Pandey, and Erik Jan van Leeuwen. In Proceedings of the 18th International Symposium on Parameterized and Exact Computation, IPEC 2023, volume 285 of Leibniz International Proceedings in Informatics (LIPIcs), pages 6:1–6:19, 2023. ([24])
- (Chapter 7) *Problems Complete for #XLP and #XALP*, by Radu Curticapean Lars Jaffke, Paloma T. Lima, and Isja Mannens. Unpublished.

2 Preliminaries

It's the job that's never started as takes longest to finish, as my old gaffer used to say.

Samwise Gamgee

This chapter contains the basic notions and other background knowledge necessary to understand this thesis. The sections will loosely be ordered from general knowledge to more specialized topics. We will assume knowledge of basic concepts in graph theory and linear algebra, but will nevertheless discuss some more specialized concepts from these fields. We start by fixing the notation we will use throughout the thesis.

2.1 Basic Notation

We write $\mathbb{N} = \{1, 2, \dots\}$ for the set of natural numbers, $\mathbb{Z}$ for the set of integers and $\mathbb{Z}_{\geq 0} = \{0, 1, 2, \dots\}$ for the set of non-negative integers. For integers $a, b \in \mathbb{Z}$, we write $[a, b] = \{a, a+1, \dots, b\}$ for the integers between a and b, and for a natural number n we short-cut $[n] = [1, n] = \{1, \dots, n\}$. Throughout the chapter, p will denote a prime number and $\mathbb{F}_p$ the finite field of order p. We will use $a \equiv_p b$ to denote that a and b are congruent modulo p, i.e. that p divides $a - b$.

For two sets A and B, we denote by $A \triangle B = (A \setminus B) \cup (B \setminus A)$ the symmetric difference of the sets. For a set $S \subseteq A$ and a function $f : A \to B$, we will use the abbreviation $f(A) = \{f(a) : a \in A\}$. The function $f|_S : S \to A$ is defined as $f|_S(a) = f(a)$ for all $a \in S$. For a function $f : A \to \mathbb{Z}$, we define the support of f as the set $\operatorname{supp}(f) = \{a \in A : f(a) \neq 0\}$.

Given a graph $G = (V, E)$ and vertices $u, v \in V$, we denote arcs and edges as vw (an arc from v to w, or an edge with v and w as endpoints). By $N(v)$ we denote the open neighborhood of v, i.e. the set of all vertices adjacent to v. We write $\delta(v)$ for the set of edges incident to v. We write $V(G)$ for the vertex set of the graph G. We will often use the shorthand $n = |V(G)|$ and $m = |E(G)|$. When considering a parameter k of different graphs G and H, we will differentiate them by writing $k(G)$ as the parameter k of graph G. When the relevant graph is clear from context, we simply write k.

Given a matrix $M \in \mathbb{C}^{m \times n}$, we write $M[i, j]$ for the entry of M at row i and column j. We write $M^\top$ for the transpose of M, i.e. $M^\top[i, j] = M[j, i]$. For a vector $a \in \mathbb{C}^n$ we

write the right hand product of M and a as Ma. Similarly for $b \in \mathbb{C}^m$ we write the left hand product of b and M as $b^\top M$.

2.2 Computational Complexity

2.2.1 Computational Problems

The basis of computational complexity is the notion of a *compuational problem*.

Definition 2.2.1. *A* computational problem Π *consists a set* $\mathcal{I}$ *of* instances *and a mapping* $f : \mathcal{I} \to \mathcal{S}$ *that maps an instance to an accepted solution (or a set of such solutions). When multiple computational problems are considered, we will indicate the corresponding elements with an index, i.e.* $\mathcal{I}_\Pi, \mathcal{S}_\Pi, f_\Pi$.

The instance is assumed to be given as input and the solution is unknown. The goal of an algorithm is to find a solution for a given instance, i.e. to compute the function f on the given I.

There are various common types of computational problem. Perhaps the most commonly examined are *decision problems*, where the set $\mathcal{S}$ of solutions is simply given by $\mathcal{S} = \{\text{True}, \text{False}\}$. For example the problem might consist of determining some property of the instance, like connectivity or 3-colorability of a graph. In the latter case one might also ask for a witness, i.e. a valid 3-coloring of the given graph. In this case we have turned the decision problem into a *search problem*, where the solution set $\mathcal{S}$ is some arbitrary set, but often this is the set of potential witnesses to some decision problem. Note that for a search problem, the function f_Π outputs the set of all accepted solutions, but we only require an algorithm to find one of these solutions. An *optimization problem* is a specific type of search problem, where we are interested in finding the solution that maximizes some objective function. If instead of finding a specific solution, we are interested in finding the total number of solutions, then we are dealing with a *counting problem*. In this case the solution set is typically given by the non-negative integers, i.e. $\mathcal{S} = \mathbb{Z}_{\geq 0}$, although one might also be interested in weighted counting, in which case the solution set could be $\mathbb{R}$ or even some abstract ring. For example one might count over the finite field $\mathbb{F}_p$, i.e. modulo some prime p. In this thesis we will primarily consider decision problems and counting problems, although some of our algorithms for decision problems can easily be made constructive and can thus solve the corresponding search problem as well. We denote the counting version of a problem by using the prefix # (e.g. #SAT, #CSP) and the counting modulo p version of by using $\#_p$ (e.g. $\#_p$SAT, $\#_p$CSP).

We will present computational problems in the following format.

q-coloring **Input:** A graph G **Question:** In how many ways can we color $V(G)$ with q colors, such that adjacent vertices receive different colors?

When analysing an algorithm or an algorithmic problem we are primarily interested in the time it takes to execute an algorithm and secondarily in the amount of

computational space required. We will express these in terms of the size of the problem instance (often denoted as 'n') and one or more other parameters of the input instance. For more details on these additional parameters, see Section 2.3. Note that the input size can be defined in various ways, depending on the nature of the problem we examine. There is usually a natural choice, such as the number of vertices in a graph, but sometimes one might choose to use a different property as its size, such as the number of edges of the graph. Whenever relevant, we will specify how we define the size of the problem instance.

2.2.2 Big-O Notation

In the field of Theoretical Computer Science, researchers are often not interested in the exact runtime or space requirements of an algorithm, as they can depend on things like hardware and the specifics of the implementation. Instead we will examine how the time and space requirements of an algorithm are affected when the various parameters of the problem are increased. For this we employ *big O-notation*.

Definition 2.2.2. *We say that a function $f : \mathbb{Z}_{\geq 0} \to \mathbb{Z}_{\geq 0}$ is $O(g(k))$, if there exists constants $C > 0$ and K such that $f(k) \leq C \cdot g(k)$ for all $k \geq K$.*

Note that the constants C and K cannot be dependent on other properties of the instance. When it is unclear what parameters are in play, we indicate them as a subscript, i.e. $O_k(f(k))$. While $O(f(k))$ is not really a number or function, we will often treat it as such in notation. For example we will write

$$g(k) = O(f(k))$$

to indicate that there is some $C > 0$ such that $g(k) \leq C \cdot f(k)$ for all $k \geq 0$. Although this is standard notation, it is somewhat of an abuse of notation, as we are not really dealing with an equality here. We will also use this notation to relate different big O statements to one another. We write

$$O(g(k)) = O(f(k))$$

to indicate that $h(k) = O(g(k))$ implies $h(k) = O(f(k))$ for any h. This notation will allow for derivations that might look as follows.

$$\begin{aligned} g(k,n) &= O(2^k k^2 n^3 / \log(n)) \\ &= O(2^{2k} n^3 / \log(n)) \\ &= O(2^{2k} n^3). \end{aligned}$$

Note that the use of the equality sign is asymmetrical in this example, as we do not have $O(2^{2k}n^3) = O(2^{2k}n^3/\log(n))$. Because of this, there are some that argue for the notation $g(k) \in O(f(k))$ and treating $O(f(k))$ as a set, however we will stick with the standard notation in this thesis.

A related notion is that of *little o-notation*.

Definition 2.2.3. *We say that $g(k) = o(f(k))$, if $\lim_{k\to\infty} \frac{g(k)}{f(k)} = 0$.*

We use the same standard abuse of notation for little o as we do for big O. We can think of the relation between big O-notation and little o-notation as being analogous to the relation between $\leq$ and $<$.

2.2.3 Complexity Classes and Hardness Conjectures

Computational problems are grouped together into *complexity classes* based on how difficult they are to solve, in terms of time (and sometimes space) complexity. A typical example is the class P, which contains all problems that can be solved in polynomial time. This class is often paired with NP, which contains all problems that can be solved in polynomial time on a non-deterministic Turing machine.

Definition 2.2.4. *A* non-deterministic Turing machine *is a Turing machine that can have multiple valid transitions from a single given state to a number of different new states.*

Non-determinism in a Turing machine could, for example, be the result of some random process or external influence. A useful way to think of a non-deterministic Turing machine is as a tree where each state with a choice branches into all potential states that can be transitioned into. An accepting run then consists of a path from the starting state to one of the accepting states. We will therefore often refer to *accepting paths* of a non-deterministic Turing machine. It is a major open problem to determine whether P and NP are equal, although the general consensus is that they assumed to be different.

A typical way of defining a class of counting problems is as the class of problems that count the number of accepting path in a non-deterministic Turing machine. For example #P is the class of problems that count the number of accepting paths of a polynomial time non-deterministic Turing machine.

One of the main uses of these classes is to show *hardness* for such a class. In order to define hardness, we first need to consider the following notion.

Definition 2.2.5. *Let* Γ *and* Π *be computational problems. A* reduction *from* Γ *to* Π *is an algorithm that computes two functions* $g : \mathcal{I}_\Gamma \to \mathcal{I}_\Pi$ *and* $h : \mathcal{S}_\Pi \to \mathcal{S}_\Gamma$*, such that* $h(f_\Pi(g(I))) = f_\Gamma(I)$ *if* f_Π *outputs a single solution and* $h(f_\Pi(g(I))) \subseteq f_\Gamma(I)$ *if* f_Π *outputs a set of solutions.*

In other words, we can transform an instance of Γ into an instance of Π, such that solving the instance of Π lets us solve the original instance of Γ. We now define hardness as follows.

Definition 2.2.6. *A computational problem* Π *is hard for a class* $\mathbb{K}$*, written* $\mathbb{K}$*-hard, if any problem in* $\mathbb{K}$ *can be* reduced *to* Π*. If* Π *is* $\mathbb{K}$*-hard and also a member of* $\mathbb{K}$*, we say that* Π *is* $\mathbb{K}$*-complete.*

Depending on the class there will be additional requirements, typically time and space requirements, for reductions. For P and NP, we require the reductions to run in polynomial time. In Sections 2.3 and 2.5 we will see examples of reductions that have other time and space requirements. If a reduction preserves the number of

solutions of the instance, i.e. $h(k) = k$, we say that the reduction is *parsimonious*. These types of reductions are useful, but not required for counting problems. As long as we are able to retrieve the number of solutions for γ from the number of solutions for π, the reduction is valid.

To determine whether it is likely that an algorithm for a problem Π is optimal in some sense, we use reductions to show that Π is at least as hard as some other problem. If for example we can reduce some NP-hard problem to Π, then Π is also NP-hard and we conclude that it is unlikely that we can find a polynomial time algorithm for Π. If we want to find more precise (probably) bounds on the possible running time of algorithms for Π, we may use conjectures such as the *Exponential Time Hypothesis* (ETH) or the related *Strong Exponential Time Hypothesis* (SETH). To properly state these hypotheses, we first need to state the so called k-Satisfiability problem (k-SAT).

k-Satisfiability
Input: A boolean formula in conjunctive normal form, with clauses of size k.
Question: Does there exist an assignment of the variables, such that the formula is true?

Let s_k be the smallest integer such that k-SAT can be solved in time $2^{s_k n+o(n)}$, where n is the number of variables. ETH and SETH can now be phrased as follows.

Conjecture 2.2.7 (Exponential time hypothesis)**.** $s_3 > 0$.

Conjecture 2.2.8 (Strong Exponential time hypothesis)**.** $\lim_{k\to\infty} s_k = 1$.

For counting problems the weaker counting versions #ETH and #SETH are often sufficient (for an example, see [55]). The definitions for these hypotheses are nearly identical to those of ETH and SETH, replacing k-SAT with #k-SAT.

2.3 Parameterized Complexity

Traditionally, computational complexity measures the efficiency of an algorithm in terms of the size of the input instance. However, as previously alluded to, one can incorporate additional parameters of the input instance into this analysis. Doing so places us into the field of *parameterized complexity*. Computational problems in this setting are labeled by some additional parameter, which can either be given as part of the problem instance, or be some structural property of the problem. We now give two examples, one with a given parameter and one with a structural parameter (see Definition 2.4.2).

q-coloring in at least k ways
Input: A graph G
Parameter: k
Question: Can we find at least k different ways to color $V(G)$ with q colors, such that adjacent vertices receive different colors?

$$\begin{array}{ccccccc} & & & & \text{vc} & & 2\,\text{tpw} \\ & & & & \rotatebox{90}{$\geq$} & & \rotatebox{90}{$\geq$} \\ \text{ctw} & \geq & \text{pw} & \geq & \text{tw} & & \end{array}$$

Figure 2.1: The relation between the various parameters we use. For the purpose of comparison, we set all edge weights to 1 when determining tpw. All inequalities shown can have arbitrarily large gaps.

q-coloring/pw **Input:** A graph G **Parameter:** pw **Question:** In how many ways can we color $V(G)$ with q colors, such that adjacent vertices receive different colors?

While in traditional complexity theory one typically hopes to find an algorithm that runs in polynomial time, in the parameterized setting we attempt to find algorithms that run in fixed-parameter tractable time.

Definition 2.3.1. *An algorithm runs in* fixed-parameter tractable *time (FPT) for parameter k, if there exist a computable function f, such that its running time is bounded by $f(k)n^{O(1)}$.*

We denote the class of all parameterized problems that can solved in FPT time by FPT.

A reduction, that runs in time $g(k)n^{O(1)}$ and produces an instance with a parameter $k' \leq g(k)$ for some computable function g, is called a fixed-parameter reduction *or* FPT-reduction.

Other common complexity classes include XP, the class of problems solvable in $n^{f(k)}$ time and W[1], the class of problems FPT-reducible to Weighted 3-SAT. In Section 2.5 we will see more examples of parameterized complexity classes, as well as a different type of reduction that takes the space requirements into account.

We will now discuss the various structural parameters we will encounter throughout this thesis. We also use some of these parameters for directed graphs. In that case, the direction of edges is ignored, i.e. the value of a parameter for a directed graph equals that parameter for the underlying undirected graph.

2.4 Graph Parameters

2.4.1 Treewidth and Pathwidth

Most of the parameters we encounter will be so called *width* parameters. These parameters are typically defined by some notion of a *decomposition*. We then define the width of said decomposition and say that the width of a graph is the minimum width over all decompositions. We will start with the closely related notions of tree- and pathwidth.

The intuition for these parameters is as follows. We cover the graph in bags, such that any edge/vertex is contained in some bag. We then connect bags that are close

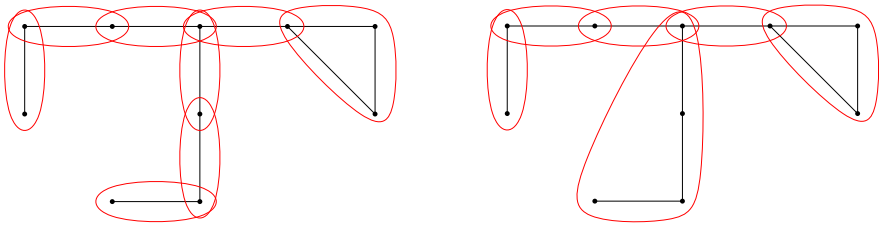


Figure 2.2: Examples of a tree decomposition of width 2 (left) and a path decomposition of width 3 (right).

together, such that the bags get a tree-/path-like structure. We now give a formal definition.

Definition 2.4.1. *A* tree decomposition *of a graph G is given by a tree $\mathbb{T}$ and, for each $x \in V(\mathbb{T})$, a* bag $B_x \subseteq V(G)$*, such that the following holds.*

1. *For every $v \in V(G)$ there is some x such that $v \in B_x$.*
2. *For every $uv \in E(G)$ there is some x such that $u, v \in B_x$.*
3. *For every $v \in V(G)$, the set $\{x \in V(\mathbb{T}) : v \in B_x\}$ induces a subtree of $\mathbb{T}$.*

The width *of such a decomposition is defined as $\max_x(|B_x|)-1$ and the* treewidth*, denoted* $\mathrm{tw}(G)$*, of a graph is defined as the minimum width among its tree decompositions.*

The pathwidth of a graph is defined in a similar fashion.

Definition 2.4.2. *A* path decomposition *of a graph is given by a tree decomposition where the tree $\mathbb{T}$ is a path.*

Again, the width *of such a decomposition is defined as $\max_x(|B_x|) - 1$ and the* pathwidth*, denoted* $\mathrm{pw}(G)$*, of a graph is defined as the minimum width among its path decompositions.*

Since every path decomposition is also a tree decomposition, we find that $\mathrm{pw} \geq \mathrm{tw}$. The gap in this inequality can be made arbitrarily large, by considering a binary tree of depth n, which has $\mathrm{tw} = 1$ and $\mathrm{pw} \geq n$.

We will often think of these decompositions as being rooted at some node r and will refer to the neighbor x of y on the unique path from y to r as the parent of y and to y as the child of x. We will refer to set of nodes y whose unique y-r path visits x as the descendants[1] of x. The union of the bags corresponding to descendants of x will be denoted G_x and we will refer to it as the part of the graph G that *lies below* x.

One useful property of tree/path decompositions is that each bag B_x forms a separator of the graph. Indeed, for any pair of vertices $v \in V(G_x) \setminus B_x$ and $u \in V(G) \setminus G_x$, items 2 and 3 of Definition 2.4.1 can be used to show that $uv \notin E(G)$, since the presence of such an edge would imply that either $u \in B_x$ or $v \in B_x$. The details of this proof are left as an exercise to the reader.

[1] Note that under this definition, x is a descendant of itself.

It is often useful to have a *nice tree decomposition*.

Definition 2.4.3. *A nice tree decomposition is a tree decomposition which contains only the following types of bags.*

- *Leaf bag:* $B_x = \emptyset$ *and* x *has no children.*
- *Node-forget bag:* $B_x = B_y \setminus \{v\}$*, where* y *is the unique child of* x *and* $v \in B_y$*.*
- *Node-introduce bag:* $B_x = B_y \cup \{v\}$*, where* y *is the unique child of* x *and* $v \in V(G) \setminus B_y$*.*
- *Join bag:* $B_x = B_{y_1} = B_{y_2}$*, where* y_1 *and* y_2 *are the two children of* x*.*
- *(optional) Edge-introduce bag:* $B_x = B_y$*, where* y *is the unique child of* x *and* $v \in B_y$*. This bag is labeled by some edge* $uv \in E(G)$ *for* $u, v \in V(G)$*. Each edge has exactly one corresponding edge-introduce bag.*

The last type of bag is labeled as optional, as we will need it for some but not all of our results[2]. If edge-introduce bags are necessary, they will be mentioned in the proof, but as they do not affect the width of the decomposition and there are at most n^2 of such bags, we will not discuss explicitly whether we use them or not. The idea behind this is that we can pretend like an edge doesn't exist, until it gets introduced by some bag. We will assume that edges are always introduced exactly once. By [118, Lemma 13.1.3], we can turn a tree decomposition of width k into a nice tree decomposition of width k with $4n$ nodes, in $O(n)$. We may therefore assume that, when we are given a tree decomposition, it is a nice one.

2.4.2 Weighted Tree Partition Width

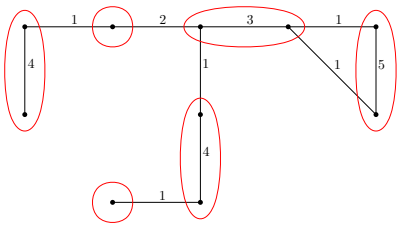


Figure 2.3: An example of a tree partition of breadth 2.

We now introduce a similar notion to treewidth, namely the *weighted tree partition width*. The decomposition will be slightly different, but we then define the corresponding width notion in the same way.

On an intuitive level the difference with treewidth is that the bags in the decomposition for weighted tree partition width do not need to cover all edges and instead must form a partition of the vertices. We now give a formal definition.

[2]Indeed, in some publications this bag type is left out of the definition

Definition 2.4.4. *A* tree partition *of a graph G is a pair $(\{B_i \mid i \in I\}, \mathbb{T})$, with $\{B_i \mid i \in I\}$ a family of subsets (called* bags*) of $V(G)$, and $\mathbb{T}$ a tree, such that*

1. *For each vertex $v \in V(G)$, there is exactly one $i \in V(\mathbb{T})$ with $v \in B_i$ (in other words, $\{B_i \mid i \in V(\mathbb{T})\}$ forms a partition of V, except that we allow that some bags are empty.)*

2. *For each edge $vw \in E(G)$, if $v \in B_i$ and $w \in B_j$ then $i = j$ or $ij \in E(\mathbb{T})$.*

Let $c : E(G) \to \mathbb{N}$ be an edge-weight function. The breadth[3] *of a tree partition $(\{B_i \mid i \in I\}, \mathbb{T})$ of the weighted graph (G, c) is the maximum of $\max_{i \in V(\mathbb{T})} |B_i|$ and $\max_{ij \in E(\mathbb{T})} c(B_i, B_j)$ with*

$$c(B_i, B_j) = \sum_{\substack{v \in B_i, w \in B_j \\ vw \in E(G)}} c(vw),$$

i.e., the maximum sum of edge weights of edges between the bags of $\mathbb{T}$. Then the weighted tree partition width $\mathrm{tpw}(G, c)$ *of (G, c) is the minimum breadth of any tree partition of (G, c). For notational convenience we will consider c to be a part of the graph G and simply write* $\mathrm{tpw}(G)$.

The notion of weighted tree partition width is defined for edge-weighted graphs and originates in the work of Bodleander et al. [17]. For our application in Chapter 6, we interpret the capacity function of a flow problem as the weight function for weighted tree partition width.

If we fix all edge weights to 1, we can relate the Weighted tree partition width to the treewidth as follows. Given a tree partition create a bag $B_{x,y}$ for each pair of adjacent nodes x and y, and place its node xy in between x and y in the tree $\mathbb{T}$. We find a tree decomposition of width $2\,\mathrm{tpw}$ and thus $2\,\mathrm{tpw} \geq \mathrm{tw}$. The gap in this inequality can be made arbitrarily large, by considering a path of length n^2 with one added vertex connected to all other vertices, which has $\mathrm{tw} = 2$ and $\mathrm{tpw} \geq n$.

2.4.3 Cutwidth

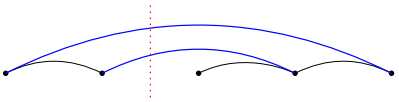


Figure 2.4: An example of a cut decomposition of width 2.

We now give a width measure that does not use a decomposition into bags, but instead uses an ordering of the vertex set as a decomposition. Intuitively the width of a cut decomposition is the maximum edges that are layered on top of each other by the ordering of the vertices, or alternatively the maximum number of edges that

[3] This notion of breadth should not be confused with the breadth of a tree decomposition and the related notion of treebreadth [64]. The breadth of a tree decomposition is defined as the maximum radius of any bag of a tree decomposition.

cross a the space between two consecutive vertices. The pathwidth of a graph can also be defined in a similar way, as the number of vertices on the left of the cut who have at least one neighbor on the right of the cut. We now give a formal definition of the cutwidth of a graph.

Definition 2.4.5. *A* cut decomposition *of a graph G is an ordering $v_1, \ldots, v_n$ of the vertex set. The* width *of such a decomposition is defined as the maximum number of edges* crossing *any* cut *of the ordering. Formally for an integer i we say an edge $v_j v_l$ crosses the i-th cut, if $j \leq i < l$. The* cutwidth, *denoted* ctw, *of G is the minimum width among all cut decompositions.*

The cutwidth of a graph is related to the path and treewidth, in the sense that $\mathrm{ctw} \geq \mathrm{pw} \geq \mathrm{tw}$. In particular there is a natural way to obtain a path decomposition from a cut decomposition, by defining bags $X_i = \{v_i\} \cup \{v_j \in V(G) : j > i, \exists k < i, v_k v_j \in E(G)\}$, i.e. the righthand neighborhood of the i-th cut plus v_i. The gap in this inequality can be made arbitrarily large, by considering a star with $2n$ leaves, which has $\mathrm{pw} = 1$ and $\mathrm{ctw} = n/2$. We will see in Chapters 4 and 5 that the relations between ctw, pw and tw allow us to study these parameters in parallel.

2.4.4 Vertex Cover

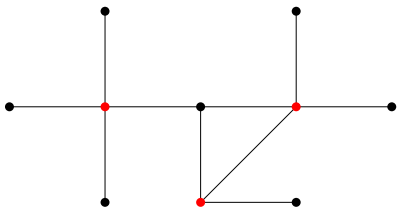


Figure 2.5: The red vertices form a vertex cover of size 3.

The *vertex cover number* captures how spread out the edges of a graph are. As the definition is fairly simple, we skip the intuition and immediately give a formal definition.

Definition 2.4.6. *A* vertex cover *of a graph G is a set $X \subseteq V(G)$ such that $X \cap \{u, v\} \neq \emptyset$ for every edge $uv \in V(G)$. Then the* vertex cover number, *denoted* $\mathrm{vc}(G)$, *of G is the size of the smallest vertex cover of G.*

A useful property of vertex covers is the fact that $V(G) \setminus X$ forms an independent set. For many applications this means that these vertices do not interact and thus can be considered one at a time.

We can relate the vertex cover number to the pathwidth as follows. Given a vertex cover X of G, we can create a path decomposition of G by defining $B_v = X \cup \{v\}$ for $v \in V(G) \setminus X$ and connecting the bags B_v arbitrarily to form a path. We find that $\mathrm{vc} \geq \mathrm{pw}$. The gap in this inequality can be made arbitrarily large, by considering a path of length $2n$, which has $\mathrm{pw} = 1$ and $\mathrm{vc} \geq n$.

2.5 Non-Deterministic Parameterized Logspace complexity

In Part III we will discuss a number of complexity classes that include logarithmic space requirements in their definition. We introduce the classes in question here and give a number of hardness results in Part III. We begin with a short discussion on what it means to use logarithmic space.

When we are given an instance I of size $|I| = n$, in principle we need some storage location of size n to store I on. When we say an algorithm uses $O(\log n)$ space, what we really mean is that the algorithm is given an instance, saved to some read-only storage location, and has only $O(\log n)$ space that it can freely manipulate. One can think of the algorithm's storage as the private storage of a small computer and the input instance as data that the can be requested at will from some large server.

2.5.1 XNLP and XALP

In this section we introduce the classes XNLP and XALP. The class XNLP was first introduced by Elberfeld et al. [71], under a different name. Bodlaender et al. [17, 21, 23] introduced the name XNLP and showed a number of problems to be XNLP-complete with pathwidth as parameter. In particular, [17] gives XNLP-completeness proofs for several flow problems with pathwidth as parameter.

Before we can introduce the classes XNLP and XALP, we need to define the associated types of Turing machines. For this we will assume the reader is familiar with the notion of a Turing machine. We say that a Turing machine is *non-deterministic* if it can make choices that are not predetermined, which we call non-deterministic steps. A co-non-deterministic step is a point where the computation branches into a number of different computations that all need to terminate in accepting states. A Turing machine is *alternating* if it alternates between non-deterministic and co-non-deterministic steps.

Definition 2.5.1. *An* XNLP machine *is a non-deterministic Turing machine such that for a given parameterized instance (I,k), where $|I| = n$:*

(i) For a Yes *instance, at least one computation path accepts.*

(ii) For a No *instance, all computation paths reject.*

(iii) There is some computable function f and some constant c such that the machine runs in at most $f(k)n^c$ time and uses at most $f(k)\log n$ space.

(iv) The machine has read-only access to an input tape of size n and write-only access to an output tape.

Definition 2.5.2. *An* XALP machine *is an alternating Turing machine M such that for each instance (I,k), where $|I| = n$, items items i, ii and iv of Definition 2.5.1 are satisfied (where computation trees take the role of computation paths), and:*

(iii') There is some computable function f and some constant c such that M uses at most $f(k)\log n$ space and such that the computation tree of M on input (I,k) has size at most $f(k)n^c$.

We now formally define the classes XNLP and XALP as follows.

Definition 2.5.3. *(i) The class* XNLP *consists of all problems that can be solved on an XNLP machine.*

(ii) The class XALP *consists of all problems that can be solved on an XALP machine.*

Chapter 6 will contain, among other things, hardness results for both XNLP and XALP. Note that the computation on an alternating Turing machine takes on a tree-like structure, which explains why we will see XALP-hardness for problems parameterized by treewidth and XNLP-hardness for those same problems parameterized by pathwidth. An equivalent definition for XALP is obtained by giving the Turing machine from the definition of XNLP access to an additional stack. This stack can have arbitrary height (only bounded by the runtime), but can only store items of logarithmic size.

Reductions in these classes are done using the following type of reduction.

Definition 2.5.4. *A* parameterized logspace reduction *or* pl-reduction *from a parameterized decision problem* Π*, with parameter* k*, to a parameterized decision problem* Γ*, with parameter* k'*, is a reduction computing* (g, h) *such that the following hold.*

- *There is an algorithm for the reduction using* $O(f(k(I)) + \log n)$ *space, with* f *a computable function and* $n = |I|$ *the number of bits to denote* I*.*
- *There is a computable function* f*, such that for all* $I \in \Pi$*,* $k'(g(I)) \leq f(k(I))$*.*

XNLP-hardness and XALP-hardness are defined with respect to pl-reductions. The main difference with FPT-reductions (Definition 2.3.1) is that the computation of the reduction must be done with logarithmic space. In most cases, existing parameterized reductions are also pl-reductions, as logarithmic space can often be achieved by not storing intermediate results but recomputing these when they are needed.

One of the interesting (probable) consequences of XNLP-/XALP-hardness is given by following conjecture due to Pilipczuk and Wrochna [137] that states that XP-algorithms for XNLP-hard problems are likely to use much memory.

Conjecture 2.5.5 (Slice-wise Polynomial Space Conjecture [137])**.** *If parameterized problem* Π *is XNLP-hard, then there is no algorithm that solves* $I \in \mathcal{I}_\Pi$ *in* $n^{f(k)}$ *time and* $f(k)n^{O(1)}$ *space, for instances* I*, with* f *a computable function,* $k = k(I)$ *and* $n = |I|$ *the size of instance* I*.*

Another interesting consequence is captured by the following lemma.

Lemma 2.5.6. *If problem* Π *is XNLP-hard, then* Π *is hard for all classes* $W[t]$*,* $t \in \mathbb{N}$*.*

The hardness proofs in Chapter 6 start from two variations of the well-known Multicolored Clique problem (see [78]). These problems are common starting points for proving hardness for XNLP and XALP respectively and thus we state them here.

Chained Multicolored Clique
Input: A graph $G = (V, E)$, a partition of V into $V_1, \ldots, V_r$, such that $|i - j| \leq 1$ for each edge $uv \in E(G)$ with $u \in V_i$ and $v \in V_j$, and a function $c\colon V \to [k]$.
Parameter: k.
Question: Is there a set of vertices $W \subseteq V$ such that for all $i \in [r-1]$, $W \cap (V_i \cup V_{i+1})$ is a clique, and for each $i \in [r]$ and $j \in [k]$, there is exactly one vertex $v \in W \cap V_i$ with $c(v) = j$?

Tree-Chained Multicolored Clique
Input: A graph $G = (V, E)$, a tree partition $(\{V_i \mid i \in I\}, T = (I, F))$ with T a tree of maximum degree 3, and a function $c\colon V \to [k]$.
Parameter: k.
Question: Is there a set of vertices $W \subseteq V$ such that for all $ii' \in F$, $W \cap (V_i \cup V_{i'})$ is a clique, and for each $i \in I$ and $j \in [k]$, there is exactly one vertex $v \in W \cap V_i$ with $c(v) = j$?

Theorem 2.5.7 (From [22] and [23])**.** *The problem Chained Multicolored Clique is XNLP-complete, and the problem Tree-Chained Multicolored Clique is XALP-complete.*

2.5.2 #XLP and #XALP

We now introduce the counting versions of XNLP and XALP. As mentioned before, a typical way of defining a class of counting problems, is by starting with a non-deterministic class of decision problems and then defining the counting equivalent as the class of all problems that count the number of accepting computations on a non-deterministic Turing machine for such a problem.

Definition 2.5.8.

(i) The class #XLP *consists of all problems that ask for the number of accepting paths of some XNLP machine.*

(ii) The class #XALP *consists of all problems that ask for the number of accepting trees in some XALP machine.*

As mentioned earlier we can think of #XLP as the counting version of XNLP. Some care needs to be taken however, as it is not immediately obvious that any problem in XNLP becomes a problem in #XLP when counting the number of solutions, since one needs to show that the number of accepting paths of the XNLP machine does in fact correspond to the number of solutions. Fortunately this typically only forms a small hurdle and most problems in XNLP have an obvious counterpart in #XLP.

2.6 Tutte Polynomial

In Part II we work towards a classification of the complexity of computing the Tutte polynomial. Chapters 3 and 4 can be read without knowledge of the Tutte polynomial. Our main results about the Tutte polynomial are given in Chapter 5.

The Tutte polynomial, named after W. T. Tutte, is a graph polynomial. A graph polynomial is a polynomial associated with a graph, meaning that its coefficients depend on the graph in question. One can also think of it as a function whose domain is a class of graphs, that maps each graph to a polynomial. The Tutte polynomial associates a two variable polynomial with each graph. We typically think of these polynomials as taking complex values as input.

There are multiple ways of defining the Tutte polynomial. We will work with the following definition

Definition 2.6.1. *For a graph G, we denote by $T(G;x,y)$ the* Tutte polynomial *of G evaluated at the point (x,y). If G has no edges we have $T(G;x,y)=1$. Otherwise we have*

$$T(G;x,y)=\sum_{A\subseteq E}(x-1)^{k(A)-k(E)}(y-1)^{k(A)+|A|-|V|},$$

where $k(A)$ indicates the number of connected components of (V,A).

A central object in the study of the Tutte polynomial is the hyperbolic curve H_α, for $\alpha\in\mathbb{C}$, of the form

$$H_\alpha=\{(x,y):(x-1)(y-1)=\alpha\}.$$

Note that for $\alpha=0$ the hyperbola collapses into two orthogonal, straight lines. We refer to these two lines as separate curves

$$H_0^x=\{(x,y):x=1\},$$
$$H_0^y=\{(x,y):y=1\}.$$

We will sometimes refer to the problem of finding the value of $T(G;a,b)$ for an individual point (a,b) as *computing the Tutte polynomial on* (a,b). We will often restrict the Tutte polynomial to the one-dimensional curve H_α. Note that in this case the polynomial can be expressed as a univariate polynomial

$$T_\alpha(G;t):=T\left(G;\frac{\alpha}{t}+1,t+1\right),$$

for $\alpha\neq 0$ and

$$T_\alpha^x(G;t):=T\left(G;1,t\right),$$
$$T_\alpha^y(G;t):=T\left(G;t,1\right),$$

for $\alpha=0$. We will refer to the problem of finding the coefficients of T_α as *computing the Tutte polynomial along* H_α. When proving hardness results for such curves, we will often use the following transformations.

Definition 2.6.2. *The* k-stretch *of a graph* G *is the graph obtained from* G *by replacing each edge with a path of length* k*. The* k-thickening *of a graph* G *is the graph obtained from* G *by replacing each edge with* k *parallel edges. We denote the* k*-stretch of* G *by* ${}^{k}G$ *and the* k*-thickening as by* G_k*.*

The Tutte polynomial is known to be computable in polynomial time on the points

$$(1,1),(-1,-1),(0,-1),(-1,0),(i,-i),(-i,i),(j,j^2),(j^2,j), \tag{2.1}$$

and along the curve H_1. It is #P-hard to evaluate it on any other point. We call the points listed in (2.1), along with the points on the curve H_1 *special points*. See also [106] for more details on (the complexity of) the Tutte polynomial.

2.7 Interpolation

In Chapters 5 and 7 we will use interpolation to retrieve the coefficients of a polynomial, given a finite set of evaluations of said polynomial. For our purposes it suffices to note that this can be done in polynomial time, for example by solving the system of linear equations given by the Vandermonde matrix and the evaluations (see e.g. [46, Section 30.1]).

Lemma 2.7.1. *Given pairs* $(x_0,y_0),\ldots,(x_d,y_d)$*, there exists an algorithm which computes the unique degree* d *polynomial* p *such that* $p(x_i)=y_i$ *for* $i=0,\ldots,d$ *and runs in time* $O(d^3)$*.*

In Chapter 7 we want to 'interpolate' polynomials on n variables using only $O(\log n)$ space. In particular we want to retrieve the value of $p(\alpha_1,\ldots,\alpha_n)$ for some given point[4] $(\alpha_1,\ldots,\alpha_n)$, using the evaluations of p on the grid $[d+1]^s$. This can be achieved by specifically using Lagrange interpolation. Let us define

$$L_{(j_1,\ldots,j_n)}(x_1,\ldots,x_n)=\prod_{\kappa\in[n]}\prod_{\substack{i\in[d+1]\\ i\neq j_\kappa}}\frac{x_\kappa-i}{j_\kappa-i}$$

and observe that $L_{\mathbf{j}}(\mathbf{k})=1$ when $\mathbf{j}=\mathbf{k}$ and $L_{\mathbf{j}}(\mathbf{k})=0$ for all other $\mathbf{j},\mathbf{k}\in[d+1]^n$. We thus have

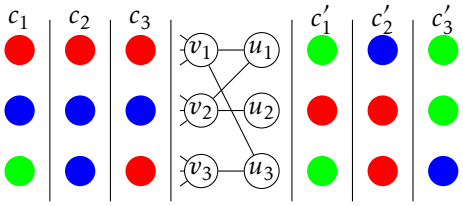
$$p(x_1,\ldots,x_s)=\sum_{\mathbf{j}\in[d+1]^n}p(\mathbf{j})L_{\mathbf{j}}(x_1,\ldots,x_n).$$

We can compute the previous expressions using $O(\log n)$ space by computing each term in the sum/product one at a time and storing only the running total. We summarize this result in the following lemma

Lemma 2.7.2. *Let* $R\subseteq\mathbb{Q}$ *be a finite set. Let* $p\in\mathbb{Z}[x_1,\ldots,x_s]$ *for* $s=O(1)$ *be a polynomial of maximum degree* d*. If there is a* $d^{O(1)}$ *time- and* $O(\log d)$ *space-bounded algorithm that computes* $p(\mathbf{j})$ *on input* $\mathbf{j}\in[d+1]^s$*, then there also exists such an algorithm for computing* $p(\xi)$ *at any input* $\xi\in R^s$*.*

[4] Note that this point needs to be storable in $O(\log n)$ space.

2.8 Rank-Based Approach



	c'_1	c'_2	c'_3
c_1	0	0	1
c_2	1	0	0
c_3	1	0	1

Figure 2.6: An example of a separator $\{v_1, v_2, v_3\}$, the remainder of the graph $\{u_1, u_2, u_3\}$, some partial colorings c_1, c_2, c_3 of $\{v_1, v_2, v_3\}$ and a partial coloring c' of $\{u_1, u_2, u_3\}$. To the right we see the associated color compatibility matrix, restricted to the set of depicted colorings.

In Chapters 3 and 4 we use the *rank-based approach* to design fast *dynamic* programming (DP) algorithms based on problem specific matrices. As this method is somewhat new, it was first introduced in [18], we will spend some time to introduce it properly and discuss the ins and outs of applying it to counting problems.

The rank-based method requires a problem whose solutions can be broken up into partial solutions. These kinds of problems are often a natural fit for dynamic programming algorithms, as one can often turn small partial solutions into bigger ones, until a full solution is found. In this thesis partial solutions are typically defined by some separator of a graph, for example a bag in a tree decomposition. An example of this can be seen in q-coloring/pw, which we saw earlier in Section 2.3. Here a partial solution consists of a coloring of some subset of the vertices, see Figure 2.6 for an example we will use throughout this section. A dynamic programming algorithm might for example keep track of all partial colorings of the bags, that can be extended to some partial coloring of the set of vertices appearing in or before said bag. One can also consider partial solutions, i.e. partial colorings, of both sides of said bag and ask if they combine into a full solution, i.e. a proper coloring of the whole graph. We say that two such partial solutions are compatible. In Figure 2.6 partial colorings c_2 and c' are compatible, but c_1 and c' are not. The notion of compatibility will play an important role in the rest of this section.

At its core, the rank-based approach is about compressing the size of the dynamic programming table, without losing relevant information. For a decision problem this typically means omitting partial solutions that, in some sense, could lead to duplicate answers. We will refer to the set that is left over as a *representative set*. More concretely, let P be the set of all partial solutions on some part I_1 of the instance and let I_2 be the remaining part of the instance. A representative set $R \subseteq I_1$ of I_1 is a set of partial solutions such that for any partial solution s_2 of I_2 we have that there is some partial solution $s_1 \in P$ that is compatible with s_2 if and only if there is some partial solution $s'_1 \in R$ that is compatible with s_2. In Figure 2.6 the set $\{c_1, c_2\}$ forms a representative set of $\{c_1, c_2, c_3\}$. In fact, any subset of size 2 is a representative set in this example, since the colorings only differ on v_3, which has degree 1.

The crucial property of a representative set, is the following. Let s be some solution to I, such that s_1, the part of s that lies on I_1, is in P. By definition, we can replace s_1 with some element of R. This means that we if we only remember R, instead of all of P, we still find a solution if and only if we would have found one from P. In particular if we run a dynamic programming algorithm were we only store some representative set of each layer of the table, we know the algorithm is still correct.

It can be useful to rephrase the notion of representative sets in terms of matrices.

Definition 2.8.1. *Let I be an instance of some computational problem, that consists of two parts I_1 and I_2. Given some notion of compatibility between partial solutions on I_1 and I_2, we define the corresponding* compatibility matrix *M (at I_1 and I_2) as follows. The rows of M are indexed by partial solutions of I_1 and the columns are indexed by partial solutions of I_2. The entries are given by*

$$M[s_1, s_2] = \begin{cases} 1 & \text{if } s_1 \text{ and } s_2 \text{ are compatible,} \\ 0 & \text{otherwise.} \end{cases}$$

See Figure 2.6 for an example matrix. We can now rephrase the notion of a representative set as follows.

Definition 2.8.2. *Let I be an instance of some computational problem, that consists of two parts I_1 and I_2. Let P be a set of partial solutions of I_1 and M the compatibility matrix at I_1 and I_2. Let $\mathrm{cov}(P)$ be the set of partial solutions s_2 of I_2 for which there is some $s_1 \in P$ such that $M[s_1, s_2] = 1$. We say that R is a* representative set *of P if $\mathrm{cov}(P) = \mathrm{cov}(R)$.*

It is probably not surprising that the word rank in the name 'rank-based approach' refers to the rank of the compatibility matrix M. It turns out that this rank plays a crucial role in determining the efficacy of this approach.

Lemma 2.8.3. *Let I be an instance of some computational decision problem, that consists of two parts I_1 and I_2. Let P be the set of all partial solutions of I_1 and M the compatibility matrix at I_1 and I_2. There exists a representative set R of P of size $|R| \leq \mathrm{rank}(M)$.*

Proof. Let R be some basis of the rowspace of M. Note that $|R| \leq \mathrm{rank}(M)$, by virtue of it being a basis. Also note that for any partial solution s_1, its row $M[s_1]$ can be written as a linear combination of the rows corresponding to elements of R, again because R is a basis. In particular this means that for any s_2, we have that $M[s_1, s_2] = 1$ implies that there is some $s_1' \in R$ such that $M[s_1', s_2] = 1$.

The reverse direction follows from the fact that $R \subseteq P$. □

Lemma 2.8.3 says that there always exists some representative set of size at most $\mathrm{rank}(M)$. This indicates that decision problems that have a low rank compatibility matrix have fast dynamic programming algorithms, assuming we have a fast enough way to find representative sets.

The matrix perspective also makes it really easy to translate the notion of representation to the counting regime. We could define some notion of a weighted representative set, but it will prove more useful and far easier to think of the layers of the dynamic programming table as vectors. These vectors are indexed by partial

solutions s_1 and each entry indicates how many other partial solutions s_1 represents. Typically counting algorithms will already do some compression, by only looking at the partial solution at the interface of I_1 and I_2.

To give a concrete example, we may want to count graph colorings on a graph of bounded pathwidth. I_1 may consist of G_x for some node x in the path decomposition and I_2 then consists of $G[(V(G) \setminus V(G_x)) \cup B_x]$. The dynamic programming table T at entry (x, c) will contain the number of partial solutions c' of G_x such that $c'|_{B_x} = c$. We can then think of the set $\{T[x, c] : c \in \text{col}(B_x)\}$ for some fixed x, as a vector $T[x]$ indexed by colorings of B_x.

We can further compress this vector by considering the following version of representation

Definition 2.8.4. *Given two vectors T and T', we say T' is an M-*representative *of T if*

$$\sum_{x \in \text{col}(X)} M[x, y]T[x] = \sum_{x \in \text{col}(X)} M[x, y]T'[x] \text{ for all } y \in \text{col}(Y),$$

i.e. $TM = T'M$.

Definition 2.8.4 actually captures representation for decision problems as well, if addition and multiplication are done over the Boolean semiring, which is the semiring consisting of 0 and 1, where addition is given by a logical OR and multiplication is given by a logical AND. The following counting version of Lemma 2.8.3 follows immediately from the fact that a rowbasis has size $\text{rank}(M)$.

Lemma 2.8.5. *Let $T \in \mathbb{R}^n$ be some vector and $M \in \mathbb{R}^{n \times m}$ for some $n, m \in \mathbb{N}$. There exists a representative vector T' of T with a support of size $|\text{supp}(T')| \leq \text{rank}(M)$.*

Note that we don't even need to know anything about the original problem, to describe the notion of representation. We can phrase it all in terms of linear algebra, which results in really clean statements. It is also worth noting that this representative vector T' can be any vector and doesn't need to have any concrete meaning in the context of the original problem. This typically is not an issue as a lot of dynamic programming algorithms will use simple operations like addition and copying of entries, which we can still apply to the entries of this vector. In practice we see that the standard DP almost always preserves representation. By this we mean that for T'_i a representative of T_i, applying a step of the DP to T'_i produces a vector T'_{i+1} that is also a representative of T_{i+1}, the result of applying that same step to T_i.

One might wonder if there is some meta-theorem that indicates when a DP preserves representation, for example if the DP step is given by a matrix of some specific type? Writing D for the matrix describing the DP step, i.e. $T_{i+1} = DT_i$, we can write the general requirement as[5] $MDa = DMa$ for all vectors a. This is equivalent to saying that the two matrices must commute. Unfortunately, this is often difficult to check for a given dynamic program, as the matrices can be highly nontrivial.

As a final note we briefly discuss a type of matrix that we will apply the rank-based approach to in Chapter 3. In this chapter we consider graph coloring parameterized by cutwidth[6], which is a problem that is highly local in nature, i.e. the

[5] For convenience, we assume that T_i and T_{i+1} have the same length.

[6] Not to be confused with the example in Figure 2.6, which is parameterized by pathwidth.

problem can be defined in terms of requirements on individual edges, rather than larger structures in the graph. As such the compatibility matrix at a particular cut is a submatrix of a highly decomposable matrix. This is often the case with local problems, which can be quite helpful for determining the rank of the compatibility matrix. In this particular case we can decompose the matrix as a *Kronecker product*.

Definition 2.8.6. *Given two matrices $A \in \mathbb{R}^{m\times r}$ and $B \in \mathbb{R}^{n\times t}$, the* Kronecker product *of A and B is an $mn \times rt$ matrix, with entries[7] given by $(A\otimes B)[(i,j),(k,l)] = A[i,j]\cdot B[k,l]$.*

For this type of product we have the following helpful property.

Lemma 2.8.7 (Folkore)**.** *Given two matrices A and B, we have* $\operatorname{rank}(A\otimes B) = \operatorname{rank}(A)\cdot \operatorname{rank}(B)$.

This means that for these kinds of local problems we often only need to determine the rank of one compatibility matrix M_0, corresponding for example to an edge. We can then easily find the rank of the compatibility matrix M corresponding to any part of the problem, as M is a submatrix of a Kronecker power of M_0.

[7] For convenience, we index by pairs (i,j) with $i \in [m]$ and $j \in [n]$.

Part II

Determining the Parameterized Complexity of the Tutte Polynomial

3 Counting List q-Colorings Parameterized by Cutwidth

The whole point is to live life and be - to use all the colors in the crayon box.

RuPaul

3.1 Introduction

In this chapter we discuss the problem of counting (List) q-colorings modulo some prime. We will give fine-grained matching upper (Section 3.2) and lower (Section 3.3) bounds on the parameterized time complexity of this problem. Additionally we will extend our results to the problem of counting connected edgesets (Section 3.4). These results will be a first step towards our goal of determining the parameterized complexity of the Tutte Polynomial, which we discuss in Chapter 5. Since the results in this chapter were first published, Bojikian et al. [26] extended the algorithm we give to *Coloring-like Problems*, a generalization of counting graph homomorphisms.

Counting the number of colorings of a graph is known to be #P-complete, even for special classes of graphs such as triangle free regular graphs [97]. Björklund and Husfeldt [13] and Koivisto [119] gave a $2^n n^{O(1)}$ time algorithm for counting q-colorings.

In this chapter we only look at parameterization by cutwidth (ctw) (see Definition 2.4.5). For treewidth (tw), a standard dynamic programming approach results in a $q^{\text{tw}} n^{O(1)}$ time algorithm, for which [123] gave a matching $(q-\varepsilon)^{\text{pw}} n^{O(1)}$ lower bound parameterized by pathwidth (pw), even for the decision version. Interestingly enough, there is a $2^{\text{ctw}} n^{O(1)}$ time randomized algorithm [109] for q-coloring, which begs the question whether one could also lose the dependency on q in the counting setting, when parameterizing by cutwidth. In this chapter we will see that this is not the case. This indicates that the cutwidth does not give any extra leverage as a parameter, when compared to the path- or treewidth, at least in the general setting.

In the modular setting we will see that there is one curious case, namely when the modulus p divides $q-1$, where we can achieve a slight speedup to $(q-1)^{\text{ctw}} n^{O(1)}$. This anomaly has everything to do with the rank of the *color compatibility matrix* as described later in Section 3.1.2. It is good to point out that without the rank-based perspective, this anomaly is difficult to explain. This chapter is the only time we

will prove results specifically for modular counting, rather that regular counting. The previously mentioned anomaly is the main reason we choose to work in the modular setting here, as it gives an interesting example of the rank-based approach (see Section 2.8), which we will apply in a later chapter as well, to achieve another crucial step towards the parameterized complexity of the Tutte Polynomial. In previous work, it was also shown that the rank of similar matrices can be used to design both algorithms [18, 57, 109, 132] and lower bounds [53, 57].

3.1.1 Results and Notation

As mentioned, in this chapter we study the complexity of two natural hard (modular) counting problems: Counting the number of q-colorings of a graph and counting the number of spanning connected edge sets, parameterized by the cutwidth of the graph. All graphs we consider will be undirected.

Counting Colorings Given a graph $G = (V, E)$, and lists $L : V \to 2^{[q]}$, a *list q-coloring* of G is a coloring $c : V \to [q]$ of its vertices such that $c(u) \neq c(v)$ for all edges uv and $c(v) \in L(v)$ for all vertices v. We will often abbreviate 'list q-coloring' to 'coloring'. Given a subset $A \subseteq V$, we use $\mathrm{col}_L(A)$ to denote the set of all list q-colorings of $G[A]$. If it is clear which lists are used, we omit the subscript. Our first result is about counting the number of list q-colorings.

#List q-Colorings/ctw
Input: A graph G, a function $L : V \to 2^{[q]}$, a width ctw cut decomposition $v_1, \ldots, v_n$ of G.
Parameter: ctw
Question: How many colorings $c : V(G) \to [q]$ are there, such that $c(u) \neq c(v)$ for all edges $uv \in E(G)$ and $c(v) \in L(v)$ for all vertices $v \in V(G)$?

Note that, if G is not connected, we can count the number of q-colorings in each connected component separately and multiply them to get the total number of q-colorings of G. Therefore, we will assume that G is connected.

We now turn to our main theorem, which reads as follows:

Theorem 3.1.1. *Let $p, q \in \mathbb{N}$ with p prime and $q \geq 3$.*

a. *If p divides $q-1$, there is no $\varepsilon > 0$ for which there exists a $(q-1-\varepsilon)^{\mathrm{ctw}} n^{O(1)}$ time algorithm that counts the number of list q-colorings modulo p assuming SETH.*

b. *If p divides $q-1$, then there is a $O((q-1)^{\mathrm{ctw}} n)$ time algorithm for counting list q-colorings modulo p of n-vertex graphs of cutwidth* ctw.

c. *If p does not divide $q-1$, there is no $\varepsilon > 0$ for which there exists a $(q-\varepsilon)^{\mathrm{ctw}} n^{O(1)}$ time algorithm that counts the number of list q-colorings modulo p, assuming SETH.*

Since the initial publishing of the results in this chapter, a paper by Bojikian et al. [26] was published that generalizes our algorithmic results to so called *coloring-like problems*. One can think of these as colorings, where an arbitrary color compatibility matrix M is given for the graph of a single edge, in addition to some other extra

parameters. Alternatively, coloring-like problems can be thought of as a slight generalization of H-coloring, the problem where one is interested in homomorphisms from the input graph G to some fixed host graph H. For context, Dyer and Greenhill [68] showed that counting the number of H-colorings is #P-complete unless H is one of a few exceptions (an independent set, a complete graph with loops on every vertex or a complete bipartite graph). Kazeminia and Bulatov [117] classified the hardness of counting H-colorings modulo a prime p for square-free graphs H. Bojikian et al. showed that, as long as the number of colorings is less than $\text{rank}(M)^2$, there still exists a $\text{rank}(M)^{\text{ctw}} n^{O(1)}$ algorithm.

Connected Spanning Edge Sets and Tutte polynomial We say that $X \subseteq E$ is a *connected spanning edge set* if $G[X]$ is connected and every vertex is adjacent to an edge in X. Our second result is about counting the number of such sets.

#Connected Edgesets/ctw
Input: A graph G, a width ctw cut decomposition $v_1, \ldots, v_n$ of G.
Parameter: ctw
Question: How many subsets $X \subseteq E(G)$ are there, $G[X]$ is connected and for every $v \in V(G)$, there is some $e \in X$, such that $v \in e$?

This problem is naturally motivated: It gives the probability that a random subgraph remains connected, and is an important special case of the Tutte polynomial. We determine the complexity of counting connected spanning edge sets by treewidth and cutwidth by giving matching lower and upper bounds:

Theorem 3.1.2. *Let p be a prime number. There is an algorithm that counts the number of connected edge sets modulo p of n-vertex graphs of treewidth* tw *in time $p^{\text{tw}} n^{O(1)}$.*

Furthermore, there is no $\varepsilon > 0$ for which there is an algorithm that counts the number of spanning connected edge sets modulo p of n-vertex graphs of cutwidth ctw *in time $(p-\epsilon)^{\text{ctw}} n^{O(1)}$, assuming SETH.*

Note that before the publication of this work, even for the treewidth parameterization, the best conditional lower bound by Dell et al. [61] only excluded $2^{o(\text{tw})} n^{O(1)}$ time algorithms for this problem.

While the algorithm follows relatively quickly by using a cut-and-count type dynamic programming approach, obtaining the lower bound is much harder.

In fact, for related counting variants of connectivity problems such as counting the number of Hamiltonian cycles or Steiner trees, $2^{O(\text{tw})} n^{O(1)}$ time algorithms do exist. So one may think that connected spanning edge sets can be counted in a similar time bound. But in Theorem 3.1.2 we show that this is not the case (by choosing p arbitrarily large).

To prove the lower bound, we make use of an existing formula for the Tutte polynomial that relates the number of connected spanning edge sets to the number of essentially distinct colorings, and subsequently apply Theorem 3.1.1.c.

3.1.2 The Color Compatibility Matrix and its Rank

In this subsection we will introduce the central object of this chapter, the *color compatibility matrix*. This will be the matrix used for this chapter's application of the rank-based approach (see Definition 2.8.4). The rank of this matrix will play a crucial role in both the running time of our algorithm and the matching lower bound we find.

We can think of the color compatibility matrix as describing the behaviour of the coloring problem around some separator in our graph. In particular, we can think of a graph separator as splitting any proper coloring into two partial colorings and as such we may ask if two given partial colorings form a proper coloring when combined. If they do, we call these partial colorings compatible:

Definition 3.1.3. *Let $A, B \subseteq V$. For colorings $x \in \mathrm{col}(A)$, $z \in \mathrm{col}(B)$, we say that x and z are* compatible, *written $x \sim z$, if*

- *$x(v) = z(v)$ for all $v \in A \cap B$, and*
- *$x(u) \neq z(v)$ for all $uv \in E$, where $u \in A$ and $v \in B$.*

For a set of colorings $\mathcal{S} \subseteq \mathrm{col}(B)$, we write $\mathcal{S}[x]$ for the set of colorings $y \in \mathcal{S}$ that are compatible with x.

If $x \sim z$, then we use the convention of writing $x \cup z$ for the q-list coloring of $G[A \cup B]$ defined as $(x \cup z)(a) := x(a)$ for all $a \in A$ and $(x \cup z)(b) := z(b)$ for all $b \in B$. This is well-defined by the definition above.

Using Definition 3.1.3 we can now define the color compatibility matrix.

Definition 3.1.4. *Let $(X \cup Y, E)$ be a bipartite graph and q a natural number. The qth* color compatibility matrix *M is indexed by all q-colorings of X and Y, with*

$$M[x,y] = \begin{cases} 1, & \text{if } x \sim y, \\ 0, & \text{otherwise,} \end{cases}$$

for $x \in \mathrm{col}(X)$ and $y \in \mathrm{col}(Y)$.

We denote the color compatibility matrix indexed by all q-colorings associated with the bipartite graph that is matching on t vertices by J_t, and use the short-hand $J := J_1$.

We conclude this section by determining (a bound on) the rank of this matrix, when taken over a finite field. We are interested in this particular version of rank because we are counting in a modular setting. In this setting the rank may be lower than over the integers and we will find that in some cases the rank of the color compatibility matrix is in fact strictly smaller over a finite field.

Lemma 3.1.5. *Let p be a prime, q a natural number and let $G = (X \cup Y, E)$ be a bipartite graph with qth color compatibility matrix M. Then the rank of M over $\mathbb{F}_p$ satisfies*

$$\mathrm{rank}_p(M) \leq \begin{cases} (q-1)^{|E|} & \text{if } p \text{ divides } q-1, \\ q^{|E|} & \text{otherwise.} \end{cases}$$

Moreover, equality is achieved if G is a perfect matching.

Proof. Recall that we denote the color compatibility matrix associated with the bipartite graph that is a matching on t pairs by J_t. We will first compute $\operatorname{rank}_p(J_1)$, which will cause the difference between the cases when p divides $q-1$ and when it does not. We then show that $\operatorname{rank}_p(J_t) = \operatorname{rank}_p(J_1)^t$. Finally we show that $\operatorname{rank}_p(M) \leq \operatorname{rank}_p(J_{|E|})$ for any graph G and its color compatibility matrix M.

Matchings: Let us first consider the case where G is a perfect matching on t edges and thus is its color compatibility matrix is $M = J_t$.

We first compute the rank of J_1. Recall that J_1 is a $q \times q$ matrix with zeros on the diagonal and ones off-diagonal. If p divides $q-1$ the rows of this matrix sum to 0 and it is easy to see that any other linear combination of rows does not sum to 0. If p does not divide $q-1$ any non-trivial linear combination of rows will not sum to 0. We conclude that this matrix has rank $q-1$ if p divides $q-1$ and full rank, i.e. rank q, otherwise.

We now consider the rank of J_r for $r \geq 2$. Recall that the Kronecker product (Definition 2.8.6) of an $m \times n$ matrix A and a $p \times q$ matrix B is defined as the following $pm \times qn$ block matrix

$$A \otimes B = \begin{pmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m_1}B & \cdots & a_{mn}B \end{pmatrix}.$$

We will show by induction on r that $M = J_r$ is the rth Kronecker power of J_1. By the well known fact that $\operatorname{rank}_p(A \otimes B) = \operatorname{rank}_p(A) \cdot \operatorname{rank}_p(B)$ we then find

$$\operatorname{rank}_p(J_r) = \operatorname{rank}_p(J_1)^r.$$

The entries on the diagonal of J_1 are zero and the off-diagonal entries are one: two colors are in conflict if and only if they are the same. Consider now some $r \geq 2$ and let $X = \{x_1, \ldots, x_r\}$, $Y = \{y_1, \ldots, y_r\}$ and $E = \{\{x_1, y_1\}, \ldots, \{x_r, y_r\}\}$. We write a coloring of X as $(c_1, \ldots, c_r)$, where c_i is the color assigned to x_i. We assume that the entries in M are indexed by (lexicographically ordered) colorings $(c_1, \ldots, c_r)$ and $(c'_1, \ldots, c'_r)$, where c_i and c'_i are the colors of x_i and y_i respectively. We will split up M into q^2 different $q^{r-1} \times q^{r-1}$-matrices:

$$M = \begin{pmatrix} M^{(11)} & \ldots & M^{(1q)} \\ \vdots & \ddots & \vdots \\ M^{(q1)} & \ldots & M^{(qq)} \end{pmatrix}.$$

For $i, j \in [q]$, the matrix $M^{(ij)}$ is indexed by colorings where x_1 receives color i and y_1 receives color j. Therefore all $M^{(ii)}$ equal the all-zeros matrix. For $i \neq j$, the entries are 1 if and only if the two colorings are compatible on $x_1, \ldots, x_r, y_2, \ldots, y_r$. We find that $M^{(ij)}$ is the color compatibility matrix J_{r-1} of the matching on $r-1$ vertices, i.e. $((X \setminus \{x_1\}) \cup (Y \setminus \{y_1\}), E \setminus \{x_1, y_1\})$. This proves that M equals the Kronecker product $J_1 \otimes J_{r-1}$. Therefore, J_r is the rth Kronecker power of J_1, so it has rank $(\operatorname{rank}(J_1))^r$. If p divides $q-1$ we find $\operatorname{rank}_p(J_t) = (q-1)^t$ and otherwise we find $\operatorname{rank}_p(J_t) = q^t$.

General case: Let us now consider the case where G is any bipartite graph with t edges and let M be the corresponding color compatibility matrix. By a similar argument to the one above, we may assume that G has no isolated vertices (else we can write M as the Kronecker product of the all ones matrix and the color compatibility matrix of the graph G from which one isolated vertex is removed). Let G_t be a perfect matching with t edges and J_t the corresponding color compatibility matrix. We claim that M is a submatrix of J_t.

Namely, note that we can obtain G from G_t by identifying some of its vertices. We observe how the color compatibility matrix changes after identifying two vertices: identifying vertices u, v from the left-hand side of the bipartition corresponds to deleting all rows where u and v are assigned a different color. Similarly, identifying vertices in the right part corresponds to deleting columns. Therefore, M can be obtained from J_t by deleting some rows and columns. In particular, $\text{rank}_p(M) \leq \text{rank}_p(J_t)$. As we have already shown that the lemma holds for J_t, this proves the lemma for M.

□

In particular, J_t is invertible mod p if and only if p does not divide $q-1$. This difference in rank, depending on the relation between p and q is the reason that the divisibility by p keeps showing up in the various results in this chapter.

3.2 The Algorithm

We start by giving an algorithm for counting q-colorings modulo p. In particular we consider the case where p divides $q-1$, since this is the only case in which we can achieve a speedup over the standard $q^{\text{ctw}} n^{O(1)}$ dynamic programming algorithm based on Lemma 3.2.1. Specifically, in this section we prove Theorem 3.1.1.b.

Theorem 3.1.1.b. *(restated) If p divides $q-1$, then there is a $O((q-1)^{\text{ctw}} n)$ time algorithm for counting list q-colorings modulo p of n-vertex graphs of cutwidth* ctw.

3.2.1 Definitions and Overview

We first introduce some additional notation and definitions needed in this section. Let q be an integer and let p be a prime that divides $q-1$. We are given a graph $G = (V, E)$ with cutwidth ordering $v_1, \ldots, v_n$ of the vertices. Without loss of generality, we may assume that G is connected. We write $G_i = G[\{v_1, \ldots, v_i\}]$ and

$$L_i = \{v \in V(G_i) : vv_j \in E \text{ for some } j > i\}.$$

Note that by definition of cutwidth, $L_i \subseteq L_{i-1} \cup \{v_i\}$ and $|L_i| \leq \text{ctw}$ for all i (since the number of endpoints of a set of edges is upper bounded by the number of edges in the set).

Let $i \in [n]$ be given and write $X_i = L_i \cup \{v_i\}$ for the set of vertices left of the cut that either have an edge in the cut, or are the rightmost vertex left of the cut. We also define $Y_i = \{v_{i+1}, \ldots, v_n\} \cap N(X_i)$. Figure 3.1 illustrates this notation.

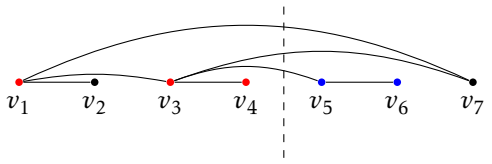


Figure 3.1: In the above graph, $L_4 = \{v_1, v_3\}$, $X_4 = \{v_1, v_3, v_4\}$ (red vertices) and $Y_4 = \{v_5, v_7\}$ (blue vertices).

Let $T_i[x]$ be the number of extensions of $x \in \mathrm{col}(X_i)$ to a coloring of G_i. Equivalently, $T_i[x]$ gives the number of colorings of G_i that are compatible with x.

A standard dynamic programming approach builds on the following observation.

Lemma 3.2.1 (Folklore)**.** *For $x \in \mathrm{col}(X_i)$,*

$$T_i[x] = \sum_{\substack{z \in \mathrm{col}(X_{i-1}) \\ z \sim x}} T_{i-1}[z].$$

Proof. By definition of the X_i, we have $X_i \setminus X_{i-1} = \{v_i\}$ and so there are no edges between $V(G_{i-1}) \setminus X_{i-1}$ and $X_i \setminus X_{i-1}$. This means a coloring of G_{i-1} is compatible with x if (and only if) its restriction to X_{i-1} is compatible with x. This 'cut property' is why the equation holds.

We now spell out the technical details. Recall that we write $\mathrm{col}(X_i)[x]$ for the set of colorings $z \in \mathrm{col}(X_i)$ that are compatible with x. We need to prove that

$$|\mathrm{col}(G_i)[x]| = |\{(z, \phi_{i-1}) : z \in \mathrm{col}(X_{i-1})[x],\ \phi_{i-1} \in \mathrm{col}(G_{i-1})[z]\}|.$$

Let $\phi \in \mathrm{col}(G_i)[x]$, that is, a list q-coloring of G_i compatible with x. Then $z = \phi|_{X_{i-1}}$ is compatible with x and $\phi_{i-1} = \phi|_{G_{i-1}}$ is compatible with z. So $\phi \mapsto (z, \phi_{i-1})$ maps from the set displayed on the left-hand side to the set displayed on the right-hand side. Its inverse is given by $(z, \phi_{i-1}) \mapsto x \cup \phi_{i-1}$. This is well-defined: x and ϕ_{i-1} are compatible because $z \in \mathrm{col}(X_{i-1})$ is compatible with x, and $\phi_{i-1}|_{X_{i-1}} = z$ (here we use the above mentioned 'cut property'). □

Since $|\mathrm{col}(X_i)|$ may be of size $q^{|X_i|}$, we cannot compute T_i in its entirety within the claimed time bound. The idea of our algorithm is to use the same dynamic programming step to compute T_i from T_{i-1}, but to compress the table to an equivalent one of significantly smaller size. Here we will use the notion of representation as described in Definition 2.8.4. For the readers convenience we will restate this definition in the context of the color compatibility matrix M.

Definition 3.2.2. *Let $H = (X \cup Y, E)$ be a bipartite graph with color compatibility matrix M. Let $T, T' : \mathrm{col}(X) \to \mathbb{F}_p$. We say T' is an M-*representative *of T if $M \cdot T \equiv_p M \cdot T'$.*

In this definition we leave the lists and the integer q implicit.

Using this definition, the main idea behind our algorithm can now be phrased as follows. If we think of T_i as a vector indexed by colorings we compute another vector T_i' with smaller support, that M-represents T_i for the color compatibility matrix M.

We now look a bit closer at what the notion of representation means in this context. Recall that the color compatibility matrix has entries $M[x,y] = 1$ if $x \in \mathrm{col}(X)$ and $y \in \mathrm{col}(Y)$ are compatible, and $M[x,y] = 0$ otherwise. Let $i \in [n-1]$ be given. Let M_i be the color compatibility matrix of the bipartite graph given by the edges between X_i and Y_i.

Then for $y \in \mathrm{col}(Y_i)$,

$$\sum_{x \in \mathrm{col}(X_i)} M_i[x,y]T_i[x]$$

gives the number of colorings of G_i compatible with y.

We can compute the the total number of colorings, by computing a T_{n-1}' that is an M_{n-1}-representative of T_{n-1}. By summing over all possible colors for v_n we can then compute the number of q-list colorings of the graph (modulo p) as[1]

$$\sum_{y \in \mathrm{col}(G[v_n])} \sum_{x \in \mathrm{supp}(T_{n-1}')} M_{n-1}[x,y]T_{n-1}'[x].$$

By expressing $M \cdot T$ as a linear combination of some column basis of M, we find a T' that M-represents T with $|\mathrm{supp}(T')| \leq \mathrm{rank}(M)$. We also need to make sure that we can actually compute this T' within the desired time complexity. Simple Gaussian elimination is too slow and therefore we give a slightly more tailored way to reduce the support in Section 3.2.2. We then prove an analogue of Lemma 3.2.1 in Section 3.2.3, and describe our final algorithm in Section 3.2.4.

3.2.2 Computing a Reduced Representative

In this subsection, we show how to find a function T' that M-represents T, while decreasing an upper bound on the size of the support of the function.

Definition 3.2.3. *For a function $f : \mathrm{col}(X) \to \mathbb{F}_p$ we say that $r \in X$ is a* reduced vertex *if $f(c) = 0$ whenever $c(r) = q$.*

The link between reduced vertices and the support of $T : \mathrm{col}(X) \to \mathbb{F}_p$ is explained as follows. If R is a set of reduced vertices of T, then we can compute a set, of size at most $(q-1)^{|R|}q^{|X|-|R|}$, of colorings containing the support of T. Indeed, we may restrict to the colorings that do not assign the color q to any vertex in R.

The following result allows us to turn vertices of degree 1 in H into reduced vertices. The assumption that the vertex has degree 1 will be useful in proving the result because it implies the associated compatibility matrix can be written as a Kronecker product with J_q and another matrix.

[1] Note that one could also sum $T_n'[y]$ over all $y \in \mathrm{col}(X_i)$. By summing T_{n-1}', the connection with representation is somewhat clearer.

Lemma 3.2.4. *There is an algorithm* ***Reduce*** *that, given a bipartite graph H with parts X, Y and associated color compatibility matrix M, a function $T : \mathrm{col}(X) \to \mathbb{F}_p$ with reduced vertices $R \subseteq X$ and a vertex $v \in X \setminus R$ of degree 1, outputs a function $T' : \mathrm{col}(X) \to \mathbb{F}_p$ with reduced vertices $R \cup \{v\}$ that is an M-representative of T.* ***Reduce*** *runs in time $O((q-1)^{|R|} q^{|X|-|R|})$.*

Proof. Let H be a bipartite graph with parts X, Y and associated color compatibility matrix M. Let $T : \mathrm{col}(X) \to \mathbb{F}_p$ be a function with reduced vertices $R \subseteq X$ and let vertex $v \in X \setminus R$ be a vertex of degree 1 in H. We need to find a function $T' : \mathrm{col}(X) \to \mathbb{F}_p$ with reduced vertices $R \cup \{v\}$ that M-represents T, in time $O((q-1)^{|R|} q^{|X|-|R|})$.

We may restrict to colorings x that do not assign value q to any element of R. There are at most $(q-1)^{|R|} q^{|X \setminus R|}$ such colorings. For the coloring x' obtained from x by changing the value of v to q, we set

$$T'[x] = \begin{cases} 0, & \text{if } x(v) = q, \\ T[x] - T[x'] & \text{otherwise.} \end{cases}$$

This computation is done in time linear in the number of the colorings x we consider, so the running time is $O((q-1)^{|R|} q^{|X \setminus R|})$.

First we will show that T' is an M-representative of T. Let $y \in \mathrm{col}(Y)$ be a coloring of the right hand side of the bipartite graph H. We need to show that

$$\sum_{\substack{x \in \mathrm{col}(X) \\ x \sim y}} T[x] \equiv_p \sum_{\substack{x \in \mathrm{col}(X) \\ x \sim y}} T'[x].$$

By definition,

$$\sum_{\substack{x \in \mathrm{col}(X) \\ x \sim y}} T'[x] = \sum_{\substack{x \in \mathrm{col}(X) \\ x \sim y \\ x(v) = q}} 0 + \sum_{\substack{x \in \mathrm{col}(X) \\ x \sim y \\ x(v) \neq q}} T[x] - T[x'].$$

Thus it remains to show that

$$\sum_{\substack{x \in \mathrm{col}(X) \\ x \sim y \\ x(v) \neq q}} -T[x'] \equiv_p \sum_{\substack{x \in \mathrm{col}(X) \\ x \sim y \\ x(v) = q}} T[x].$$

Let $x \in \mathrm{col}(X)$ with $x(v) = q$. We show the equality by proving that the number of times $T[x]$ appears on the left hand side equals the number of times $T[x]$ appears on the right hand side, modulo p. Let $w \in Y$ be the unique neighbor of the vertex v.

First assume that $x \sim y$. Then $y(w) \neq q$. If we adjust x to the coloring x_i, which is equal to x apart from assigning color i to v instead of q, then $x_i \sim y$ if and only if $i \neq y(w)$. Hence the term $-T[x]$ appears $q-2$ times on the left hand side, and $T[x]$ appears once on the right hand side. Since p divides $q-1$, we find $q-2 \equiv_p -1$ and hence both contributions are equal modulo p.

If $x \not\sim y$, then either x does not appear on both sides (because $x|_{X \setminus \{v\}}$ is already incompatible with y) or $y(w) = q$. If $y(w) = q$, then the term $T[x]$ appears $q-1 \equiv_p 0$ times on the left hand side by a similar argument as the above, and does not appear on the right hand side. This shows the claimed equality and finishes the proof. □

We say that a function $T : \mathrm{col}(X) \to \mathbb{F}_p$ is *fully reduced* if every vertex $v \in X$ of degree 1 is a reduced vertex of T. In order to keep the running time low, we will ensure that R is relatively large whenever we apply Lemma 3.2.4.

3.2.3 Computing T_i' from T_{i-1}'

We now show how to compute T_i' from T_{i-1}' and what effect this computation has on the reduced vertices.

Lemma 3.2.5. *Let $i \in [n-1]$. Suppose that T_{i-1}' is an M_{i-1}-representative of T_{i-1} and that T_{i-1}' is fully reduced. Given T_{i-1}' and a set R_{i-1} of reduced vertices for T_{i-1}', we can compute a function T_i' that is an M_i-representative of T_i in time $O((q-1)^{|R_{i-1}|}q^{|X_{i-1}|-|R_{i-1}|+1})$, along with a set R_i of reduced vertices for T_i' such that $|X_i \setminus R_i| \leq (\mathrm{ctw}-|R_i|)/2+1$.*

Proof. Analogous to Lemma 3.2.1, we define, for $x \in \mathrm{col}(X_i)$,

$$T_i'[x] = \sum_{\substack{z \in \mathrm{col}(X_{i-1}) \\ z \sim x}} T_{i-1}'[z].$$

Note that

$$\sum_{\substack{z \in \mathrm{col}(X_{i-1}) \\ z \sim x}} T_{i-1}'[z] = \sum_{\substack{z \in \mathrm{supp}(T_{i-1}') \\ z \sim x}} T_{i-1}'[z].$$

We compute T_i' from T_{i-1}' as follows. Let

$$\mathcal{S}_{i-1}' = \{c \in \mathrm{col}(X_{i-1}) : c(r) \neq q \text{ for all } r \in R_{i-1}\}.$$

By the definition of a reduced vertex, $\mathcal{S}_{i-1}'$ contains the support of T_{i-1}'. Recall that $X_i \setminus \{v_i\} \subseteq X_{i-1}$, so any $x \in \mathrm{col}(X_i)$ is determined if we provide colors for the vertices in $X_{i-1} \cup \{v_i\}$. For a color $c \in [q]$, let $f_c : \{v_i\} \to \{c\}$ be the function that assigns color c to v_i. For each $z \in \mathcal{S}_{i-1}'$, for each $c \in [q]$ for which $z \sim f_c$, we compute

$$x = (z \cup f_c)|_{X_i} \in \mathrm{col}(X_i)$$

and increase $T_i'[x]$ by $T_{i-1}'[z]$ if it has been defined already, and initialise it to $T_{i-1}'[z]$ otherwise. The remaining values are implicitly defined to 0. The running time is as claimed because $|\mathcal{S}_{i-1}'| \leq (q-1)^{|R_{i-1}|}q^{|X_{i-1}|-|R_{i-1}|}$ and $|[q]| \leq q$.

Next, we compute a set R_i of reduced vertices for T_i'. We set $R_i = X_i \setminus (A_i \cup B_i \cup \{v_i\})$, where

$$A_i = \{u \in X_i \setminus \{v_i\} : |N(u) \cap Y_i| \geq 2\}$$

and

$$B_i = \{u \in X_i \setminus \{v_i\} : |N(u) \cap Y_i| = 1 \text{ and } uv_i \in E\}.$$

It is easy to see that A_i and B_i are disjoint. Within the $(i-1)$th cut, each vertex in $A_i \cup B_i$ has at least two edges going across the cut, so $|R_i| + 2|A_i| + 2|B_i| \leq \mathrm{ctw}$. Therefore, $|X_i \setminus R_i| \leq (\mathrm{ctw}-|R_i|)/2+1$.

We now show that R_i is indeed a set of reduced vertices. Suppose not, and let $r \in R_i$ and $c \in \mathrm{col}(X_i)$ with $c(r) = q$ yet $T_i'[c] \neq 0$. Since $T_i'[c] \neq 0$, there exists $z \in \mathrm{col}(X_{i-1})$

with $z \sim c$ and $T'_{i-1}[z] \neq 0$. By definition $r \in X_i \setminus \{v_i\} \subseteq X_{i-1}$. Moreover, $z(r) = q$ since $z \sim c$ and $c(r) = q$. Therefore r is not reduced for T'_{i-1}. We now show r moreover has degree 1 in the bipartite graph between X_{i-1} and Y_{i-1} (corresponding to the $(i-1)$th cut), contradicting our assumption that T'_{i-1} is fully reduced. Since $r \notin A_i \cup B_i$, it has at most one edge going over the $(i-1)$th cut. Moreover, $r \in X_i \setminus \{v_i\} \subseteq L_i$, and so it has at least one edge to $Y_i \subseteq Y_{i-1}$. So r has exactly one neighbor in Y_{i-1}.

It remains to prove that T'_i is M_i-representative of T_i. Recall that we work over $\mathbb{F}_p$ and therefore all equalities and computations below take place modulo p. Let $y \in \mathrm{col}(Y_i)$. We need to show that

$$\sum_{x \in \mathrm{col}(X_i)} M_i[x,y]T_i[x] \equiv_p \sum_{x \in \mathrm{col}(X_i)} M_i[x,y]T'_i[x]. \tag{3.1}$$

By Lemma 3.2.1, for all $x \in \mathrm{col}(X_i)$,

$$T_i[x] = \sum_{z \in \mathrm{col}(X_{i-1})[x]} T_{i-1}[z],$$

and we crucially use in the computation below that we used the same expression when defininin T'_i, which will allow us to exploit the fact that T'_{i-1} is an M_{i-1}-representative of T_{i-1}. We find

$$\begin{aligned}\sum_{x \in \mathrm{col}(X_i)} M_i[x,y]T_i[x] &= \sum_{x \in \mathrm{col}(X_i)} M_i[x,y]\left(\sum_{z \in \mathrm{col}(X_{i-1})[x]} T_{i-1}[z]\right) \\ &= \sum_{z \in \mathrm{col}(X_{i-1})} \sum_{\substack{x \in \mathrm{col}(X_i) \\ x \sim z \\ x \sim y}} T_{i-1}[z].\end{aligned}$$

In the second sum, recall that each $x \in \mathrm{col}(X_i)$ compatible with z is of the form $x = (z \cup f_c)|_{X_i}$ with f_c, z compatible and $f_c : \{v_i\} \to \{c\}$ for some $c \in [q]$. We find that x is compatible with y if and only if f_c is compatible with y and z is compatible with y, so the previous expression equals

$$\begin{aligned}&= \sum_{z \in \mathrm{col}(X_{i-1})} \sum_{f_c \in \mathrm{col}(\{v_i\})} T_{i-1}[z]1_{z \sim f_c}1_{z \sim y}1_{f_c, \sim y} \\ &= \sum_{f_c \in \mathrm{col}(\{v_i\})} 1_{f_c \sim y} \sum_{z \in \mathrm{col}(X_{i-1})} T_{i-1}[z]1_{z \sim f_c}1_{z \sim y},\end{aligned}$$

where 1_b denotes the indicator variable which is equal to 1 if condition b is true and equals 0 otherwise. Since $Y_{i-1} \subseteq Y_i \cup \{v_i\}$, we can consider $y' = (f_c \cup y)|_{Y_{i-1}}$ and find $M_{i-1}[z, y'] = 1$ exactly when z is compatible with both f_c and y. Using this to rewrite the previous expression, and then noting that $y' \in \mathrm{col}(Y_{i-1})$ and T'_{i-1} is a M_{i-1}-representative of T_{i-1}, we find

$$\begin{aligned}&= \sum_{f_c \in \mathrm{col}(\{v_i\})} 1_{f_c \sim y} \sum_{z \in \mathrm{col}(X_{i-1})} T_{i-1}[z]M_{i-1}[z, (f_c \cup y)|_{Y_{i-1}}] \\ &\equiv_p \sum_{f_c \in \mathrm{col}(\{v_i\})} 1_{f_c \sim y} \sum_{z \in \mathrm{col}(X_{i-1})} T'_{i-1}[z]M_{i-1}[z, (f_c \cup y)|_{Y_{i-1}}],\end{aligned}$$

which for the same reasons as above

$$\begin{aligned}
&= \sum_{f_c \in \mathrm{col}(\{v_i\})} 1_{f_c \sim y} \sum_{z \in \mathrm{col}(X_{i-1})} T'_{i-1}[z] 1_{z \sim f_c} 1_{z \sim y} \\
&= \sum_{z \in \mathrm{col}(X_{i-1})} \sum_{f_c \in \mathrm{col}(\{v_i\})} T'_{i-1}[z] 1_{z \sim f_c} 1_{z \sim y} 1_{f_c \sim y} \\
&= \sum_{z \in \mathrm{col}(X_{i-1})} \sum_{\substack{x \in \mathrm{col}(X_i) \\ x \sim z \\ x \sim y}} T'_{i-1}[z] \\
&= \sum_{x \in \mathrm{col}(X_i)} \left(\sum_{z \in \mathrm{col}(X_{i-1})[x]} T'_{i-1}[z] \right) M_i[x, y] \\
&= \sum_{x \in \mathrm{col}(X_i)} M_i[x, y] T'_i[x].
\end{aligned}$$

This shows (3.1) and completes the proof. □

3.2.4 The Final Algorithm

We now conclude by describing the full algorithm.

We initialize $T_1 = \mathbf{1}$, the all-ones vector. Indeed, each $x \in \mathrm{col}(\{v_1\})$ has a unique extension to G_1 (namely itself). We then repeatedly apply the **Reduce** algorithm from Lemma 3.2.4 until we obtain a fully reduced function T'_1 that is an M_1-representative of T_1, with some set of reduced vertices R_1. For $i = 2, \ldots, n$, we repeat the following two steps.

1. Apply Lemma 3.2.5 with inputs (T'_{i-1}, R_{i-1}) in order to obtain the vector T'_i that is an M_i-representative of T_i, and a set of reduced vertices R_i for T'_i.
2. While $X_i \setminus R_i$ has a vertex v of degree 1, apply the **Reduce** algorithm from Lemma 3.2.4 to (T'_i, R_i), and add v to R_i.

At the end of step 2, we obtain a fully reduced function T'_i that is an M_i-representative of T_i. Moreover, the set R_i of reduced vertices has only increased in size compared to the set we obtained in step 1. We apply Lemma 3.2.4 at most $|X_i|$ times in the second step.

We eventually compute T'_{n-1}, which is an M_{n-1}-representative of T_{n-1} with a fully reduced set R_{n-1}. We output

$$\sum_{y \in \mathrm{col}(Y_{n-1})} \sum_{x \in \mathrm{col}(X_{n-1})} T'_{n-1}[x] M_{n-1}[x, y].$$

Since T'_{n-1} is an M_{n-1}-representative of T_{n-1}, this gives the number of list colorings of G modulo p. We may compute the expression above efficiently by reducing the second summation to the colorings in

$$\mathcal{S}'_{n-1} = \{c \in \mathrm{col}(X_{n-1}) : c(r) \neq q \text{ for all } r \in R_{n-1}\}.$$

The total running time is now bounded by

$$C \sum_{i=1}^{n-1} |X_i|(q-1)^{|R_i|} q^{|X_i|-|R_i|}$$

for some constant $C > 0$. By Lemma 3.2.5, $|X_i \setminus R_i| \leq (\mathrm{ctw} - |R_i|)/2 + 1$ for all $i \in [n-1]$. For $q \geq 3$, $q^{1/2} < q - 1$ and so

$$(q-1)^{|R_i|} q^{|X_i|-|R_i|} \leq q(q-1)^{|R_i|} (q^{1/2})^{\mathrm{ctw}-|R_i|} < q(q-1)^{\mathrm{ctw}}.$$

This shows the total running time is of order $O((q-1)^{\mathrm{ctw}} n)$. This finishes the proof of Theorem 3.1.1.b.

3.3 Lower Bounds

In this section we give a reduction from $\#_p$SAT to $\#_p$List q-Coloring, the problem of counting the number of valid list q-colorings of a given graph G with color lists $(L_v)_{v \in V(G)}$. We use this to conclude the lower bounds of Theorem 3.1.1 and Theorem 3.1.2.

We can use existing results to find lower bounds for $\#_p$SAT, as follows. There exists an efficient reduction from SAT to the problem $\#_p$SAT of counting the number of satisfying assignments for a given boolean formula modulo p [41]. There also exists a reduction from SAT to CSP (q, r), which preserves the number of solutions [81]. Putting these two together gives a reduction from SAT to $\#_p$CSP (q, r).

3.3.1 Controlling the Number of Extensions Modulo p

Our main gadget can be attached to a given set of vertices, and has the property that for each precoloring of the 'glued on' vertices, there is a specified number of extensions. This is made precise in the result below.

Theorem 3.3.1. *Let $k \in \mathbb{N}$ and $f : [q]^k \to \mathbb{N}$. There exist a graph G_f, a set of vertices $B = \{b_1, \ldots, b_k\} \subseteq V(G_f)$ of size k and lists $(L_v)_{v \in V(G_f)}$, such that for any $\alpha \in [q]^k$, there are exactly $f(\alpha)$ list q-colorings c of G_f with $c(b_i) = \alpha(i)$ for all $i \in [k]$. Additionally, $|V(G_f)| \leq 20kq^{k+1} \max(f)$ and G_f has cutwidth at most $6kq^{k+2}$.*

We first reduce the lists of each b_i to $\{1, 2\}$ using the following gadget.

Lemma 3.3.2. *Let $q, k \in \mathbb{N}$ and let $a \in [q]$. There is a graph G with $b, b' \in V(G)$ and color lists $L_v \subseteq [q]$ for $v \in V(G)$, such that $L_b = [q]$ and the following two properties hold:*

- *for all $c_b \in [q]$, there is a unique list coloring c of G with $c(b) = c_b$,*
- *for all list colorings c of G, if $c(b) = a$, then $c(b') = 1$ and if $c(b) \neq a$ then $c(b') = 2$.*

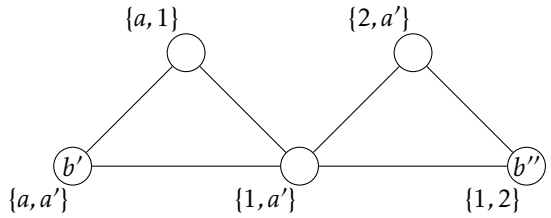


Figure 3.2: A gadget to 'relabel colors'. It has two special vertices b' and b'', and lists are depicted with sets. For any list coloring c of the depicted gadget, if $c(b') = a$, then $c(b'') = 1$ and if $c(b') = a'$, then $c(b'') = 2$. In both cases, there is a unique way to color the remaining vertices.

Proof. We first note that it is easy to 'relabel colors', as shown in the construction[2] in Figure 3.2. We can therefore first make a gadget for which b' has color list $\{a, a'\}$ for some $a' \neq a$, and then relabel a, a' to $1, 2$. By symmetry, we can therefore assume that $a = 1$ (or simply replace 1 with a and 2 with a' in the argument below). Let

$$V = \{b, b'\} \cup \{s_i : i = 2, \ldots, q\} \cup \{t_i : i = 2, \ldots, q\}$$

and

$$E = \{s_i b : i = 2, \ldots, q\} \cup \{s_i b' : i = 2, \ldots, q\} \cup \{s_i t_j : i, j = 2, \ldots, q\}.$$

Now let $L_b = [q]$, $L_{b'} = \{1, 2\}$ and $L_{t_i} = L_{s_i} = \{1, i\}$ for $i \in \{2, \ldots, q\}$. A depiction is given in Figure 3.3.

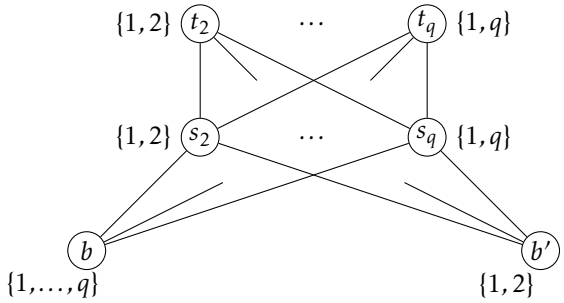


Figure 3.3: The construction of the list coloring instance of the proof of Lemma 3.3.2.

If a list coloring c of G satisfies $c(b) = 1$, then $c(s_i) = i$ and thus $c(t_i) = 1$ for each $i \in \{2, \ldots, q\}$. In particular $c(s_2) = 2$ and $c(b') = 1$.

If $c_b \in \{2, \ldots, q\}$, then any list coloring c with $c(b) = c_b$ satisfies $c(s_i) = 1$ and $c(t_i) = i$ for all $i \in \{2, \ldots, q\}$, and so $c(b') = 2$.

[2] If $a = 1$ or $a' = 2$ we slightly change the construction by removing the top left or top right vertex respectively.

This proves that, starting with the color $c_b \in [q]$ for b, there is always a unique extension to a list coloring of G, and this satisfies the property that vertex b' receives color 1 if $c_b = 1$, and receives color 2 otherwise. □

We use of the following construction to control the number of extensions for a given coloring of B.

Lemma 3.3.3. *Let $k, \ell \in \mathbb{N}$. There is a graph G, a subset of vertices $B = \{b_1, \dots, b_k\} \subseteq V$ of size k, and color lists L_v for all $v \in V(G)$ such that:*

- *$L_{b_i} = \{1, 2\}$ for all $i \in \{1, \dots, k\}$,*
- *there are exactly ℓ list colorings c of G with $c(B) = \{1\}$,*
- *for each partial coloring c_B of B with $c_B(B) \neq \{1\}$, there is a unique extension of c_B to a list coloring of G.*

Proof. We start with $V = B$ and add a path[3] $w_1, \dots, w_{\ell-1}$ with color lists

$$L_{w_i} = \begin{cases} \{2,3\} & \text{if } i \equiv_3 1, \\ \{1,3\} & \text{if } i \equiv_3 2, \\ \{1,2\} & \text{if } i \equiv_3 0, \end{cases}$$

and add edges $b_i w_1$ for $i = 1, \dots, k$. A depiction is given in Figure 3.4.

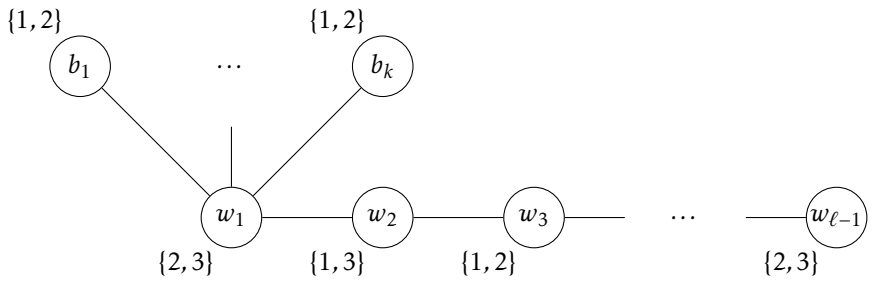


Figure 3.4: Construction in the proof of Lemma 3.3.3 when $\ell \equiv_3 2$.

If a list coloring c satisfies $c(b_i) = 2$ for some $i \in [k]$, then $c(w_1) = 3$, $c(w_2) = 1$, $c(w_3) = 2$ etcetera. Hence there is a unique extension of any partial coloring of B that assigns color 2 somewhere.

If $c(b_i) = 1$ for all $i \in [k]$, then we have a choice for the color of w_1. If $c(w_1) = 3$ then we get the same propagation as before, however if $c(w_1) = 2$, then we have a choice for the color of w_2. Using a simple induction argument we find that the number of possible list colorings with $c(B) = 1$ equals ℓ. □

We are now ready to construct the main gadget.

[3] When $\ell = 1$, we add no vertices of the form w_i and the statements of the lemma immediately follow.

Proof of Theorem 3.3.1. Let $f \in [q]^k$. We create vertices $b_1, \ldots, b_k$ and give them all $[q]$ as list.

For each partial coloring $\alpha \in [q]^k$, we create a graph G_α that contains $b_1, \ldots, b_k$ in their vertex set, but the graphs are on disjoint vertex sets otherwise (we 'glue' the graphs on the special vertices $b_1, \ldots, b_k$). It suffices to show that we can find lists for the 'private' vertices of G_α such that the number of extensions of a coloring c_B of B is 1 if $c_B(b_i) \neq \alpha(i)$ for some $i \in [k]$, and $f(\alpha)$ otherwise. The resulting gadget will then have $1 \cdot 1 \cdot \ldots \cdot 1 \cdot f(\alpha) = f(\alpha)$ possible extensions for the precoloring α, as desired.

We now turn to constructing the gadget G_α for a fixed coloring $\alpha \in [q]^k$. We first reduce to the case in which each b_i has $\{1, 2\}$ as list. Let $i \in [k]$. Using Lemma 3.3.2 with $a = \alpha(i)$, we obtain a gadget $H_{b,b'}$ and identify the special vertex b with b_i. For each α, we obtain a new set of vertices $b'_1, \ldots, b'_k$ with lists $\{1, 2\}$. We then glue these onto the special vertices from a gadget obtained by applying Lemma 3.3.3 with $\ell = f(\alpha)$. If $b_1, \ldots, b_k$ are colored as specified by α, then $b'_1, \ldots, b'_k$ all receive color 1 and G_α has $f(\alpha)$ possible extensions; however if some b_i receives the wrong color, the corresponding b'_i receives color 2 and there is a unique extension to the rest of G_α.

It remains to show the bounds on the number of vertices and the cutwidth. We give the very rough upperbound of $6kq^{k+2}$ on the cutwidth. The gadget from Lemma 3.3.2 has cutwidth at most $q^2 + 6$ (since this is an upper bound on the number of edges in that construction). The gadgets from Lemma 3.3.3 have cutwidth at most k. A final cut decomposition can be obtained by first enumerating the vertices in B, and then adding the cut decompositions of each G_α, one after the other.

Finally the number of vertices of the graph is upper bounded by q^k times the maximum number of vertices of the graph G_α. The gadget of Lemma 3.3.2 has at most $2q+6$ vertices and there are k of them, so they contribute at most $12kq$ vertices. The gadgets from Lemma 3.3.3 add at most $f(\alpha)$ vertices. In total,

$$|V(G_f)| \leq 20kq^{k+1} \max(f). \qquad \square$$

3.3.2 Reduction for Counting q-Colorings Modulo p

Now that we have constructed the gadgets we need, we can prove the main result of this section, Theorem 3.1.1.c.

Theorem 3.1.1.c. *(restated) If p does not divide $q-1$, there is no $\varepsilon > 0$ for which there exists a $(q-\varepsilon)^{\mathrm{ctw}} n^{O(1)}$ time algorithm that counts the number of list q-colorings modulo p, assuming SETH.*

Suppose now that p divides $q-1$. Let $q' = q-1$. Then p does not divide $q'-1 = q-2$ and so the result above applies. Noting that any algorithm for #List q-Coloring also works for #List q'-Coloring, we find the Theorem 3.1.1.a as a corollary.

Theorem 3.1.1.a. *(restated) If p divides $q-1$, there is no $\varepsilon > 0$ for which there exists a $(q-1-\varepsilon)^{\mathrm{ctw}} n^{O(1)}$ time algorithm that counts the number of list q-colorings modulo p assuming SETH.*

The reduction in the proof of Theorem 3.1.1.c starts from a $\#_p$CSP (q, r) instance, which is a so called *constraint satisfaction problem* (CSP). Informally, a CSP asks if

there is an assignment of values from a given domain to a set of variables such that they satisfy a given set of relations. We denote by CSP(q,r) the CSP with domain $[q]$ and constraints of arity at most r. We use #CSP (q,r) to denote the problem of counting the number of solutions of a given instance of CSP(q,r) and $\#_p$CSP (q,r) to denote the same problem, except with counting modulo p. The following result, based on a theorem from [81], indicates the complexity of $\#_p$CSP (q,r).

Theorem 3.3.4 ([81], Theorem 2.5)**.** *For each prime p, for every integer $q \geq 2$ and $\varepsilon > 0$ there is an integer r, such that the following holds. Unless the SETH fails, $\#_p$CSP (q,r) with n variables and m constraints cannot be solved in time $(q-\varepsilon)^n(n+m)^{O(1)}$.*

This theorem follows from the proof of [81, Theorem 2.5], since their reduction preserves the number of solutions.

Proof of Theorem 3.1.1.a. Let $q \in \mathbb{N}$ and let p be a prime that does not divide $q-1$. Fix $\epsilon > 0$ and let r be given from Theorem 3.3.4. We will reduce a given instance of $\#_p$CSP (q,r) with constraints $C_1,\ldots,C_m$ and variables $x_1,\ldots,x_n$ to an instance (G,L) of $\#_p$List q-Coloring on $O_{p,r,q}(nm)$ vertices of cutwidth $n + O_{p,r,q}(1)$. A $(q-\varepsilon)^{\mathrm{ctw}}n^{O(1)}$ algorithm for $\#_p$List q-Coloring then implies a $(q-\varepsilon)^n(n+m)^{O(1)}$ for $\#_p$CSP (q,r), which contradicts Theorem 3.3.4.

The general structure of the reduction will be similar to typical gridshaped SAT reduction. We will use the gadgets constructed in Theorem 3.3.1 both to check the whether a clause is satisfied and to ensure proper propagation of the variable assignments.

We think of the graph G as containing $2m$ columns and n rows: for each constraint C_j, and for each variable x_i, we create two vertices $s_{i,j}$ and $t_{i,j}$ (where $j \in [m]$ and $i \in [n]$), which all get $\{1,\ldots,q\}$ as list. For all $j \in [m-1]$, we place an edge between $t_{i,j}$ and $s_{i,j+1}$.

The color assigned to $s_{i,1}$ will be interpreted as the value assigned to variable x_i. Fix $j \in [m]$. We create gadgets on some vertex set V_j using Theorem 3.3.1, that are 'glued' on subsets of vertices from $S_j = \{s_{i,j}, t_{i,j} : i \in [n]\}$ as follows.

1. For each $i \in [n]$, if $j < m$, we create a gadget on boundary set $\{s_{i,j}, t_{i,j}\}$ which ensures that we may restrict to counting list colorings c of (G,L) with $c(s_{i,j}) = c(s_{i,j+1})$.

2. We create a gadget on the (at most r) $s_{i,j}$ corresponding to the variables involved in the jth constraint, for which the number of extensions of any coloring of the boundary to this gadget is equivalent to 0 modulo p whenever the jth constraint is not satisfied, and equal to one otherwise.

A broad overview of the construction is depicted in Figure 3.5.

For the first property, we need the fact that p does not divide $q-1$: this ensures that the color compatibility matrix of a single edge is invertible, which will allow us to 'transfer all information about the colors'.

We start by describing the gadgets that perform this information transfer (item 1). Let $j \in [m-1]$ and $i \in [n]$. We will apply Theorem 3.3.1 to a function $f_{i,j}$ with boundary set $B_{i,j} = (s_{i,j}, t_{i,j})$ and $\max(f_{i,j}) = p$, resulting in a graph on $O(q^3p)$ vertices. Let

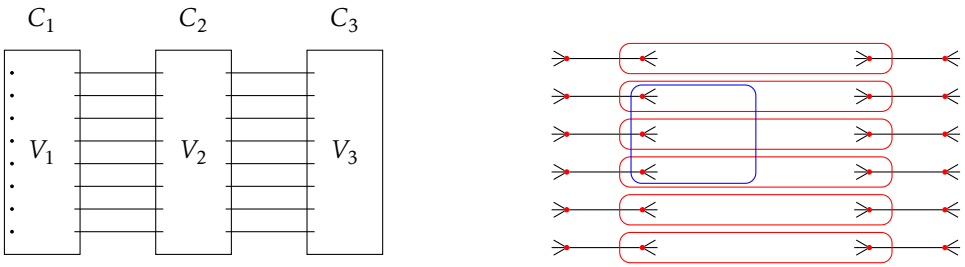


Figure 3.5: A sketch overview of the construction is given on the left-hand side and a more detailed view of two of the columns is given on the right-hand side. The red areas ensure the preservation of information, as described in point 1. The blue area checks whether the clause is satisfied, as described in point 2.

J_1 be the $q \times q$ coloring compatibility matrix of a single edge, and let J_1^{-1} denote its inverse over $\mathbb{F}_p$ (that is, $J_1^{-1} J_1 \equiv_p I_q$, the $q \times q$ identity matrix). We 'choose a representative' $\widetilde{J_1}$, which has entries in $\{1, \ldots, p\}$ that are equivalent to those in J_1^{-1} modulo p. For $c_1, c_2 \in [q]$ possible colors for $s_{i,j}$ and $t_{i,j}$ respectively, we set

$$f_{i,j}(c_1, c_2) = \widetilde{J_1}[c_1, c_2].$$

Let $V_{i,j}$ denote the vertices in the gadget obtained by applying Theorem 3.3.1 to $(f_{i,j}, B_{i,j})$ that are not in $B_{i,j}$. Let $c_1, c_3 \in [q]$. The number of list colorings c of the graph induced on $B_{i,j} \cup V_{i,j} \cup \{s_{i,j+1}\}$ with $c(s_{i,j}) = c_1$ and $c(s_{i,j+1}) = c_3$ is equal to

$$\sum_{c_2 \in [q]} f_{i,j}(c_1, c_2) J_1[c_2, c_3] = (\widetilde{J_1} J_1)[c_1, c_3],$$

since for any coloring c_2 we have $f_{i,j}(c_1, c_2) J_1[c_2, c_3]$ such colorings with $c(t_{i,j}) = c_2$, by definition of f_i, j and J_1. Therefore, modulo p this number of extensions is equal to 1 if $c_1 = c_3$ and 0 otherwise, as desired.

We now describe the gadgets that check the constraints (item 2). Let $j \in [m]$ and let $i_1, \ldots, i_\ell$ be given so that the jth constraint only depends on the variables $x_{i_1}, \ldots, x_{i_\ell}$ (where by assumption $\ell \leq r$). We will apply Theorem 3.3.1 to a function g_j with boundary set $B_j = (s_{i_1,j}, \ldots, s_{i_\ell,j})$ and $\max(g_j) = p$, resulting in a graph on $O(\ell q^{\ell+1} p) = O(r q^{r+1} p)$ vertices. We set $g_j(c_1, \ldots, c_\ell)$ to be equal to 1 if the assignment $(c_1, \ldots, c_\ell)$ to $(x_{i_1}, \ldots, x_{i_\ell})$ satisfies the jth constraint, and p otherwise. This ensures that modulo p the number of extentions is 1 if all constraints are satisfied and 0 otherwise.

We obtain a cutwidth decomposition of the graph by first partially ordering the vertices as

$$S_1 \cup V_1,\ S_2 \cup V_2, \ldots, S_m \cup V_m.$$

Within $S_j \cup V_j$, we first list $s_{1,j}, t_{1,j}$ and the vertices in the gadget that has those vertices as boundary set, and then repeat this for $s_{2,j}, t_{2,j}$, et cetera. Finally, we order the vertices in the gadget that verifies the jth constraint arbitrarily. At each point, the cutwidth is bounded by $n + 6 r q^{r+2}$. □

3.4 Connected Edgesets

The results for coloring that we found before also have implications for the problem of counting Connected Edgesets. We start by deriving a lower bound, based on our lower bound for coloring and then give a matching algorithm.

3.4.1 Lower Bound

We first extend the lower bound of Theorem 3.1.1.c to counting connected edge sets via the following problem.

Definition 3.4.1. *Given a graph G, two q-colorings c and c' are equivalent if there is some permutation $\pi : [q] \to [q]$ such that $c = \pi \circ c'$. We will refer to these equivalence classes as* essentially distinct q-colorings *and denote the problem of counting the number of essentially distinct q-colorings modulo a prime p by $\#_p$*Essentially distinct q-coloring.

A simple reduction now gives us the following lower bound for $\#_p$Essentially distinct q-coloring.

Corollary 3.4.2. *Let p be a prime and $q \in \mathbb{N}$ an integer such that p does not divide $q-1$. Assuming SETH, there is no $\epsilon > 0$ for which there exists an algorithm that counts the number of essentially distinct q-colorings mod p for a given n-vertex graph that is not $(q-1)$-colorable, with a given cut decomposition of cutwidth* ctw*, in time $(q-\epsilon)^{\mathrm{ctw}} n^{O(1)}$.*

Proof. Let (G, L) be an instance of list coloring with cut decomposition $v_1, \ldots, v_n$. We construct an instance of $\#_p$Essentially distinct q-coloring. The graph G' has vertex set

$$V(G') = V(G) \cup \{u_c^i : c \in [q], i \in [n]\}.$$

We add edges such that the vertices $\{u_c^i : c \in [q]\}$ induce a q-clique for all $i \in [n]$, and for $i \in [n-1]$ we add the edges $u_c^i u_{c'}^{i+1}$ for all $c \neq c'$. This ensures that, if u_c^1 is colored c, then u_c^i is colored c for all $i \in [n]$. We now also add edges $u_c^i u_i$ for all $c \notin L_{v_i}$. See Figure 3.4.1 for an example.

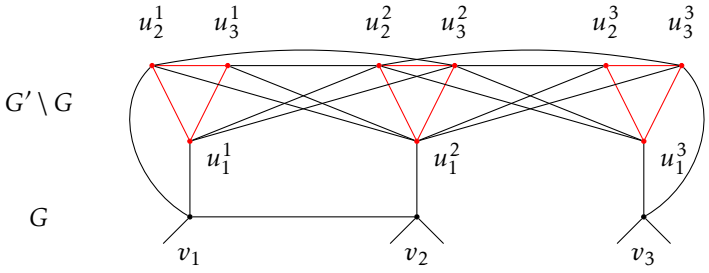


Figure 3.6: Example of the construction on (a part of) a graph G, with the cliques indicated in red. In this case $q = 3$ and we have $L_{v_1} = \{3\}, L_{v_2} = \{2, 3\}$ and $L_{v_3} = \{2\}$.

Our new cut decomposition is

$$v_1, u_1^1, \ldots, u_q^1, v_2, u_1^2, \ldots, u_q^{n-1}, v_n, u_1^n, \ldots, u_q^n.$$

Note that $\mathrm{ctw}(G') \leq \mathrm{ctw}(G) + q^2$, $|V(G')| \leq (q+1)|V(G)|$ and that G' is not $(q-1)$-colorable. By Theorem 3.1.1, it suffices to show that the number of essentially distinct q-colorings of G' equals the number of list q-colorings of (G, L). We will do this by defining a bijective map.

Let α be a list coloring of (G, L). Then we can color G' by setting $\alpha'(v) = \alpha(v)$ for $v \in V(G)$ and $\alpha'(u_c^i) = c$ for $c \in [q]$ and $i \in [n]$. This gives us a mapping $\gamma : \alpha \mapsto \overline{\alpha'}$, where $\overline{\alpha'}$ is the equivalence class of α'. We find an inverse map by first fixing a representative α' for $\overline{\alpha'}$, such that $\alpha'(u_c^1) = c$ for $c \in [q]$. We can do this since $G'[\{u_1^1, \ldots, u_q^1\}]$ is a clique and thus each u_c^1 must get a unique color. Also note that since every color is now used, the rest of the coloring is also fixed and thus we find a unique representative this way. We now map $\overline{c'}$ to $c'|_{V(G)}$. Note that these two maps are well defined and compose to the identity map. We conclude that the number of list colorings of (G, L) is equal to the number of essentially distinct colorings of G'. □

To achieve the lower bound in Theorem 3.1.2, we use an existing argument from [5] to extend this bound to $\#_p$Connected Edge Sets. For this we recall the following Definition.

Definition 2.6.2. *The* k-stretch *of a graph G is the graph obtained from G by replacing each edge with a path of length k. The* k-thickening *of a graph G is the graph obtained from G by replacing each edge with k parallel edges. We denote the k-stretch of G by ${}^kG$ and the k-thickening as by ${}_kG$.*

Note that ${}^kG$ has the same cutwidth as G.

Theorem 3.4.3. *Assuming SETH, there is no $\epsilon > 0$ for which there exists an algorithm that counts the number of spanning connected edge sets mod p of n-vertex graphs of cutwidth at most* ctw *in time $O((p-\epsilon)^{\mathrm{ctw}} n^{O(1)})$.*

Proof. This proof closely follows a reduction from Annan [5], using ideas from Jaeger, Vertigan and Welsh [106].

Let G be any graph with cutwidth ctw and let p be a prime. Note that the number of spanning connected edgesets of G is equal to the value of $T(G; 1, 2)$, the Tutte polynomial of G, computed at the point $(1, 2)$. The following equality is found in ([106], proof of Theorem 2)

$$T({}^kG; a, b) = (1 + a + \cdots + a^{k-1})^{n-r(G)} T\left(G; a^k, \frac{b + a + \cdots + a^{k-1}}{1 + a + \cdots + a^{k-1}}\right).$$

Choosing $a = 1, b = 2$ and $k = p - 1$, gives

$$T({}^{p-1}G; 1, 2) = (p-1)^{n-r(G)} T\left(G; 1, \frac{2 + p - 2}{p-1}\right) \equiv_p (-1)^{n-r(G)} T(G; 1-p, 0).$$

Here we use the fact that we have $P(x,y) \equiv_p P(x+tp, y+sp)$ for any multivariate polynomial, for any $s,t \in \mathbb{Z}$. We find that, since the k-stretch of a graph G has the same cutwidth as G, an algorithm that counts the number of spanning connected edgesets (mod p) on a graph with bounded cutwidth, also gives the Tutte polynomial at $(1-p,0)$ mod p.

Using another well known interpretation of the Tutte polynomial [135] we can relate the value $T(G;1-p,0)$ to the chromatic polynomial $P(G;p)$ as follows:

$$P(G;p) = (-1)^{r(G)} p^{k(G)} T(G;1-p,0).$$

Note that we may assume that the number of connected components $k(G) = 1$ (and thus $r(G) = n-1$), since the number of spanning connected edgesets is trivially 0 if G has more than one component. We want to get rid of the remaining factor of p (since we will work mod p). To do this we will count the number of essentially distinct colorings using exactly p colors instead. This will turn out to be at least as hard as counting all colorings.

Let G be a graph that is not $(p-1)$-colorable. With this assumption, the number of p colorings of G is $p!$ times the number $C_p(G)$ of essentially distinct p-colorings of G, since any coloring uses all colors and thus can be mapped to $p!$ equivalent colorings by permuting the colors. So

$$(-1)^{n-1} p T(G,1-p,0) = P(G;p) = p(p-1)! C_p(G),$$

which holds over the real numbers hence we may divide both sides by p. By Wilson's Theorem $(p-1)! \equiv_p -1$, so we find

$$(-1)^{n-1} T(G,1-p,0) \equiv_p -C_p(G).$$

Hence we can use the number of spanning connected edgesets (mod p) of the $(p-1)$-stretch of G to find the Tutte polynomial at $(1-p,0)$ mod p and then also the number of essentially distinct colorings. The claimed result now follows from Corollary 3.4.2 (with $q = p$). □

3.4.2 Algorithm

We now give an algorithm that matches the lower bound we just showed.

Theorem 3.4.4. *Let p be a prime number. There is an algorithm that counts the number of connected edge sets modulo p of n-vertex graphs of treewidth* tw *in time $p^{\mathrm{tw}} n^{O(1)}$.*

Together with Theorem 3.4.3 this proves Theorem 3.1.2.

The algorithm uses dynamic programming on the tree decomposition, reminiscent of cut-and-count. We will first recall some additional notions about tree decompositions.

Let G be an n-vertex graph with a tree decomposition $(T, (B_x)_{x \in V(T)})$ such that T is a rooted tree and $|B_x| \leq \mathrm{tw}+1$ for all $x \in V(T)$. Since T is rooted, we can consider the children and descendants of x in T. In polynomial time we can adjust the tree decomposition to be a nice tree decomposition, as described in Definition 2.4.3. This

means that we may assume that such a decomposition is given. We write G_x for the graph on the vertices in the bags that are a descendant of x in T, which has as edge set those edges that have been 'introduced' by some bag below B_x.

Proof of Theorem 3.4.4. We assume that a nice tree decomposition $(T,(B_x)_{x\in V(T)})$ for G has been given. For each $x \in V(T)$ and each partition $X_1,\dots,X_p$ of B_x, we define

$$T_x[X_1,\dots,X_p] = |\{(X,V_1,\dots,V_p) : V_1 \sqcup \dots \sqcup V_p = V(G_x), \\ X \subseteq E(G_x) \setminus (\cup_{i<j} E(V_i,V_j))\}|,$$

where $\sqcup$ denotes the disjoint union (that is, we take the union and assume the sets are disjoint). Thus, we count the number of edge sets X of G_x and number of vertex partitions P of G_x such that X does not cross P and P induces the partition $(X_1,\dots,X_p)$ on B_x. Note that this is not done modulo p, but over the real numbers, and also that we do not require the edgesets X_i to be connected spanning edgesets.

We claim that for r the root of T,

$$\frac{1}{p}\sum_{R_1,\dots,R_p} T_r[R_1,\dots,R_p]$$

equals the number of connected spanning edge sets of G modulo p, where the sum is over the possible partitions of B_r. Note that $\sum_{R_1,\dots,R_p} T_r[R_1,\dots,R_p]$ counts the number of tuples $(X,V_1,\dots,V_p)$ such that X is an edge set of G that does not cross the partition $(V_1,\dots,V_p)$. For an edge set X of G, let $U_1,\dots,U_k$ denote the connected components (which may be single vertices) of $G[X]$, the graph with vertex set $V(G)$ and edge set X. The number of partitions $(V_1,\dots,V_p)$ of G for which X respects the partition equals p^k: we assign each connected component to one of the p sets of the partition. Therefore, the contribution of each X to $\sum_{R_1,\dots,R_p} T_r[R_1,\dots,R_p]$ is divisible by p, and is divisible by p^2 if and only if $G[X]$ is not connected. This proves the claim since $G[X]$ is connected if and only if X is connected and spanning.

What remains to show is that we can calculate the table entries in the claimed time complexity. If $x \in V(T)$ is a leaf, then we can calculate the table entry naively. We now assume that all strict descendants of x already have their table entry calculated. We consider two cases.

Join bags Suppose first that B_x is a 'join bag': x has two children y and z with $B_x = B_y = B_z$, and $E(G_y) \cap E(G_y) = \emptyset$. For any partition $(X_1,\dots,X_p)$ of B_x, we can also consider this as a partition of the bags B_y and B_z. We claim that

$$T_x[X_1,\dots,X_p] = T_y[X_1,\dots,X_p] \cdot T_z[X_1,\dots,X_p].$$

This follows from the fact that any $(V_1,\dots,V_p,X)$ counted for $T_x[X_1,\dots,X_p]$, is determined by the following parts: the partition $X_1,\dots,X_p$ of B_x (which is 'fixed'), the remaining partition of the vertices in

$$V(G_x) \setminus V(B_x) = (V(G_y) \setminus V(B_x)) \sqcup (V(G_z) \setminus V(B_x))$$

(where we obtain a disjoint union by the standard tree decomposition properties) and the edge set $X \subseteq E(G_x) = E(G_y) \sqcup E(G_z)$ (where we find a disjoint union by our additional 'edge-introduce' property).

Single child bags Suppose now that x has a single child y. Fix a partition $P_x = (X_1, \ldots, X_p)$ for B_x and consider a partition $P_y = (Y_1, \ldots, Y_p)$ of B_y that agrees with the partition P_x, in the sense that $X_i \cap Y_j = \emptyset$ for $i \neq j$. Note that since $|B_y \setminus B_x| \leq 1$, there are at most p partitions P_y that agree with P_x. We can therefore efficiently compute

$$C \cdot \sum_{P_y : P_y \sim P_x} T_y[Y_1, \ldots, Y_p], \tag{3.2}$$

where the sum is over those $P_y = (Y_1, \ldots, Y_p)$ that agree with $P_x = (X_1, \ldots, X_p)$, and where $C = C(X_1, \ldots, X_p, y)$ is given by the number of possible subsets of $E(G_x) \setminus E(G_y)$ that respect the partition P_x (always counting the empty set). We will now show that (3.2) equals $T_x[X_1, \ldots, X_p]$.

When we fix a partition $(V_1, \ldots, V_p)$ of $V(G_x)$ agreeing with P_x, this uniquely defines a partition P_y agreeing with P_x, namely $Y_i = V_i \cap B_y$ for all $i \in [p]$. Similarly, any partition $(V_1', \ldots, V_p')$ of $V(G_y)$ agreeing with P_y (for some P_y agreeing with P_x), uniquely defines a partition $(V_1, \ldots, V_p)$ of $V(G_x)$ that agrees with P_x (where we may need to add a vertex from $B_x \setminus B_y$ in the place specified by P_x).

If x is not introducing an edge, then $E(G_x) = E(G_y)$. Therefore, there is a one-to-one correspondence between tuples $(V_1, \ldots, V_p, X)$ counted for T_x and tuples $(P_y, V_1 \cap B_y, \ldots, V_p \cap B_y, X)$ counted for (3.2). Since in this situation $C = 1$, (3.2) indeed equals $T_x[X_1, \ldots, X_p]$.

If x is introducing the edge $uv \in E(G_x) \setminus E(G_y)$ and uv respects the partition P_x, then there is a two-to-one correspondence between tuples $(V_1, \ldots, V_p, X')$ and $(V_1, \ldots, V_p, X' \cup \{uv\})$ counted for T_x and tuples $(P_y, V_1 \cap B_y, \ldots, V_p \cap B_y, X')$ counted for (3.2), where $X' \subseteq E(G_y)$. In this case $C = 2$. If uv does not respect the partition, there is a one-to-one correspondence again and $C = 1$. This proves that (3.2) equals $T_x[X_1, \ldots, X_p]$ in all cases.

The number of partitions $(X_1, \ldots, X_p)$ of a bag B_x is at most $p^{|B_x|}$ (assign each vertex of B to an element of $[p]$). The computation of a single table entry is always done in time polynomial in n, and the size of the tree is also polynomial in n. Therefore, the running time of this algorithm is at most $p^{\mathrm{tw}} n^{O(1)}$. □

3.5 Conclusion

In this chapter we have given tight lower and upper bounds for counting the number of (list) q-colorings and connected spanning edge sets of graphs with a given cutwidth decomposition of small cutwidth. Our results specifically relate to list q-coloring and essentially distinct q-coloring, but they can easily be extended to normal q-coloring for certain cases. In particular, if $q < p$, we may apply Corollary 3.4.2, since in the setting of the corollary, the values differ by $q!$ which is nonzero modulo p. If the chromatic number $\chi(G) \geq p$, then the number of q-colorings is trivially $0 \mod p$, since the number of q-colorings is a multiple of $\chi(G)!$. This leaves us with the rather specific case of $\chi(G) < p \leq q$, for which the exact complexity remains

unresolved. In any case the lower bound also holds for regular (i.e. non-modular) q-coloring. Thus, we show that under the cutwidth parameterization, the (modular) counting variant of q-coloring is much harder than the decision version, as the latter can be solved in $2^{\mathrm{ctw}} n^{O(1)}$ time with a randomized algorithm [109].

Our results on the modular counting of colorings show that the modulus can influence the complexity in interesting ways, and that in some cases this effect can be directly explained by the rank of the compatibility matrix.

In the section on Connected Edgesets, we used a connection with the Tutte Polynomial. In Chapter 5 we will explore this connection further. We also showed how to use the rank based approach to speed up dynamic programming algorithms. We will see another application of this approach, this time to counting forests, with a more dramatic speedup in Chapter 4.

4 Counting Forests Parameterized by Cutwidth

A society grows great when old men plant trees whose shade they know they shall never sit in.

Greek proverb

4.1 Introduction

In this chapter we consider the problem of counting the number of forests in a graph, parameterized by path- and treewidth.

#Forests/tw
Input: A graph G, a tree decomposition $((B_x)_{x\in V(\mathbb{T})}, \mathbb{T})$ of G.
Parameter: tw
Question: How many edge subsets $A \subseteq E(G)$ are there, such that $(V(G), A)$ is a forest?

The study of counting forests dates back to Cayley's formula [144], stated in 1889, which counts forests on n vertices with exactly s trees, rooted at prelabled vertices. For counting forests as subgraphs of a given graph some research has been done into counting k-component forests for a given k, both in general [131] and parameterized by treewidth [136]. For counting forests with any number of components a $2^{o(n)}$ lower bound is known [28] (see also Theorem 4.1.2) but to our knowledge no (parameterized) algorithms have been given, beyond the obvious brute-force and dynamic programming algorithms.

Counting forests corresponds to the value $T(G;2,1)$ of the Tutte Polynomial. As is the case with the results in Chapter 3, this relation makes it an important case in determining the parameterized complexity of computing the Tutte Polynomial. We will discuss this further in Chapter 5.

We will focus on the algorithmic side of the story, since we may use the following (previously mentioned) theorem by Del et.al [28, Theorem 1] to find a lower bound in terms of cutwidth.

Theorem 4.1.1. *[Theorem 1 in [28]] If #ETH holds then there exists constants $\varepsilon, C > 0$ such that no $O(2^{\varepsilon n})$ time algorithm can compute the number of all forests in a given simple n-vertex with at most Cn edges.*

From the trivial bound of $\mathrm{ctw}(G) \leq |E(G)|$ we now find the following theorem as a corollary.

Theorem 4.1.2. *Let* ctw *be the cutwidth of a given n-vertex graph. Computing the Tutte polynomial along the curve H_0^y cannot be done in time $2^{o(\mathrm{ctw})}n^{O(1)}$, unless #ETH fails.*

We suspect that a $(4-\varepsilon)^{\mathrm{ctw}}n^{O(1)}$ lower bound for any $\varepsilon > 0$, based on SETH, also holds, but that it will take significant additional technical effort.

To complement this lower bound, we give algorithms to count the number of forests in a graph G in c^k time, where k is either the pathwidth (pw) or the treewidth (tw). For more detailed descriptions of these parameters, see Section 2.3.

Theorem 4.1.3.

a. *There exist an algorithm that, given a graph G with a path decomposition of width* $\mathrm{pw}(G)$, *computes the number of forests in the graph in time $O(4^{\mathrm{pw}}\,\mathrm{pw}^{5/2}\,n)$.*

b. *There exist an algorithm that, given a graph G with a tree decomposition of width* $\mathrm{tw}(G)$, *computes the number of forests in the graph in time $O(64^{\mathrm{tw}}\,\mathrm{tw}^{1/2}\,n)$.*

The algorithm uses a rank based approach, the runtime of which depends on the rank of the so called *forest compatibility matrix*. In Section 4.2 we introduce this matrix and examine its rank. Recall that it is not hard to show that $\mathrm{tw} \leq \mathrm{pw} \leq \mathrm{ctw}$ (see also Section 2.4) and thus any algorithmic result for treewidth with a matching lowerbound in terms of cutwidth, gives tight results for all three parameters.

In Chapter 5 we will see that the results from this chapter indicate an inherent asymmetry in the problem.

4.1.1 Notation

For sets $A, B \subseteq [n]$, we will write $A < B$ to indicate that $a < b$ for all $a \in A$ and $b \in B$.

We write $\pi \vdash S$ to indicate that π is a partition of S. We will typically use $S = [n]$ with the standard ordering on $[n]$. We write $\pi|_S$ for the partition given by restricting elements of π to the set $S \subseteq [n]$. Given two partitions $\pi_1 \vdash S$ and $\pi_2 \vdash S$, we say that π_1 is coarser than π_2, written $\pi_1 \geq \pi_2$, if every element of π_2 is a subset of an element of π_1. Given two partitions $\pi \vdash S$ and $\rho \vdash S'$ we define the join $\pi \sqcup \rho \vdash S \cup S'$ of the partitions as the finest partition of $S \cup S'$ such that both $(\pi \sqcup \rho)|_S \geq \pi$ and $(\pi \sqcup \rho)|_{S'} \geq \rho$. Intuitively put π and ρ together and merge any overlapping elements.

Given two permutations p and q, we will write $p \circ q$ for the composition of these two permutations. We will write (i, j) for the permutation that swaps elements i and j.

We will consider matrices indexed by partitions. Let M be such a matrix. We will write $M[\pi, \rho]$ for the entry of M in the row corresponding to π and the column corresponding to ρ. We will write $M[\pi]$ for the vector containing all elements in the row corresponding to π.

4.2 Rank Bound

In this section we give an upperbound on the rank of the so called *forest compatibility matrix* F_n. Before we can define the forest compatibility matrix, we first need the following definitions.

Definition 4.2.1. *We say that a boundaried graph $G = ([n] \cup V, E)$, with boundary $[n]$, is a* representative forest *for a partition $\pi \vdash [n]$, if for every $S \in \pi$ there is some connected component $C \subseteq V(G)$ of G such that $C \cap [n] = S$.*

Given two boundaried graphs G and H, both with boundary $[n]$, we define the glue *$G \oplus H$ of G and H as follows. First take the disjoint union of G and H. Then identify each $v \in [n]$ in G with its analogue in H.*

Throughout this chapter, we will assume that $V \cap [n] = \emptyset$. This definition shows how one can relate forests and partitions. Throughout this section we will mostly consider partitions as they capture all the information we need. The following definition elaborates on this by lifting the concept of cycles in a glue of two trees to a cycle induced by two partitions.

Definition 4.2.2. *Let $\pi, \rho \vdash [n]$ and let G_π and G_ρ be representative forests of π and ρ respectively. We say that π and ρ* induce a cycle *if $G_\pi \oplus G_\rho$ contains a cycle.*

It is not hard to see that it does not matter which representatives G_π and G_ρ we choose, since one only needs to know the connected components on $[n]$ to determine whether a cycle is present in the glue of the two forests. This means that this definition is indeed well-defined. For this same reason, in the following definition, we only need one row and column for each partition of the separator.

Definition 4.2.3. *We define the* forest compatibility matrix *F_S of a set S by*

$$F_S[\pi, \rho] := \begin{cases} 0 & \text{if } \pi \text{ and } \rho \text{ induce a cycle,} \\ 1 & \text{otherwise,} \end{cases}$$

for any $\pi, \rho \vdash S$. We will write $F_n := F_{[n]}$.

The main theorem of this section can now be stated as follows.

Theorem 4.2.4. *The rank of F_n is at most C_n, the n^{th} Catalan number. In particular* $\operatorname{rank}(F_n) = O(4^n n^{-3/2})$.

The *Catalan numbers* referenced in this theorem are given by

$$C_n = \frac{1}{n+1}\binom{2n}{n}.$$

The Catalan numbers count various things, like the number of ways to order n pairs of parenthesis in a valid way, the number of full binary trees with $n+1$ leaves or the number of non-crossing partitions on n elements.

This last interpretations is the one we will use in the proof of Theorem 4.2.4. Let us now define what it means for a partition to be crossing.

Definition 4.2.5. *We say that two sets $A, B \in \pi$ are* crossing *on an ordering $<$, if there are $a_1, a_2 \in A$ and $b_1, b_2 \in B$ such that either $a_1 < b_1 < a_2 < b_2$ or $b_1 < a_1 < b_2 < a_2$. If a partition contains two sets that are crossing on $<$, we say that it is* crossing on $<$. *We omit "on $<$" and simply call a partition of a pair of sets crossing, if the ordering $<$ is clear from context.*

Throughout this section it will sometimes be convenient to think of the ordering as a permutation.

The general idea behind the proof of Theorem 4.2.4 is to show that the non-crossing partitions form a basis of the matrix. We show this by proving that any partition can be uncrossed, i.e. its row in F_n can be written as a linear combination of rows, corresponding to non-crossing partitions.

4.2.1 Manipulating Partitions

For the proof of Theorem 4.2.4 we will need the following operations, which will allow us to manipulate partitions by contracting and expanding intervals and projecting down to subsets of the ground set.

Definition 4.2.6. *An* interval *is a subset $I \subseteq [n]$ of consecutive numbers, i.e. there is no $b \notin I$ such that $a_1 < b < a_2$ for some $a_1, a_2 \in I$. Given an interval I and a partition π of $[n]$, we define the* contraction *$\pi -_i I$ of π by I as the partition of the set $[n] -_i I := ([n] \cup \{i\}) \setminus I$ given by merging all sets that intersect I and replacing I by a single element i, i.e.*

$$\pi -_i I := \{S \in \pi : S \cap I = \emptyset\} \cup \left\{\left(\bigcup \{S \in \pi : S \cap I \neq \emptyset\} \cup \{i\}\right) \setminus I\right\}.$$

If we have an ordering on $[n]$, we place i in the same place in the ordering as I, that is for any $a \in [n] \setminus I$ and $b \in I$, we have $a < b$ if and only if $a < i$.

We define the blowup *$\pi +_i I$ of π by I as the partition of the set $[n] +_i I := ([n] \cup I) \setminus \{i\}$, given by adding all elements of I to the set that contains i and then removing i, i.e.*

$$\pi +_i I := \{S \in \pi : i \notin S\} \cup \{(S \setminus \{i\}) \cup I : i \in S\}.$$

Again we place I in the same place in the ordering as i.

We will sometimes abuse notation and refer to $[n] -_i I$ as simply $[n']$ for $n' = n - |I| + 1$.

We now turn our attention to a number of useful lemmas. The first lemma intuitively says that summation is preserved under contraction of intervals.

Lemma 4.2.7. *Let π be a partition of $[n]$ and let I be an interval such that $I \subseteq A \in \pi$ for some A. We set $n' = n - |I| + 1$. Suppose that for some set of partitions $\mathcal{R}$ of $[n']$, we have $F_{n'}[\pi -_i I] = \sum_{\rho \in \mathcal{R}} a_\rho F_{n'}[\rho]$. Then $F_n[\pi] = \sum_{\rho \in \mathcal{R}} a_\rho F_n[\rho +_i I]$.*

Proof. Let χ be some partition of $[n]$. Our aim will be to show that given some assumptions we have $F_n[\rho +_i I, \chi] = F_n[\rho, \chi -_i I]$ for any ρ. This would immediately imply that for such χ

$$F_n[\pi, \chi] = F_n[\pi -_i I, \chi -_i I] = \sum_{\rho \in R} a_\rho F_n[\rho, \chi -_i I] = \sum_{\rho \in R} a_\rho F_n[\rho +_i I, \chi],$$

which proves the lemma.

Note that if $|S' \cap I| \geq 2$ for some $S' \in \chi$, we have that $F_n[\pi, \chi] = F_n[\rho +_i I, \chi] = 0$. Thus we may assume that $|S' \cap I| \leq 1$. Let S be the set such that $i \in S \in \rho$. If there is some $S' \in \chi$ such that $|S' \cap S| \geq 2$, then by the previous assumption at most one of the elements in this intersection is in I and thus at least one is in $S \setminus I$. We find that $F_n[\rho +_i I, \chi] = F_n[\rho, \chi -_i I] = 0$ and thus we may assume that $|S' \cap S| \leq 1$.

We now show that for χ, such that $|S' \cap A| \leq 1$ for all $S' \in \chi$, we have $F_n[\rho +_i I, \chi] = F_n[\rho, \chi -_i I]$ for any ρ. First note that if ρ and $\chi +_i I$ induce a cycle, that does not involve I, then $\rho -_i I$ and χ also induce that same cycle and vice versa.

Now suppose that $\rho +_i I$ and χ induce only cycles involving I. We take such a cycle and note that by our second assumption there are no sets that intersect S at two elements. It follows that all cycles must be of length 4 or longer. We find that there are some S' and S'' in the cycle that intersects S, one of which, say S', intersects I. If $S'' \cap I = \emptyset$, the cycle is not affected by the contraction of I and thus we may assume that both S' and S'' intersect S at I. We find that after contraction S' and S'' are merged into one set S''' that intersects S only at i. Since the cycle was of length 4 or longer, we find that the cycle still has at least one other set from ρ in it and thus removing S, S' and S'' from the cycle and adding S''' gives a cycle induced by ρ and $\chi -_i I$.

In the reverse direction we assume that ρ and $\chi -_i I$ induce a cycle involving i, then it is clear to see that this cycle survives after blowing up i, using one of the sets in χ that intersect I. This proves the claim and thus the lemma. □

This next lemma intuitively says that if we project our partition to a subset of the ground set, then any decomposition of the resulting smaller partition gives the same decomposition of the larger partition.

Lemma 4.2.8. *Let π be a partition of $[n]$ and let $n' < n$. Suppose that for some set of partitions $\mathcal{R}$ of $[n']$, we have $F_{n'}[\pi|_{[n']}] = \sum_{\rho \in \mathcal{R}} a_\rho F_{n'}[\rho]$, then $F_n[\pi] = \sum_{\rho \in \mathcal{R}} a_\rho F_n[\rho \sqcup \pi|_{[n]\setminus[n']}]$.*

Proof. Let χ be some partition of $[n]$. If χ and $\pi|_{[n]\setminus[n']}$ induce a cycle, then the statement trivially holds. In the rest of the proof we will therefore assume that for any ρ, any cycle induced by χ and $\rho \sqcup \pi|_{[n]\setminus[n']}$ requires the use of ρ.

We first define an equivalence relation $\sim$ on $[n]$ by defining two elements to be equivalent if they are either in the same set of χ or in the same set of $\pi|_{[n]\setminus[n']}$. We then complete this to a full equivalence relation, by exhaustively adding pairs $a \sim c$ to the relation, for which we have $a \sim b$ and $b \sim c$ for some c. We now define the partition χ' of $[n']$ as the set of equivalence classes of $\sim$, restricted to $[n']$.

We claim that $F_n[\rho \sqcup \pi|_{[n]\setminus[n']}, \chi] = F_{n'}[\rho, \chi']$ for any ρ, which would immediately imply that

$$F_n[\pi, \chi] = F_{n'}[\pi|_{[n']}, \chi'] = \sum_{\rho \in \mathcal{R}} a_\rho F_{n'}[\rho, \chi'] = \sum_{\rho \in \mathcal{R}} a_\rho F_n[\rho \sqcup \pi|_{[n]\setminus[n']}, \chi]$$

which proves the lemma.

Suppose that $\rho \sqcup \pi|_{[n]\setminus[n']}$ and χ induce some cycle. Suppose the cycle uses some set in $\pi|_{[n]\setminus[n']}$, then this set lies on a subpath of the cycle that starts and ends at sets

in ρ (these may be the same set). Since all elements in this path are equivalent, this path must lie entirely inside of a set $S' \in \chi'$ and thus replacing any such path with S' results in a cyclic graph induced by ρ and χ'. Note that we may have multiple paths lying in the same S', however this is not an issue, since this will result in either multiple cycles intersecting at S' or S' intersecting some set in ρ at more than one element. Either way we still find a cycle.

Similarly, in the reverse direction we take a cycle induced by ρ and χ' and blow up any sets of χ' into a path in the corresponding connected component to find a cycle induced by $\rho \sqcup \pi|_{[n]\setminus[n']}$ and χ. □

The following two lemmas ensure that our operations do not introduce new crossings. The first of the two lemmas shows us that we can safely blow up an interval, as long as it is contained in a set of the partition.

Lemma 4.2.9. *Let $I \subseteq [n]$ be an interval of $[n]$. Let $\pi \vdash [n] -_i I$ be a non-crossing partition. Then $\pi +_i I$ is also non-crossing.*

Proof. Suppose that there are $C, D \in \pi +_i I$ that are crossing. W.l.o.g. there are $c_1, c_2 \in C$ and $d_1, d_2 \in D$ such that $c_1 < d_1 < c_2 < d_2$. Since π is non-crossing, this crossing does not exist in π and thus at least one of these elements is in I. By definition of a blowup, we must have either $I \subseteq C$ or $I \subseteq D$. Since I is an interval, it then follows that exactly one of the previously mentioned elements is in I. We still find a crossing in π by replacing this element by i. For example, if $d_1 \in I$, we find a crossing $c_1 < i < c_2 < d_2$ in π. This again contradicts the assumption that π is non-crossing. We conclude that $\pi +_i I$ is also non-crossing. □

This next lemma shows us that, if we have a partition with a single pair of crossing sets, then replacing these sets with some non-crossing partition cannot introduce new crossings.

Lemma 4.2.10. *Let $\pi \vdash [n]$ be a partition such that only $A, B \in \pi$ cross each other and all other pairs of sets in π are non-crossing. Then for any non-crossing partition ρ of $A \cup B$ we have that $\rho \cup \pi|_{[n]\setminus(A\cup B)}$ is non-crossing.*

Proof. Suppose there are sets $C, D \in \rho \sqcup \pi|_{[n]\setminus(A\cup B)}$ that cross each other. By assumption $\pi|_{[n]\setminus(A\cup B)}$ is non-crossing and thus w.l.o.g. $C \in \rho$. Also note that since ρ is non-crossing, this implies that $D \in \pi|_{[n]\setminus(A\cup B)}$.

Let I be the interval spanned by $A \cup B$, then since D crosses $C \subseteq I$, we find that $D \cap I \neq \emptyset$ and D is not an interval itself. We claim that this implies that, in π, D crosses either A or B. This would contradict the assumption that the only crossings in π are between A and B, which would then imply the lemma.

Note that $D \cap I$ cannot include either the rightmost or the leftmost element of the interval, since these must be elements of $A \cup B$. Therefore if neither A nor B crosses D, we must have that w.l.o.g. $A < D \cap I < B$. This is not possible, since A and B must cross at least once. □

4.2.2 Proof of the Rank Bound

With Lemmas 4.2.7 to 4.2.10 in hand, we are now ready to describe the main uncrossing operation.

Lemma 4.2.11. *Let $\pi \vdash [n]$ be a partition non-crossing on an ordering p of $[n]$. In time $O(n)$ we can find constants c_ρ, such that $F_n[\pi] = \sum_{\rho \in \mathcal{N}} c_\rho F_n[\rho]$, where $\mathcal{N}$ is the set of partitions that are non crossing on $p \circ (i, i+1)$.*

Proof. Throughout the proof, we will consider the partition π on the ordering $p \circ (i, i+1)$. We first note that since π is non-crossing on p, any crossing of π must involve both i and $i+1$. Let $i \in A \in \pi$ and $i+1 \in B \in \pi$. If $A = B$, then π is non-crossing and thus we may assume that $A \neq B$.

Since π was non-crossing on p, A and B were non-crossing on p. After swapping i and $i+1$, we find that when viewed as a partition of $A \cup B$, $\pi|_{A\cup B}$ consists of either 4 or 5 intervals which alternate between A and B. Define π' as the partition given by contracting these intervals. We find that π' is a partition on n' elements, where either $n' = 4$ or $n' = 5$, with intervals of size 1 (see Figure 4.1).

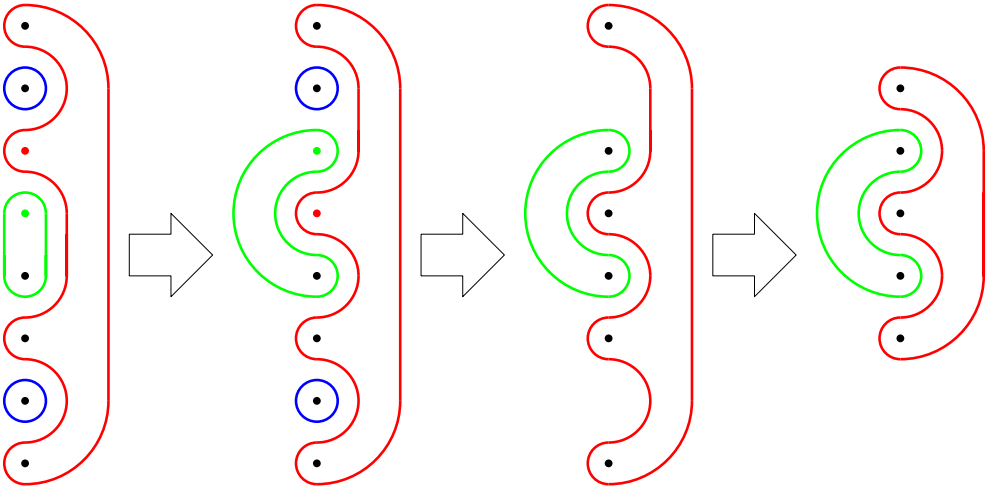


Figure 4.1: From left to right, these are examples of π before the swap, π after the swap, $\pi|_{A\cup B}$ and π'.

We can explicitly construct the forest compatibility matrices for $n' \in \{4, 5\}$ and check that the non-crossing partitions span the rowspace. With the original publication of this work [126] we provided a MATLAB script that verifies this[1], by decomposing each row as a weighted sum of the rows corresponding to non-crossing

[1] https://archive.softwareheritage.org/browse/directory/2e6936582c19e5fd2f127b3d1e601ecb9a1136f1/?origin_url=https://github.com/isja-m/ForestRank4-5&revision=92e7701e89464c3fc4a2eccfe232600e9fa91605&snapshot=9399fd1d60243aa5a851ec5a7c7ccaa25f169623

partitions on $p \circ (i, i+1)$. Thus we can write

$$F_{n'}[\pi'] = \sum_{\rho \in \mathcal{R}} c_\rho F_{n'}[\rho],$$

where $\mathcal{R}$ is the set of non-crossing partitions of $[n']$. By applying Lemma 4.2.7 n' times, we find that

$$F_{A \cup B}[\pi|_{A \cup B}] = \sum_{\rho \in \mathcal{R}} c_\rho F_{A \cup B}[\rho +_{i_1} I_1 + \cdots +_{i_{n'}} I_{n'}].$$

By Lemma 4.2.9, each $\rho +_{i_1} I_1 + \cdots +_{i_{n'}} I_{n'}$ is still non-crossing. By Lemma 4.2.8, we find

$$F_n[\pi] = \sum_{\rho \in \mathcal{R}} c_\rho F_{A \cup B}[(\rho +_{i_1} I_1 + \cdots +_{i_{n'}} I_{n'}) \cup \pi|_{[n] \setminus (A \cup B)}].$$

By Lemma 4.2.10, each $(\rho +_{i_1} I_1 + \cdots +_{i_{n'}} I_{n'}) \cup \pi|_{[n] \setminus (A \cup B)}$ is still non-crossing. We conclude that $F_n[\pi]$ can be written as a linear combination of rows corresponding to non-crossing partitions.

Note that we can construct π' in $O(n)$ time. We then find the c_ρ in $O(1)$ time and reconstruct the $(\rho +_{i_1} I_1 + \cdots +_{i_{n'}} I_{n'}) \cup \pi|_{[n] \setminus (A \cup B)}$ in $O(n)$ time. □

By repeatedly applying Lemma 4.2.11, we can prove the following theorem.

Theorem 4.2.12. *For a given ordering p, the rows corresponding to partitions non-crossing on p span a row basis of the forest compatibility matrix F_n.*

Proof. Let π be a partition of $[n]$ such that we can turn it into a non-crossing partition by swapping two consecutive elements i and $i+1$ in the order of $[n]$. By Lemma 4.2.11 we can write the row $F_n[\pi]$ corresponding to π as a linear combination of rows corresponding to non-crossing partitions of $[n]$. This shows that, for B_p the set of rows corresponding to non-crossing partitions on p, we have $B_{p \circ (i,i+1)} \subseteq \mathrm{span}(B_p)$. Since every partition is non-crossing for some permutation and every permutation can be decomposed into 2-cycles on consecutive elements, by induction we find that every row can be written as a linear combination of rows corresponding to non-crossing partitions on p. □

From this we immediately find a proof for Theorem 4.2.4.

Proof of Theorem 4.2.4. By Theorem 4.2.12 the non-crossing partitions form a basis of F_n. Since there are C_n such partitions we find $\mathrm{rank}(F_n) \leq C_n$. □

4.3 The Algorithm

We will now describe the algorithm for counting forests. We first define the dynamic programming table and the notion of representation. We then handle each type of node in the tree/path decomposition separately and summarize at the end.

Definition 4.3.1. *Let G be a graph and let $(\mathbb{T}, (B_x)_{x\in V(\mathbb{T})})$ be a tree/path decomposition of G. Recall that G_x is defined as the graph induced by the union of all bags, whose nodes are descendants of x in $\mathbb{T}$. For each node x and partition $\pi \vdash B_x$, we define the entry $\tau_x[\pi]$ of the dynamic programming table τ by*

$$\tau_x[\pi] := |\{X \subseteq E(G_x) : (V, X) \text{ is acyclic }, \\ \forall u, v \in B_x \text{ there is a path in } (V, X) \text{ from } u \text{ to } v \text{ iff } \exists S \in \pi \text{ s.t. } u, v \in S\}|.$$

In other words, the table entry $\tau_x[\pi]$ counts the number of forests in G_x whose connected components agree with π. In the rest of this section, we will refer to the set $\mathrm{supp}(\tau_x)$ of indices π with nonzero entries $\tau_x[\pi]$ of the dynamic programming table for a given x as the support of τ_x. Our aim will be to ensure that the support of our rows remains contained in the entries corresponding to non-crossing partitions for some ordering on the bag B_x. This is captured in the following definition.

Definition 4.3.2. *We say a vector a, indexed by partitions, is* reduced *on an ordering p, if $a_\pi = 0$ for any partition π that is crossing on p. If p is clear from context, we will simply say that a is* reduced.

In order to ensure that we do not lose any relevant information we will reduce our table, while retaining an F_{B_x}-representation. For the readers convenience, we restate the definition of representation here

Definition 2.8.4. *Given two vectors T and T', we say T' is an M-*representative *of T if*

$$\sum_{x\in \mathrm{col}(X)} M[x, y]T[x] = \sum_{x\in \mathrm{col}(X)} M[x, y]T'[x] \text{ for all } y \in \mathrm{col}(Y),$$

i.e. $TM = T'M$.

We now describe how the algorithm behaves on the various types of nodes. In each case we apply one step of a naive dynamic programming algorithm and then reduce the table if it becomes too big. For ease of notation we will write $\pi \sim \rho$ if the partitions $\pi, \rho \vdash [n]$ are compatible, i.e. they do not induce a cycle.

We start by setting $\tau'_x[\emptyset] := \tau_x[\emptyset] = 1$ for all leaf nodes x. We trivially find that τ'_x is reduced and F_0-represents τ_x.

4.3.1 Vertex-Introduce Node

Lemma 4.3.3. *Let x be a vertex-introduce node with a child node y. Suppose that τ'_y is reduced and F_{B_y}-represents τ_y. We can compute a row τ'_x that is reduced and F_{B_x}-represents τ_x in time $O(\mathrm{rank}(F_{B_x}))$.*

Proof. If x is a vertex-introduce node, introducing v. We set

$$\tau'_x[\pi \cup \{\{v\}\}] := \tau'_y[\pi]$$

and

$$\tau'_x[\pi] := 0$$

for any π in which v does not appear as a singleton. Clearly for any non-crossing partition π, we have that $\pi \cup \{v\}$ is still non-crossing and thus τ'_x is reduced.

Note that by definition, we need to show that $F_{B_x}\tau'_x = F_{B_x}\tau_x$. In the following derivation we show that this equality holds at the entry corresponding to any arbitrary partition $\rho \vdash B_x$.

$$\begin{aligned}
\sum_{\pi\sim\rho} \tau_x[\pi] &= \sum_{\substack{\pi\sim\rho \\ \{v\}\in\pi}} \tau_y[\pi \setminus \{\{v\}\}] \\
&= \sum_{\pi'\sim\rho|_{B_y}} \tau_y[\pi'] \\
&= \sum_{\pi'\sim\rho|_{B_y}} \tau'_y[\pi'] \\
&= \sum_{\substack{\pi\sim\rho \\ \{v\}\in\pi}} \tau'_y[\pi \setminus \{\{v\}\}] \\
&= \sum_{\pi\sim\rho} \tau'_x[\pi].
\end{aligned}$$

□

4.3.2 Vertex-Forget Node

Lemma 4.3.4. *Let x be a vertex-forget node with a child node y. Suppose that τ'_y is reduced and F_{B_y}-represents τ_y. We can compute a row τ'_x that is reduced and F_{B_x}-represents τ_x in time $O(\mathrm{rank}(F_{B_x}))$.*

Proof. Let x be a vertex-forget node, forgetting v. We set

$$\tau'_x[\pi] := \sum_{\pi'|_{B_x}=\pi} \tau'_y[\pi'].$$

Clearly for any non-crossing partition π', we have that $\pi'|_{B_x}$ is still non-crossing and thus τ'_x is reduced.

Again we now show that $F_{B_x}\tau'_x = F_{B_x}\tau_x$, by focussing on the entry of the vector at coordinate ρ.

$$\sum_{\pi\sim\rho} \tau_x[\pi] = \sum_{\pi\sim\rho} \sum_{\pi'|_{B_x}=\pi} \tau_y[\pi'].$$

Note that π' projects down to a partition that is compatible with ρ if and only if $\pi' \sim (\rho \cup \{\{v\}\})$. The above then equals

$$\begin{aligned}\sum_{\pi \sim \rho} \sum_{\pi'|_{B_x} = \pi} \tau_y[\pi'] &= \sum_{\pi' \sim (\rho \cup \{\{v\}\})} \tau_y[\pi'] \\ &= \sum_{\pi' \sim (\rho \cup \{\{v\}\})} \tau'_y[\pi'] \\ &= \sum_{\pi \sim \rho} \sum_{\pi'|_{B_x} = \pi} \tau'_y[\pi'] \\ &= \sum_{\pi \sim \rho} \tau'_x[\pi].\end{aligned}$$

□

4.3.3 Edge-Introduce Node

Lemma 4.3.5. *Let x be an edge-introduce node with a child node y. Suppose that τ'_y is reduced on p and F_{B_y}-represents τ_y. We can compute a row τ'_x that is reduced on some ordering p' and F_{B_x}-represents τ_x in time $O(\mathrm{rank}(F_{B_x})|B_x|^2)$.*

Before we prove this lemma, we introduce the following technical lemma. This lemma will be useful to show that representation is preserved after applying the dynamic programming step.

Lemma 4.3.6. *Let $\pi, \chi, \rho \vdash [n]$ be partitions such that $\pi \sim \chi$ and $\rho \sim \chi$. We have that $\pi \sqcup \chi \sim \rho$ if and only if $\pi \sim \rho \sqcup \chi$.*

Proof. Recall the definition of a representative forest 4.2.2. Let G_π, G_χ and G_ρ be representative forests of π, χ and ρ respectively. Suppose that $\pi \sqcup \chi \sim \rho$. Since $\pi \sim \chi$, $G_\pi \oplus G_\chi$ is a forest. Moreover it is a representative forest of $\pi \sqcup \chi$. By the same reasoning we find that $G_\rho \oplus G_\chi$ is a representative forest of $\rho \sqcup \chi$. Since $\pi \sqcup \chi \sim \rho$, we find that $(G_\pi \oplus G_\chi) \oplus G_\rho = G_\pi \oplus (G_\chi \oplus G_\rho)$ is a forest and thus $\pi \sim \rho \sqcup \chi$.

The reverse direction follows from a similar argument. □

Proof of Lemma 4.3.5. Let x be an edge-introduce node for edge uv. It is not hard to see that if u and v are adjacent in the vertex ordering of B_x, then $\pi \sqcup \pi_{uv}$ is non-crossing if and only if π is non-crossing. We will aim to find a F_{B_y}-representative τ''_y of τ'_y, that is reduced on an ordering p' in which u and v are adjacent.

By applying Lemma 4.2.11 to each entry of τ'_y we can find an F_{B_y}-representative of τ'_y, that is reduced on $p \circ (i, i+1)$, that is we can swap two consecutive elements. Using at most $|B_y|$ of these swaps we can ensure that u and v are adjacent. Each such swap costs $|B_y|$ time per non-zero entry of the current vector. Since any reduced vector has at most $\mathrm{rank}(F_{B_y})$ non-zero entries, we find a runtime of $O(\mathrm{rank}(F_{B_y})|B_y|^2)$.

We can now compute the desired τ'_x. We first define

$$\pi_{uv} := \{\{w\} : w \in B_y \setminus \{u, v\}\} \cup \{\{u, v\}\}$$

and set

$$\tau'_x[\pi] := \tau''_y[\pi] + \sum_{\pi' \sqcup \pi_{uv} = \pi} F_n[\pi', \pi_{uv}]\tau''_y[\pi'],$$

which is still reduced on p', since u and v are adjacent. Finally we again show that $F_{B_x}\tau'_x = F_{B_x}\tau_x$.

$$\begin{aligned}\sum_{\pi\sim\rho}\tau_y[\pi] &= \sum_{\pi\sim\rho}\left(\tau_x[\pi] + \sum_{\pi'\sqcup\pi_{uv}=\pi} F_n[\pi',\pi_{uv}]\tau_x[\pi']\right)\\ &= \sum_{\pi\sim\rho}(\tau_x[\pi]) + \sum_{\pi\sim\rho}\left(\sum_{\pi'\sqcup\pi_{uv}=\pi} F_n[\pi',\pi_{uv}]\tau_x[\pi']\right).\end{aligned}$$

Since τ'_x F_n-represents τ_x this equals

$$\begin{aligned}\sum_{\pi\sim\rho}\tau_y[\pi] &= \sum_{\pi\sim\rho}(\tau'_x[\pi]) + \sum_{\pi\sim\rho}\left(\sum_{\pi'\sqcup\pi_{uv}=\pi} F_n[\pi',\pi_{uv}]\tau_x[\pi']\right)\\ &= \sum_{\pi\sim\rho}(\tau'_x[\pi]) + \sum_{\pi'\sqcup\pi_{uv}\sim\rho} F_n[\pi',\pi_{uv}]\tau_x[\pi'].\end{aligned}$$

If $\rho \sim \pi_{uv}$, by Lemma 4.3.6 this equals

$$\begin{aligned}\sum_{\pi\sim\rho}\tau_y[\pi] &= \sum_{\pi\sim\rho}(\tau'_x[\pi]) + \sum_{\pi'\sim\rho\sqcup\pi_{uv}}(\tau_x[\pi'])\\ &= \sum_{\pi\sim\rho}(\tau'_x[\pi]) + \sum_{\pi'\sim\rho\sqcup\pi_{uv}}(\tau'_x[\pi'])\\ &= \sum_{\pi\sim\rho}(\tau'_x[\pi]) + \sum_{\pi\sim\rho}\left(\sum_{\pi'\sqcup\pi_{uv}=\pi} F_n[\pi',\pi_{uv}]\tau'_x[\pi']\right)\\ &= \sum_{\pi\sim\rho}\left(\tau'_x[\pi] + \sum_{\pi'\sqcup\pi_{uv}=\pi} F_n[\pi',\pi_{uv}]\tau'_x[\pi']\right)\\ &= \sum_{\pi\sim\rho}\tau'_y[\pi].\end{aligned}$$

Otherwise we find $\pi' \sqcup \pi_{uv} \nsim \rho$ for any π' and thus

$$\begin{aligned}\sum_{\pi\sim\rho}\tau_y[\pi] &= \sum_{\pi\sim\rho}(\tau'_x[\pi])\\ &= \sum_{\pi\sim\rho}(\tau'_x[\pi]) + \sum_{\pi\sim\rho}\left(\sum_{\pi'\sqcup\pi_{uv}=\pi} F_n[\pi',\pi_{uv}]\tau'_x[\pi']\right)\\ &= \sum_{\pi\sim\rho}\tau'_y[\pi].\end{aligned}$$

□

4.3.4 Join Node

Lemma 4.3.7. *Let x be a join node with child nodes y_1 and y_2. Suppose that τ'_{y_i} is reduced and $F_{B_{y_i}}$-represents τ_{y_i} for $i = 1, 2$. We can compute a row τ'_x that is reduced and F_{B_x}-represents τ_x in time $O(\mathrm{rank}(F_{B_x})^3 |B_x|^3)$.*

Proof. We begin by setting

$$\tau''_x[\pi] := \sum_{\pi_1 \sqcup \pi_2 = \pi} F_n[\pi_1, \pi_2] \tau'_{y_1}[\pi_1] \tau'_{y_2}[\pi_2].$$

We will first prove that τ''_x F_{B_x}-represents τ_x and then reduce it afterwards.

$$\sum_{\pi \sim \rho} \tau_x[\pi] = \sum_{\pi \sim \rho} \sum_{\pi_1 \sqcup \pi_2 = \pi} F_n[\pi_1, \pi_2] \tau_{y_1}[\pi_1] \tau_{y_2}[\pi_2].$$

By changing the order in which we pick π, π_1 and π_2 we can rewrite this expression as

$$\begin{aligned}\sum_{\pi \sim \rho} \tau_x[\pi] &= \sum_{\pi \sim \rho} \sum_{\pi_1 \leq \pi} \tau_{y_1}[\pi_1] \sum_{\pi_1 \sqcup \pi_2 = \pi} F_n[\pi_1, \pi_2] \tau_{y_2}[\pi_2] \\ &= \sum_{\pi_1} \tau_{y_1}[\pi_1] \sum_{\substack{\pi \sim \rho \\ \pi_1 \leq \pi}} \sum_{\pi_1 \sqcup \pi_2 = \pi} F_n[\pi_1, \pi_2] \tau_{y_2}[\pi_2].\end{aligned}$$

We can now merge the two inner sums into one, which results in

$$\sum_{\pi \sim \rho} \tau_x[\pi] = \sum_{\pi_1} \tau_{y_1}[\pi_1] \sum_{\pi_1 \sqcup \pi_2 \sim \rho} F_n[\pi_1, \pi_2] \tau_{y_2}[\pi_2].$$

If $\rho \sim \pi_1$, by Lemma 4.3.6 we find

$$\begin{aligned}\sum_{\pi_1 \sqcup \pi_2 \sim \rho} F_n[\pi_1, \pi_2] \tau_{y_2}[\pi_2] &= \sum_{\pi_2 \sim \rho \sqcup \pi_1} \tau_{y_2}[\pi_2] \\ &= \sum_{\pi_2 \sim \rho \sqcup \pi_1} \tau''_{y_2}[\pi_2] \\ &= \sum_{\pi_1 \sqcup \pi_2 \sim \rho} F_n[\pi_1, \pi_2] \tau''_{y_2}[\pi_2].\end{aligned}$$

Otherwise we find

$$\sum_{\pi_1 \sqcup \pi_2 \sim \rho} F_n[\pi_1, \pi_2] \tau_{y_2}[\pi_2] = 0 = \sum_{\pi_1 \sqcup \pi_2 \sim \rho} F_n[\pi_1, \pi_2] \tau''_{y_2}[\pi_2].$$

Either way we find

$$\sum_{\pi \sim \rho} \tau_x[\pi] = \sum_{\pi_1} \tau_{y_1}[\pi_1] \sum_{\pi_1 \sqcup \pi_2 \sim \rho} F_n[\pi_1, \pi_2] \tau''_{y_2}[\pi_2].$$

By applying the same operations as before, but in reverse, we find

$$\begin{aligned}\sum_{\pi\sim\rho}\tau_x[\pi] &= \sum_{\pi_1}\tau_{y_1}[\pi_1]\sum_{\substack{\pi\sim\rho\\ \pi_1\leq\pi}}\sum_{\pi_1\sqcup\pi_2=\pi}F_n[\pi_1,\pi_2]\tau''_{y_2}[\pi_2]\\ &= \sum_{\pi\sim\rho}\sum_{\pi_1\leq\pi}\tau_{y_1}[\pi_1]\sum_{\pi_1\sqcup\pi_2=\pi}F_n[\pi_1,\pi_2]\tau''_{y_2}[\pi_2]\\ &= \sum_{\pi\sim\rho}\sum_{\pi_1\sqcup\pi_2=\pi}F_n[\pi_1,\pi_2]\tau_{y_1}[\pi_1]\tau''_{y_2}[\pi_2].\end{aligned}$$

By applying the same reasoning to τ_{y_1}, we find

$$\begin{aligned}\sum_{\pi\sim\rho}\tau_x[\pi] &= \sum_{\pi\sim\rho}\sum_{\pi_1\sqcup\pi_2=\pi}F_n[\pi_1,\pi_2]\tau''_{y_1}[\pi_1]\tau''_{y_2}[\pi_2]\\ &= \sum_{\pi\sim\rho}\tau''_x[\pi].\end{aligned}$$

We now describe how we reduce τ''_x to find τ'_x. For each partition π such that $\tau''_x[\pi]\neq 0$, we first determine an ordering p' for which π is non-crossing. Note that we can transform p' into p by performing at most $|B_x|^2$ swaps, where we swap the order of two consecutive elements. Again by applying Lemma 4.2.11 to each entry of τ''_x we can find an F_{B_x}-representative of[2] $e_\pi\cdot\tau''_x[\pi]$, that is reduced on $p'\circ(i,i+1)$.

We perform at most $O(|B_x|^2)$ such swaps each costing at most $O(\mathrm{rank}(F_{B_x})|B_x|)$, since the support of the vector cannot exceed $\mathrm{rank}(F_{B_x})$ as a result of these swaps. After we have done this for every such π, we sum the resulting vectors to find an F_{B_x}-representative τ'_x of τ''_x. Finding the vectors takes $O(\mathrm{rank}(F_{B_x})|B_x|^3)$ per non-zero entry of τ''_x and thus takes $O(\mathrm{supp}(\tau''_x)\mathrm{rank}(F_{B_x})|B_x|^3)$ time in total. Summing all the vectors takes at most $O(\mathrm{rank}(F_{B_x})\mathrm{supp}(\tau'_x))$ time. Since we assumed τ'_{y_1} and τ'_{y_2} to be reduced, we find that $\mathrm{supp}(\tau''_x)\leq\mathrm{rank}(F_{B_x})^2$ and thus the algorithm runs in time $O(\mathrm{rank}(F_{B_x})^3|B_x|^3)$. □

4.3.5 Algorithmic Results

The previous lemmas together prove Theorem 4.1.3.

Theorem 4.1.3.a. *(restated) There exist an algorithm that, given a graph G with a path decomposition of width* $\mathrm{pw}(G)$*, computes the number of forests in the graph in time* $O(4^{\mathrm{pw}}\,\mathrm{pw}^{5/2}\,n)$.

Proof. W.l.o.g. we assume we are given a nice path decomposition, where the first and last nodes correspond to empty bags. As mentioned in the section of leaf nodes we can directly compute a representative solution on the first node. By applying Lemma's 4.3.3, 4.3.4 and 4.3.5, we can compute representative solutions for all

[2] We use the standard notation e_π to indicate the vector with a 1 at coordinate π and 0's elsewhere.

nodes. The row corresponding to the last node will contain a single entry, which gives the number of forests in the graph.

By Lemma's 4.3.3, 4.3.4 and 4.3.5, each step in the dynamic program takes at most $O(\text{rank}(F_{\text{pw}})\,\text{pw}^2) = O(4^{\text{pw}}\,\text{pw}^{1/2})$ time. Since there are at most $O(n\,\text{pw}^2)$ edges in the graph and thus $O(n\,\text{pw}^2)$ nodes in the path decomposition, we find a total running time of $O(4^{\text{pw}}\,\text{pw}^{5/2}\,n)$. □

Theorem 4.1.3.b. *(restated) There exist an algorithm that, given a graph G with a tree decomposition of width* $\text{tw}(G)$*, computes the number of forests in the graph in time* $O(64^{\text{tw}}\,\text{tw}^{1/2}\,n)$.

Proof. W.l.o.g. we assume we are given a nice tree decomposition, where the leaf nodes correspond to empty bags. We will root this decomposition at one of the leaf nodes r. As mentioned in the section of leaf nodes we can directly compute a representative solution on the first node. By applying Lemma's 4.3.3, 4.3.4, 4.3.5 and 4.3.7, we can compute representative solutions for all nodes. τ_r will contain a single entry, which gives the number of forests in the graph.

By Lemma's 4.3.3, 4.3.4, 4.3.5 and 4.3.7, each step in the dynamic program takes at most $O(\text{rank}(F_{\text{tw}})^3\,\text{tw}^3) = O(64^{\text{tw}}\,\text{tw}^{-3/2})$ time. Since there are $O(n\,\text{tw}^2)$ edges in the graph and thus $O(n\,\text{tw}^2)$ nodes in the tree decomposition, we find a total running time of $O(64^{\text{tw}}\,\text{tw}^{1/2}\,n)$. □

4.4 Conclusion

In this chapter we have given an algorithm for counting forests in a graph, that matches existing ETH bounds. To achieve this we have deployed a novel application of the rank-based approach. The matrix involved in this application is indexed by partitions of a set $[n]$ and has the interesting property that the non-crossing partitions span the row space of the matrix. This implies that the rank of the matrix is upper bounded by the Catalan numbers. Strictly speaking we have not shown that the non-crossing partitions form a basis, since we have not shown that they index linearly independent rows in the matrix. However, explicit computation of the rank for $n = 1,\ldots,8$ seems to suggest that the Catalan numbers do in fact give the rank of the forest compatibility matrix.

We believe that our rank upper bound should have more applications for counting forests with different properties. For example, it seems plausible that it can be used to count all Feedback Vertex Sets in time $2^{O(\text{tw})}n^{O(1)}$ or the number of spanning trees with k components in time $2^{O(\text{tw})}n^{O(1)}$. The latter result would improve over a result by Peng and Fei Wan [136] that show how to count the number of spanning forests with k components (or equivalently, $n-k-1$ edges) in $\text{tw}^{O(\text{tw})}\,n^{O(1)}$ time.

5 Computing the Tutte Polynomial Parameterized by Various Parameters

General Dwight D. Eisenhower himself described Tutte's work as one of the greatest intellectual feats of the Second World War.

Nancy Harper, in the University of Waterloo's magazine

5.1 Introduction

In this chapter we study the parameterized complexity of computing the Tutte Polynomial. We will use the results from Chapters 3 and 4 to determine the complexity of some of the cases and will handle the remaining cases in this chapter. We now state the problem for cutwidth and note that the definition is analogues for path- and treewidth.

$\textsc{Tutte}^{(x,y)}$/ctw **Input:** A graph G, a width ctw cut decomposition $v_1,\ldots,v_n$ of G. **Parameter:** ctw **Question:** What is the value of $T(G;x,y)$?

Due to its generality the Tutte polynomial is of great interest to a variety of fields, including knot theory, statistical physics and combinatorics. For a number of these fields it is important to understand how difficult it is to compute the Tutte polynomial. A series of papers, culminating in the work by Jaeger, Vertigan, and Welsh [106] has given a complete dichotomy showing that the problem of evaluating the Tutte polynomial is #P-hard on all points except on the following *special points* on which it is known to be computable in polynomial time:

$$(1,1),(-1,-1),(0,-1),(-1,0),(i,-i),(-i,i),(j,j^2),(j^2,j),\ H_1, \tag{5.1}$$

where $j = e^{2\pi i/3}$ and $i = \sqrt{-1}$, and H_α denotes the hyperbola $\{(x,y) : (x-1)(y-1) = \alpha\}$. These hyperbolic curves turn out to be of great importance to understanding the complexity of the Tutte Polynomial, as the problem is generally equally hard on all points of the same curve, except for the *special points* listed in (5.1).

Further refinements of the result by Jaeger et al. [106] have since been made: Among others, a more fine-grained examination of the complexity was done by

Brand et al. [28] (building on earlier work by Dell et. al. [61]): they showed that for almost all points the Tutte polynomial cannot be evaluated in $2^{o(n)}$ time on n-vertex graphs, assuming (a weaker counting version of) the Exponential Time Hypothesis. This is tight because, on the positive side, Björklund et al. [14] showed that the Tutte polynomial can be evaluated on any point in $2^n n^{O(1)}$ time.

We are interested in parameterized versions of such complexity classifications. On the subject of evaluating the Tutte polynomial parameterized by width measures, research has already been done over twenty years ago: Noble [134] has given a polynomial time algorithm for evaluation the Tutte Polynomial on bounded treewidth graphs. Noble mostly focused on the dependence on the number of vertices and edges, and showed each point of the Tutte polynomial can be evaluated in linear time, assuming the treewidth of the graph is constant. See also an independently discovered (but slower) algorithm by Andrzejak [4]. In Section 5.5 we will give an algorithm with a better dependency on the treewidth, to match some of our lower bounds. This improvement over Noble comes at the expense of a worse dependency on the input size n.

In this chapter, we determine the fine-grained complexity for each integer point (x, y) of the problem of evaluating the Tutte polynomial (x, y), under parameterization by treewidth, pathwidth and cutwidth.

As was done in previous works, we base our lower bounds on the Exponential Time Hypothesis (ETH) and the Strong Exponential Time Hypothesis (SETH) formulated by Impagliazzo and Paturi [105]. For a given width parameter k, the former will be used to exclude run times of the form $k^{o(k)}n^{O(1)}$ and $c^{o(k)}$ for some constant c, while the latter will be used to exclude run times of the form $(c-\varepsilon)^k n^{O(1)}$ for any $\varepsilon > 0$.

Specifically we consider the *treewidth*, *pathwidth* and *cutwidth* of the graph. The first two, in some sense, measure how close the graph is to looking like a tree or path respectively. The cutwidth measures how many edges are layered on top of each other when the vertices are placed in any linear order. For more precise definitions of these parameters, see Section 2.3 of the preliminaries.

Width measures in particular are interesting because instances where such structural parameters are small come up a lot in practice. For example, the curve H_2 corresponds to the partition function of the Ising model, which is widely studied in statistical physics, on graphs with particular topology such as lattice graphs or open/closed Cayley trees ([121]). In all such graphs with n vertices, even the cutwidth (the largest parameter we study) is at most $O(\sqrt{n})$.

5.1.1 Complexity Classification

Our classification handles points (x, y) differently based on whether $(x-1)(y-1)$ is negative, zero or positive, and reads as follows:

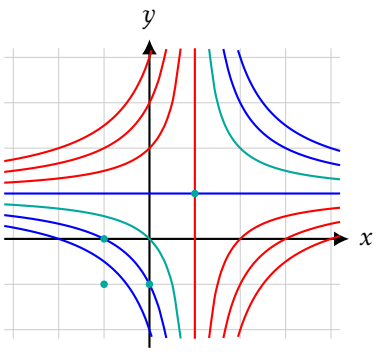


Figure 5.1: The red points have time complexity of the form $k^{O(k)}n^{O(1)}$, the blue points have time complexity of the form $c^k n^{O(1)}$ for some constant c and the green points have polynomial time complexity.

Theorem 5.1.1. *Let G be a graph with given tree, path and cut decompositions of width* tw, pw *and* ctw *respectively. Let $(x,y) \in \mathbb{Z}^2$ be a non-special point, then up to some polynomial factor in $|V(G)|$, the following holds.*

a. *If $(x-1)(y-1) < 0$ or $x = 1$, then $T(G;x,y)$ can be computed in time $\mathrm{tw}^{O(\mathrm{tw})}\, n^{O(1)}$ and cannot be computed in time $\mathrm{ctw}^{o(\mathrm{ctw})}\, n^{O(1)}$ under ETH.*

b. *If $y = 1$, then $T(G;x,y)$ can be computed in time $4^{\mathrm{pw}} n^{O(1)}$ or $64^{\mathrm{tw}} n^{O(1)}$ and cannot be computed in time $2^{o(\mathrm{ctw})} n^{O(1)}$ under #ETH.*

c. *If $(x-1)(y-1) = q > 1$, then $T(G;x,y)$ can be computed in time $q^{\mathrm{tw}} n^{O(1)}$. Furthermore,*

1. *if $x \neq 0$, then $T(G;x,y)$ cannot be computed in time $(q-\varepsilon)^{\mathrm{ctw}} n^{O(1)}$ under SETH.*
2. *if $x = 0$, then $T(G;x,y)$ cannot be computed in time $(q-\varepsilon)^{\mathrm{pw}} n^{O(1)}$ and $(q-\varepsilon)^{\mathrm{ctw}/2} n^{O(1)}$ under SETH.*

This is a fine-grained classification for evaluating the Tutte polynomial at any given integer point, simultaneously for all the parameters treewidth, pathwidth and cutwidth. This is because if a graph has cutwidth ctw, pathwidth pw and treewidth tw, then $\mathrm{tw} \leq \mathrm{pw} \leq \mathrm{ctw}$. Our result implies that, for evaluating the Tutte polynomial at a given integer point, it does not give a substantial advantage to have small cutwidth instead of small treewidth. This is somewhat surprising since, for example, for computing the closely related chromatic number of a graph there exists a $2^{\mathrm{ctw}} n^{O(1)}$ time algorithm, but any $\mathrm{pw}^{o(\mathrm{pw})}\, n^{O(1)}$ time algorithm would contradict the ETH [124].

Of particular interest are the upper bounds in Theorem 5.1.1.b for the points $\{(x,y) : y = 1\}$, which are closely related to the problem of computing the number of forests in the input graph. One reason why this results stands out in particular is that it indicates an inherent asymmetry between the x- and y-axes, in this parameterized setting. In the general setting, problems related to the Tutte Polynomial often have a natural dual problem, which one can obtain by interchanging the x- and y-coordinates. For example the chromatic polynomial can be found (up to

some computable term f) as $\chi_G(\lambda) = f(\lambda)T(1-\lambda, 0)$, while the flow polynomial can be found as $C_G(\lambda) = g(\lambda)T(0, 1-\lambda)$. These two problems are equivalent on planar graphs, in the sense that the chromatic number of a planar graph is equal to the flow number of its dual graph.

We note that for this curve we have a #ETH bound, while for the other results of the form $c^{\mathrm{tw}}n^{O(1)}$ we have a stronger SETH bound. We also note that most our bounds use non-counting versions of ETH and SETH. The reason for this is that we will use the results from Chapter 3, which gives bounds for modular counting and thus cannot use #ETH or #SETH. We suspect that this is purely a technical hurdle and not a fundamental difference between the cases. Also note that for $(x-1)(y-1) = 2$, we **can** use #SETH, but this has been left out of the main theorem for the sake of brevity.

Techniques In order to get the classification, our first step follows the method of [106] to reduce the evaluation of $T(G; x, y)$ for all points in hyperbola $H_\alpha = \{(x, y) : (x-1)(y-1) = \alpha\}$ to the evaluation to a *single* point in H_α. This is achieved in [106] by some graph operations (*stretch* and *thickening*), but these may increase the involved width parameters. We refine these operations in Section 5.3 to avoid this.

With this step being made, several cases of Theorem 5.1.1 then follow from a combination of new arguments, some from earlier chapters, and previous work (including some very recent work such as [55]). We discuss the various cases in Section 5.4.

5.2 Preliminaries

Computational Model In this chapter we frequently have real (and some intermediate lemma's are even stated for complex) numbers as intermediate results of computations. However, as is common in this area we work in the word RAM model in which all basic arithmetic operations with such numbers can be done in constant time, and therefore this does not influence our running time bounds.

5.2.1 Brylawski's Tensor Product Formula

In Section 5.3 we will make use of Brylawski's tensor product formula [29] to reduce the computation of $T(G; x, y)$ to that of $T(G; x', y')$ for some other point (x', y'). The original formula is formulated in terms of pointed matroids, however we will only need the formulation for (multi)graphs. Before we can state the formula, we first need to introduce some notation.

Given graphs G and H, where an edge $e \in E(H)$ is labeled as a special edge, we define the *pointed tensor product*[1] $G \otimes_e H$ of G and H as the graph given by the following procedure. For every edge $f \in E(G)$ we first create a copy H_f of H, then identify f with the copy of the edge e in H_f and finally remove the edge f (and thus also the edge e) from the graph.

[1] Note that this is different from the standard tensor product for graphs.

Intuitively it might be easier to think of this product as replacing every edge of G with a copy $H \setminus e$, where two of the vertices in H are designated as gluing points. For example one could replace every edge with a path of length k by taking as H the cycle C_{k+1} on $k+1$ vertices, as seen in figure 5.2. This particular transformation and a closely related one will prove useful in our proofs. We have seen this transformation in before in Chapter 3, but recall the definition here.

Definition 2.6.2. *The* k-stretch *of a graph G is the graph obtained from G by replacing each edge with a path of length k. The* k-thickening *of a graph G is the graph obtained from G by replacing each edge with k parallel edges. We denote the k-stretch of G by ${}^{k}G$ and the k-thickening as by ${}_{k}G$.*

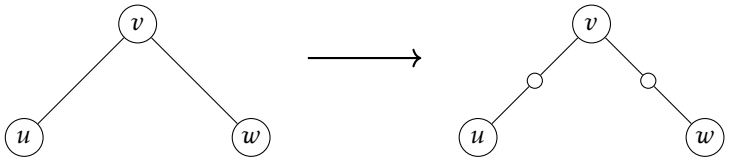


Figure 5.2: The pointed tensor product of the left-hand graph with a 3-cycle is given by the right-hand graph. The resulting operation is also referred to as a 2-stretch.

Note that the result of this product is not always unique, as one can choose which endpoint is identified with which. In this chapter we will only consider graphs H that are symmetric over e and thus the product is actually well-defined.

We are now ready to state Brylawski's tensor product formula. Let T_C and T_L be the unique polynomials that satisfy the following system of equations

$$(x-1)T_C(H;x,y) + T_L(H;x,y) = T(H \setminus e;x,y)$$
$$T_C(H;x,y) + (y-1)T_L(H;x,y) = T(H/e;x,y).$$

We define

$$x' = \frac{T(H\setminus e;x,y)}{T_L(H;x,y)} \qquad\qquad y' = \frac{T(H/e;x,y)}{T_C(H;x,y)}.$$

Let $n = |V(H)|$, $m = |E(H)|$ and $k = k(E(H))$. Brylawski's tensor product formula states that

$$T(G \otimes_e H;x,y) = T_C(H;x,y)^{m-n+k} T_L(H;x,y)^{n-k} T(G;x',y').$$

When we apply this formula to the k-stretch we find the following equality

$$(1 + a + \cdots + a^{k-1})^{k(E)} T\left(G; a^k, \frac{b + a + \cdots + a^{k-1}}{1 + a + \cdots + a^{k-1}}\right) = T({}^{k}G;a,b).$$

For the k-thickening we find

$$(1 + b + \cdots + b^{k-1})^{k(E)} T\left(G; \frac{a + b + \cdots + b^{k-1}}{1 + b + \cdots + b^{k-1}}, b^k\right) = T({}_{k}G;a,b).$$

5.3 Reducing Along the Curve H_α

In this section we describe how we can lift hardness results from a single point $(a,b) \in H_\alpha$ to the whole curve H_α. We summarize the results from this section in the following theorem.

Theorem 5.3.1. *Recall that $T(G;x,y)$ gives the Tutte polynomial of G and $\alpha := (a-1)(b-1)$.*

a. *Let $(a,b) \in \mathbb{C}^2$, such that $|a| \notin \{0,1\}$. Then there exists a polynomial time reduction from computing T along H_α for graphs of given tree-, path- or cutwidth, to computing T on (a,b). In this reduction, the treewidth remains* tw*, the cutwidth remains* ctw *and the pathwidth become at most* $\mathrm{pw}+2$.

b. *Let $(a,b) \in \mathbb{C}^2$, such that $|b| \notin \{0,1\}$ and $a \neq 0$. Then there exists a polynomial time reduction from computing T along H_α for graphs of given tree-, path- or cutwidth, to computing T on (a,b). In this reduction, the treewidth remains* tw*, the cutwidth becomes at most* $\mathrm{ctw}+2$ *and the pathwidth become at most* $\mathrm{pw}+2$.

c. *Let $(a,b) \in \mathbb{C}^2$, such that $|b| \notin \{0,1\}$ and $a = 0$. Then there exists a polynomial time reduction from computing T along H_α for graphs of given tree-, path- or cutwidth, to computing T on (a,b). In this reduction, the treewidth remains* tw*, the cutwidth becomes at most* $2\,\mathrm{ctw}$ *and the pathwidth become at most* $\mathrm{pw}+2$.

d. *Let $(a,b) \in \mathbb{C}^2$, such that $|a|,|b| \in \{0,1\}$. Then there exists a polynomial time reduction from computing T along H_α for graphs of given tree-, path- or cutwidth, to computing T on (a,b). In this reduction, the treewidth remains* tw*, the cutwidth becomes at most* $12\,\mathrm{ctw}$ *and the pathwidth become at most* $\mathrm{pw}+2$.

Note that there is a trivial reduction in the opposite direction, i.e. if we can compute T along H_α, then we can compute T on any $(a,b) \in H_\alpha$. Therefore, Theorem 5.3.1 lets us lift both algorithms and lower bounds from a point (a,b) to the whole curve H_α. While Theorem 5.1.1 only requires Theorem 5.3.1 to be stated for integer valued points, we will state it as the most general version we can prove. We note that for Theorem 5.1.1.a, we do not care too much about constant multiplicative factors in the cutwidth, since we have an ETH bound of the form $\mathrm{ctw}(G)^{o(\mathrm{ctw}(G))}$. For Theorem 5.1.1.b we only need fine-grained bounds on the treewidth and pathwidth, since we again have an ETH bound for cutwidth, this time of the form $2^{o(\mathrm{ctw})}$. Thus the blowup in the cutwidth is only relevant for Theorem 5.1.1.c. In this case the only integer valued points that fall under Theorem 5.3.1.d are $(-1,0),(0,-1)$ and $(-1,-1)$. These are all special points, which means that this item is not relevant for Theorem 5.1.1.c.

Earlier we defined the k-stretch and k-thickening of a graph (Definition 2.6.2). A new variant we introduce to keep the cutwidth low is defined as follows:

Definition 5.3.2. *We define the* insulated k-thickening $_{(k)}G$ *as the graph obtained by replacing every edge by a path of length* 3 *and then replacing the middle edge in each of these paths by k parallel edges.*

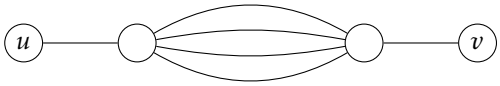


Figure 5.3: The result of applying the insulated 4-thickening to an edge between u and v.

5.3.1 Effect on Width Parameters

We now give three lemmas that show how these transformations effect the parameters we use.

Lemma 5.3.3. *Let G be a graph and $k \geq 1$ an integer. Then we have that* $\mathrm{tw}({}^{k}G) = \mathrm{tw}(G)$, $\mathrm{tw}({}_{k}G) = \mathrm{tw}(G)$ *and* $\mathrm{tw}({}_{(k)}G) = \mathrm{tw}(G)$.

Proof. First note that in all three cases G is a minor of the transformed graph and thus we have $\mathrm{tw}(G) \leq \mathrm{tw}({}^{k}G)$, $\mathrm{tw}(G) \leq \mathrm{tw}({}_{k}G)$ and $\mathrm{tw}(G) \leq \mathrm{tw}({}_{(k)}G)$. It remains to prove the upper bounds.

Note that parallel edges do not affect the treewidth of a graph, since any bag covering one of these edges will necessarily cover all of them. This means that the original tree decomposition is also a tree decomposition for the k-thickening ${}_{k}G$ and thus $\mathrm{tw}({}_{k}G) \leq \mathrm{tw}(G)$. It also means that for the purposes of finding a tree decomposition, the insulated k-thickening is equivalent to a 3-stretch. It remains to show that the k-stretch does not increase the treewidth.

Note that $\mathrm{tw}(G) = 1$ if and only if G is a tree. Since the k-stretch of a tree is also a tree, we find $\mathrm{tw}({}^{k}G) = 1$.

Now suppose that $\mathrm{tw}(G) \geq 2$. We will show that subdividing an edge does not affect the treewidth of the graph. By repeatedly subdividing edges we then find that the treewidth of the k-stretch ${}^{k}G$ is at most that of G.

Let $uv \in E(G)$ and let G' be the graph obtained by subdividing uv into uw and wv. Let x be some node in the tree decomposition of G such that $u, v \in B_x$. We create a tree decomposition of G' by adding a node x', with a corresponding bag $B_{x'} = \{u, v, w\}$, and connecting x' to x. It is easy to see that the resulting decomposition is still a tree decomposition. We conclude that $\mathrm{tw}({}^{k}G) \leq \mathrm{tw}(G)$. □

Lemma 5.3.4. *Let G be a graph and $k \geq 1$ an integer. Then we have that* $\mathrm{pw}({}_{k}G) = \mathrm{pw}(G)$, $\mathrm{pw}(G) \leq \mathrm{pw}({}^{k}G) \leq \mathrm{pw}(G) + 2$ *and* $\mathrm{pw}(G) \leq \mathrm{pw}({}_{(k)}G) \leq \mathrm{pw}(G) + 2$.

Proof. Like in the previous proof, we first note that in all three cases G is a minor of the transformed graph and thus we have $\mathrm{pw}(G) \leq \mathrm{pw}({}^{k}G)$, $\mathrm{pw}(G) \leq \mathrm{pw}({}_{k}G)$ and $\mathrm{pw}(G) \leq \mathrm{pw}({}_{(k)}G)$. It remains to prove the upper bounds.

Note that parallel edges do not affect the pathwidth of a graph. This means that the original path decomposition is also a path decomposition for the k-thickening ${}_{k}G$ and thus $\mathrm{pw}({}_{k}G) \leq \mathrm{pw}(G)$. Again, this also means that for the purposes of finding a path decomposition, the insulated k-thickening is equivalent to a 3-stretch. It remains to show that the k-stretch does not increase the pathwidth by more than an additive factor of 2.

Suppose we are given a path decomposition of G, of width $\mathrm{pw}(G)$. Whenever a new vertex v is introduced in a bag B_x, in the path decomposition of G, we add the following bags. For each edge $uv \in E(G)$ such that $u \in B_x$ let $w_1^{uv}, \dots, w_k^{uv} = v$ be the path replacing uv in ${}^kG$. We add the bags $B_x \cup \{w_i^{uv}, w_{i+1}^{uv}\}$ for $i = 1, k$ in order, one path at a time. Clearly the decomposition has width $\mathrm{pw}(G) + 2$ and every vertex and edge of ${}^kG$ is covered. Since the intermediate vertices on the paths only appear in two consecutive bags and any vertex from G is retained until all adjacent paths have been covered, no vertices are forgotten and then reintroduced. We find that it is a valid path decomposition and thus $\mathrm{pw}({}^kG) \leq \mathrm{pw}(G) + 2$. □

Lemma 5.3.5. *Let G be a graph and $k \geq 1$ an integer. Then we have that* $\mathrm{ctw}({}^kG) = \mathrm{ctw}(G)$, $\mathrm{ctw}({}_kG) = k\,\mathrm{ctw}(G)$ *and* $\mathrm{ctw}(G) \leq \mathrm{ctw}({}_{(k)}G) \leq \mathrm{ctw}(G) + k - 1$.

Proof. Again, we first note that in all three cases G is a minor of the transformed graph and thus we have $\mathrm{ctw}(G) \leq \mathrm{ctw}({}^kG)$, $\mathrm{ctw}(G) \leq \mathrm{ctw}({}_kG)$ and $\mathrm{ctw}(G) \leq \mathrm{ctw}({}_{(k)}G)$.

For the upper bounds, in each case we will show that, given a cut decomposition of width $\mathrm{ctw}(G)$, we can construct a decomposition that respects the given bounds.

Let $\pi = (v_1, \dots, v_n)$ be a cut decomposition of G, of width $\mathrm{ctw}(G)$. First note that the given decomposition already gives a decomposition of ${}_kG$, of width $k\,\mathrm{ctw}(G)$, and thus $\mathrm{ctw}({}_kG) \leq k\,\mathrm{ctw}(G)$. Since any parallel edges are necessarily cut by the same cuts we also find that any cut decomposition of ${}_kG$, of width $k\,\mathrm{ctw}(G)$, gives a decomposition of G, of width $\mathrm{ctw}(G)$. We therefore find a stronger lower bound of $k\,\mathrm{ctw}(G) \leq \mathrm{ctw}({}_kG)$.

Next, we will examine the k-stretch ${}^kG$. We will show that subdividing an edge does not increase the cutwidth. By repeatedly subdividing edges we then find that the k-stretch does not have larger cutwidth, i.e. $\mathrm{ctw}({}^kG) \leq \mathrm{ctw}(G)$. Let $uv \in E(G)$ such that $u < v$ in a given cut decomposition $\pi = (v_1, \dots, v_n)$ of G. We create a new graph G' from G, by adding a vertex w, edges uw and wv, and removing the edge uv. We construct a cut decomposition π' of G' as follows. If $v_i, v_j \in V(G)$, such that $i < j$ (in π), then we also set $v_i < v_j$ in π'. For w, we set $w < v_i$ if $u < v_i$ in π and $v_i < w$ otherwise. Note that for any cut of the decomposition, we have one of the following situations. (i) The cut appears before u or after v, in which case the cut contains the same edges in both decompositions. (ii) The cut appears[2] between u and v, in which case it contains uv in π and either uw or wv in π', but not both. In either case we find that the cut has not increased in width in π' and thus the decomposition has the same width.

Finally we examine the insulated k-thickening. As seen before we can subdivide edges without increasing the cutwidth. We will first create the 3-stretch of G, by subdividing each edge twice. We will take special care to fully subdivide an edge before moving on to the next one, so that the two new vertices on the edge appear next to each other in the cut decomposition. We then replace the middle edge of each created 3-path with k parallel edges. Since the endpoints of any such bundle of edges are next to each other in the cut decomposition, each cut contains at most

[2] In this case we use the convention that the cut immediately before and the cut immediately after w get associated with the same cut π.

one bundle and thus we increase the cutwidth by at most $k-1$ and thus $\mathrm{ctw}({}_{(k)}G) \leq \mathrm{ctw}(G)+k-1$. □

We remark that the only significant blowup is that of the cutwidth, when applying the k-thickening. We will therefore limit our use of this transformation as much as possible.

5.3.2 Reductions

We can now prove Theorem 5.3.1. We prove each case of the theorem separately. Note that Theorem 5.3.1.d follows from first applying Lemma 5.3.6 and then one of the other cases of Theorem 5.3.1.

Theorem 5.3.1.a. *(restated) Let $(a,b) \in \mathbb{C}^2$, such that $|a| \notin \{0,1\}$. Then there exists a polynomial time reduction from computing T along H_α for graphs of given tree-, path- or cutwidth, to computing T on (a,b). In this reduction, the treewidth remains* tw*, the cutwidth remains* ctw *and the pathwidth become at most* pw$+2$.

We prove this case using essentially the same proof as given in [106]. Note that in our setting we use Lemmas 5.3.3, 5.3.4 and 5.3.5 to ensure that relevant parameters are not increased by the operations we perform.

Proof. Recall that by Brylawski's tensor product formula [29], we find the following expression for the k-stretch of the graph G

$$(1+a+\cdots+a^{k-1})^{k(E)}T\left(G;a^k,\frac{b+a+\cdots+a^{k-1}}{1+a+\cdots+a^{k-1}}\right)=T({}^kG;a,b). \tag{5.2}$$

Note that

$$a^k-1=(1+a+\cdots+a^{k-1})(a-1)$$

and

$$\frac{b+a+\cdots+a^{k-1}}{1+a+\cdots+a^{k-1}}-1=\frac{b-1}{1+a+\cdots+a^{k-1}}.$$

We find that the point on which we evaluate $T(G)$ in (5.2) also lies on H_α.

By examining the formula for the Tutte polynomial, we find that for $n=|V(G)|$ the degree of the Tutte polynomial is at most n^2+n. By choosing $k=0,\ldots,n^2+n$, since $|a| \notin \{0,1\}$, we can find $T(G;x,y)$, for n^2+n+1 different values of $(x,y) \in H_\alpha$. By Lemma 2.7.1, we can now interpolate the univariate restriction

$$T_\alpha(G;t)=T\left(G;\frac{\alpha}{t}+1,t+1\right).$$

of $T(G)$ along H_α.

Note that by Lemmas 5.3.3 and 5.3.5 the k-stretch preserves both the cutwidth and the treewidth of the graph and by Lemma 5.3.4 the pathwidth increases by an additive constant. We find that any fine-grained parameterized lower bound for H_α extends to points (a,b). □

The next case is proven in a similar way, however it takes a bit more effort to make the numbers line up.

Theorem 5.3.1.b. *(restated) Let $(a,b) \in \mathbb{C}^2$, such that $|b| \notin \{0,1\}$ and $a \neq 0$. Then there exists a polynomial time reduction from computing T along H_α for graphs of given tree-, path- or cutwidth, to computing T on (a,b). In this reduction, the treewidth remains* tw*, the cutwidth becomes at most* ctw$+2$ *and the pathwidth become at most* pw$+2$.

Proof. By Theorem 5.3.1.a we may assume that $|a| = 1$. In the case that $a = 1$ we can still use the k-stretch, since

$$\begin{aligned}(1+a+\cdots+a^{k-1})^{-k(E)}T({}^kG;a,b) &= T\left(G;a^k,\frac{b+a+\cdots+a^{k-1}}{1+a+\cdots+a^{k-1}}\right)\\ &= T\left(G;1,\frac{b+k-1}{k}\right)\\ &= T\left(G;1,\frac{b-1}{k}+1\right).\end{aligned}$$

Where the first equality is (5.2). Since $b \neq 1$ we can find arbitrarily many points on the curve H_0^x this way. By Lemma 2.7.1 we can interpolate to find the $T(G)$ on the whole curve.

In the remaining case, i.e. $0 \neq a \neq 1$, we use the insulated k-thickening.

This results in the following transformation.

$$T({}_{(k)}G;a,b) = ((a+1)(1+b+\cdots+b^{k-1})+a^2)^{k(E)}(1+b+\cdots+b^{k-1})^{|V|-k(E)}T(G;A,B)$$

where

$$\begin{aligned}A &= a^2\left(1+\frac{a-1}{1+b+\cdots+b^{k-1}}\right) = a^2\left(\frac{a+b+\cdots+b^{k-1}}{1+b+\cdots+b^{k-1}}\right)\\ B &= 1+\frac{b^k-1}{(a+1)(1+b+\cdots+b^{k-1})+a^2}.\end{aligned}$$

Which allows us to move to a point with $|A| \notin \{0,1\}$, assuming that $|b| \notin \{0,1\}$ and $1 \neq a \neq 0$. We can then apply Theorem 5.3.1.a to to conclude the proof. Note that by Lemma 5.3.5 this transformation only increases the cutwidth by an additive factor of $k-1$.

It is not too dificult to see that it suffices to take either $k = 2$ or $k = 3$. Suppose that $k = 2$ does not work, then we have

$$\left|\frac{a+b}{1+b}\right| = 1.$$

From this we can deduce that $b = c \cdot a^{1/2}$ for some $c \in \mathbb{R}$. Now suppose that in this case $k = 3$ also does not work. We then find that

$$\left|\frac{a+c\cdot a^{1/2}+c^2\cdot a}{1+c\cdot a^{1/2}+c^2\cdot a}\right| = \left|\frac{a^{1/2}+c+c^2\cdot a^{1/2}}{1+c\cdot a^{1/2}+c^2\cdot a}\right| = 1.$$

By squaring this term and simplifying the resulting equations we find $a = 1$ and note that this case was handled previously. □

In the case that $a = 0$ (and $|b| \notin \{0, 1\}$), we first use the 2-thickening to compute

$$(1+b)^{|V|-k(E)} T\left(G; \frac{a+b}{1+b}, b^2\right) = T({}_2G; a, b).$$

and then apply Theorem 5.3.1.b. Note that this approach increases the cutwidth by a factor of 2 and thus for any lower bound $f(\mathrm{ctw})$ we would get on the curve H_α, we find a lower bound of $f(\mathrm{ctw}/2)$ for the point $(0, 1-\alpha)$. We find Theorem 5.3.1.c follows.

Theorem 5.3.1.c. *(restated) Let $(a, b) \in \mathbb{C}^2$, such that $|b| \notin \{0, 1\}$ and $a = 0$. Then there exists a polynomial time reduction from computing T along H_α for graphs of given tree-, path- or cutwidth, to computing T on (a, b). In this reduction, the treewidth remains* tw*, the cutwidth becomes at most* $2\,\mathrm{ctw}$ *and the pathwidth become at most* $\mathrm{pw}+2$*.*

The remaining case concerns points where $|a|, |b| \in \{0, 1\}$. We show that we can reduce non-special points of this type to a point that is covered by one of the other cases of Theorem 5.3.1. As noted before, this then proves Theorem 5.3.1.d.

Lemma 5.3.6. *Let $(a, b) \in \mathbb{C}^2$, such that $|a|, |b| \in \{0, 1\}$. If (a, b) is not one of the 8 special points or on H_1, then there exists some transformation f and a computable function g, such that $g(a, b) \neq 0$ and*

$$T(f(G); a, b) = g(a, b) T(G; a', b')$$

where either $|a'| \notin \{0, 1\}$ or $|b'| \notin \{0, 1\}$ and such that $\mathrm{tw}(f(G)) \leq \mathrm{tw}(G)$, $\mathrm{ctw}(f(G)) \leq 6\,\mathrm{ctw}(G)$ and $\mathrm{pw}(f(G)) \leq \mathrm{pw}(G) + 2$.

Proof. We adapt a proof due to [106].

Suppose that for every such transformation f we have either $|a'|, |b'| \in \{0, 1\}$ or a' and b' are not well-defined. We will show that in this case (a, b) must be one of the 8 special points or on H_1.

First assume that $|a| = |b| = 1$. Note that applying the 2-stretch gives

$$(a', b') = \left(a^2, \frac{b+a}{1+a}\right).$$

By assumption, we have either $a = -1$ or

$$\left|\frac{b+a}{1+a}\right| \in \{0, 1\},$$

which implies $b = -a$, $b = a^2$ or $b = 1$. Using the 2-thickening we find $b = -1$, $a = -b$, $a = b^2$ or $a = 1$. This reduces the list of possible points to

$$\begin{aligned}(a, b) \in \{&(1, 1), (-1, -1), (j, j^2), (j^2, j), (-1, i), (-1, -i),\\ &(i, -1), (-i, -1)\} \cup \{(a, -a) : |a| = 1\}\end{aligned}$$

This list can be further reduced by applying the 3-stretch and 3-thickening to find that only

$$(a,b) \in \{(1,1),(-1,-1),(j,j^2),(j^2,j),(-i,i),(i,-i)\}$$

remain, which are all special points.

Now suppose that $b = 0$. Note that, again applying the 2-stretch gives

$$(a',b') = \left(a^2, \frac{a}{1+a}\right).$$

By assumption we have either $a = -1$, $a = 0$ or $|1+a| = 1$. If $a = -1$ or $a = 0$, we find $(-1,0)$, which is a special point, and $(0,0) \in H_1$. If $|1+a| = 1$, we must have $a \in \{0, j, j^2\}$. If $a \in \{0, j, j^2\}$, then $(a',b') \in \{(0,0),(j,-j),(j^2,-j^2\}$ and we may again apply a 3-stretch or 3-thickening to conclude that

$$(a,b) \in \{(1,1),(-1,-1),(j,j^2),(j^2,j),(-i,i),(i,-i)\}.$$

We find that the only points left are $(-1,0)$, which is a special point, and $(0,0) \in H_1$.

Finally suppose that $a = 0$ and $|b| = 1$. We apply a 2-thickening to find

$$(a',b') = \left(\frac{b}{1+b}, b^2\right).$$

By assumption we either have $b = -1$ or $|1+b| = 1$. In the former case, we find $(0,-1)$, which is a special point. In the latter case we must have $b \in \{j, j^2\}$. If $b \in \{j, j^2\}$, then $(a',b') \in \{(-j,j),(-j^2,j^2\}$ and we may again apply a 3-stretch or 3-thickening to conclude that

$$(a,b) \in \{(1,1),(-1,-1),(j,j^2),(j^2,j),(-i,i),(i,-i)\}.$$

Note that the worst blowup in the cutwidth occurs when we apply a 2-thickening, followed by a 3-thickening, which effectively results in a 6-thickening and thus by Lemma 5.3.5 a multiplicative blowup in the cutwidth of 6. The treewidth and pathwidth bounds follow by Lemma 5.3.3 and Lemma 5.3.4 respectively. □

5.4 The complexity of computing T along H_α

In this section determine the complexity of computing T along the various curves H_α. When combined with the results of Section 5.3 we find proofs for the various cases of Theorem 5.3.1.

5.4.1 The Curve H_2

The curve H_2 is equivalent to the partition function of the Ising model. Both our proofs for the upper and lower bound on the complexity will make use of this fact

Theorem 5.4.1. *Computing the Tutte polynomial along the curve H_2 cannot be done in time $(2-\varepsilon)^{\mathrm{ctw}} n^{O(1)}$, unless #SETH fails.*

Proof. The Tutte polynomial on this curve specializes to the partition function of the Ising model on $G = (V, E)$ [10]. Computing this function in its entirety is equivalent to computing the generating function

$$C_G(z) = \sum_{k=0}^{\infty} c_k z^k$$

of the closed subgraphs in G [116]. Here c_k gives the the number of closed subgraphs with k edges, i.e. the number of edgesets $A \subseteq E$ such that every vertex has even degree in (V, A) and $|A| = k$. Computing all coefficients of C_G is clearly not easier than computing the number of closed subgraphs of maximum cardinality.

We finally show that computing the number of perfect matchings reduces to computing the number of maximum closed subgraphs, using a reduction from [111] which can be slightly altered to only increase the cutwidth by an additive factor of 2. There is a lower bound of 2^{ctw} for counting perfect matchings, due to [55], which finishes the proof. It remains to show that we can reduce #PerfectMatchings to #MaximumClosedSubgraphs, while increasing the cutwidth by at most 2.

First note that if every vertex in G has odd degree, then F is a perfect matching if and only if $E \setminus F$ is a maximum closed subgraph and thus the number of perfect matchings on G is equal to the number of maximum closed subgraphs. We will now construct a graph G' that has the same number of perfect matchings as G, but has only vertices with odd degree. Using the above remark we then find a reduction from #PerfectMatchings to #MaximumClosedSubgraphs. Also note that we can determine whether a graph has at least one perfect matching in polynomial time and thus we may assume that G has at least one perfect matching.

Let $v_1, v_2, \ldots, v_n$ be a cut decomposition of G of width ctw. Now let $v_1^e, v_2^e, \ldots, v_l^e$ be the vertices with even degree, in order of appearance in the cut decomposition. Since G has at least one perfect matching we find that n is even. Since the number of odd degree vertices in a graph is always even we also find that l is even. We now connect v_{2i-1}^e to v_{2i}^e using a 3-star, see figure 5.4, and call the resulting graph G'. Note that every vertex in G' has odd degree and that in a perfect matching the 'dangling' vertex d_i of a 3-star has to be matched to the center c_i of the 3-star. We find that the 3-stars have no effect on the number of perfect matchings of the graph.

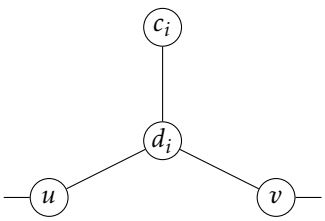


Figure 5.4: A 3-star between u and v.

We find a new cut decomposition by simply inserting the vertices c_i and d_i directly after v_{2i-1}^e in the cut decomposition. □

We now prove the following matching upper bound.

Theorem 5.4.2. *Let G be a graph with a given tree decomposition of width* tw. *There exists an algorithm that computes $T(G;a,b)$, for $(a,b) \in H_2$, in time $2^{\mathrm{tw}} n^{O(1)}$.*

Proof. As mentioned in the proof of Theorem 5.4.1, computing the Tutte polynomial along the curve is equivalent to computing the partition function of the Ising model [10]. Computing this function in its entirety is equivalent to computing the generating function

$$C_G(z) = \sum_{k=0}^{\infty} c_k z^k,$$

where c_k gives the the number of closed subgraphs with k edges [116]. We can compute the coefficients of this polynomial by computing the following dynamic programming table.

Let $S[x,p,k]$ be the number of edgesets A of size $|A| = k$ in the graph below bag B_x such that $deg_{G[A]}(v) \equiv_2 p(v)$. Note that the number of entries in the table is $2^{\mathrm{tw}} n^{O(1)}$, since we only need to consider $p \in \{0,1\}^{\mathrm{tw}}$ and $k \in [n^2]$. We may compute new entries as follows where $\emptyset$ denotes the empty vector. If B_x is a leaf bag then $S[x,\emptyset,0] = 1$ and $S[x,\emptyset,k] = 0$ otherwise. If x is a vertex-forget node for vertex v, then

$$S[x,p,k] = S[y,p',k],$$

where p' is the vector given by $p'(u) = p(u)$ for $u \neq v$ and $p'(v) = 0$ and y is the child node of x. If x is a vertex-introduce node for vertex v, then

$$S[x,p,k] = \begin{cases} S[y,p_{B_y},k] & \text{if } p(v) = 0, \\ 0 & \text{otherwise,} \end{cases}$$

where y is the child node of x. If x is an edge-introduce node for edge e, then

$$S[x,p,k] = S[y,p,k] + S[y,p +_2 \mathbf{1}_e, k-1],$$

where $\mathbf{1}_e(u) = 1$ if and only if u is one of the endpoints of e and where we use $+_2$ to indicate addition in $\mathbb{Z}_2^\ell$ for some ℓ. If x is a join node, with children y_1 and y_2, we use fast subset convolution as described in [58, Theorem A.6] to compute

$$S[x,p,k] = \sum_{i=0}^{k} \sum_{p_1 +_2 p_2 = p} S[y_1,p_1,i] S[y_2,p_2,k-i].$$

We can now find c_k as the entry $S[r,\mathbf{0},k]$. □

5.4.2 The Curve H_q for $q \in \mathbb{Z}_{\geq 3}$

These curves contain the points $(1-q,0)$, which count the number of q-colorings. Using previous results and a folklore algorithm, we find matching upper and lower bounds for these points and thus for the whole curves.

Theorem 5.4.3. *Let $q \in \mathbb{Z}_{\geq 3}$. Computing the Tutte polynomial along the curve H_q cannot be done in time $(q-\varepsilon)^{\mathrm{ctw}} n^{O(1)}$, unless SETH fails.*

Proof. Note that H_q contains the point $(1-q, 0)$. Computing the Tutte polynomial on this point is equivalent to counting the number of q-colorings of the graph G.

By choosing a modulus $p > q$ we can apply Theorem 3.1.1.c to find a lower bound of q^{ctw} on the time complexity of counting q-colorings modulo p. This lower bound clearly extends to general counting. □

In order to find a matching algorithm we combine the following folkore result with Theorem 5.3.1.

Theorem 5.4.4 (Folklore)**.** *Let G be a graph with a given tree decomposition of width* tw *and $q \in \mathbb{Z}_{\geq 3}$. There exists an algorithm that computes the number of q-colorings of G in time $q^{\mathrm{tw}} n^{O(1)}$.*

We immediately find the following algorithm.

Theorem 5.4.5. *Let G be a graph with a given tree decomposition of width* tw *and $q \in \mathbb{Z}_{\geq 3}$. There exists an algorithm that computes $T(G; a, b)$ for $(a, b) \in H_q$ in time $q^{\mathrm{tw}} n^{O(1)}$.*

Combining Theorems 5.3.1, 5.4.1, 5.4.2, 5.4.3 and 5.4.5, proves Theorem 5.1.1.c. We note that the cutwidth lower bound for $x = 0$ is equal to $(q-\varepsilon)^{\mathrm{ctw}/2} n^{O(1)}$, because of the factor 2 blowup in Theorem 5.3.1.c.

5.4.3 The Curve H_{-q} for $q \in \mathbb{Z}_{>0}$

These curves contain the points $(1+q, 0)$. Using the same results we used to prove Theorem 5.4.3 and exploiting the fact these results hold for modular counting, we find an ETH lower bound which matches the running time of the general algorithm described in Theorem 5.5.1.

Theorem 5.4.6. *Let $q \in \mathbb{Z}_{>0}$. Computing the Tutte polynomial along the curve H_{-q} cannot be done in* $\mathrm{ctw}^{o(\mathrm{ctw})}$ *time, unless ETH fails.*

Proof. Like mentioned earlier H_{-q} contains the point $(1+q, 0)$. For a prime $p > q$ we have that $T(G; 1+q, 0) \equiv_p T(G; 1+q-p, 0)$. This means that computing the Tutte polynomial modulo p at the point $(1+q, 0)$ is equivalent to counting the number of $p-q$-colorings of G modulo p. Since $q > 0$ and $p > q$ we find that $0 < p-q < p$ and thus as before, we may follow the proof or Theorem 3.1.1.c. Since the cutwidth of the construction in Theorem 3.1.1.c is $n + 6rp^{r+2}$ for some r dependant on $p-q$ and ε. From the $(p-q)^n (n+m)^{O(1)}$ lower bound on $\#_p\mathrm{CSP}(p-q, r)$, we find that there is no algorithm running in time $(p-q-\varepsilon)^{\mathrm{ctw}-6rp^{r+2}} n^{O(1)}$. We choose p such that for some

constant $\alpha < 1/6r$ we have $p = (\alpha\,\mathrm{ctw})^{1/(r+2)}$, we find a lower bound of

$$\begin{aligned}((\alpha\,\mathrm{ctw})^{1/(r+1)} - q - \varepsilon)^{(\mathrm{ctw}-6r\alpha\,\mathrm{ctw})} n^{O(1)} &= ((\alpha\,\mathrm{ctw})^{1/(r+1)} - q - \varepsilon)^{\mathrm{ctw}(1-6r\alpha)} n^{O(1)} \\ &\geq ((\alpha\,\mathrm{ctw} - q - \varepsilon)^{1/(r+1)})^{\mathrm{ctw}(1-6r\alpha)} n^{O(1)} \\ &\geq (\alpha\,\mathrm{ctw} - q - \varepsilon)^{(\mathrm{ctw}(1-6r\alpha))/(r+2)} n^{O(1)} \\ &\geq (\alpha\,\mathrm{ctw} - q - \varepsilon)^{\mathrm{ctw}(1/(2r)-3\alpha)} n^{O(1)}\end{aligned}$$

Since q is a constant we find that we cannot have an algorithm that runs in time $\mathrm{ctw}^{o(\mathrm{ctw})}$, unless ETH fails. □

By Theorem 5.5.1 points on this curve can be computed in time $\mathrm{tw}^{O(\mathrm{tw})}\, n^{O(1)}$.

5.4.4 The Curve H_0^x

The curve H_0^x contains the point $(1,2)$, which counts the number of connected edge-sets of a connected graph. Using results from Chapter 3 this gives an ETH lower bound which matches the running time of the general algorithm described in Theorem 5.5.1.

Theorem 5.4.7. *Let $0 < \alpha < 1$. Computing the Tutte polynomial along the curve H_0^x cannot be done in time $\mathrm{ctw}^{o(\mathrm{ctw})}\, n^{O(1)}$, unless ETH fails.*

Proof. Theorem 3.4.3 gives a lower bound of $(p-\varepsilon)^{\mathrm{ctw}} n^{O(1)}$ for counting connected edgesets modulo p. In the reduction we reduced to counting essentially distinct q-coloring modulo p, with cutwidth $\mathrm{ctw} + q^2$ and $p = q$. Thus we find a lower bound of $p^{\mathrm{ctw}-p^2} = (\alpha\,\mathrm{ctw})^{(1-\alpha)\,\mathrm{ctw}/2}$ for $p = (\alpha\,\mathrm{ctw})^{1/2}$. Since α is a constant we find that we cannot have an algorithm that runs in time $\mathrm{ctw}^{o(\mathrm{ctw})}$, unless ETH fails. □

By Theorem 5.5.1 points on this curve can be computed in time $\mathrm{tw}^{O(\mathrm{tw})}\, n^{O(1)}$. Combining Theorems 5.3.1, 5.4.6, 5.4.7 and 5.5.1 proves Theorem 5.1.1.a.

5.4.5 The Curve H_0^y

The curve H_0^y contains the point $(2,1)$, which counts the number of spanning forests of a graph. Using results from Chapter 4 this gives an algorithm which, as mentioned in Chapter 4, matches the an ETH lower bound we can trivially obtain from a bound by Brand et al. [28, Theorem 1 or Corollary 2].

Theorem 4.1.1. *[Theorem 1 in [28]] If #ETH holds then there exists constants $\varepsilon, C > 0$ such that no $O(2^{\varepsilon n})$ time algorithm can compute the number of all forests in a given simple n-vertex with at most Cn edges.*

Using Theorem 4.1.3 and Theorem 5.3.1 we find the following algorithms.

Theorem 5.4.8. *Let G be a graph with either a given tree decomposition of width* tw *or a path decomposition of width* pw*. There exist algorithms that computes $T(G;a,b)$ for $(a,b) \in H_0^y$ in time $64^{\mathrm{tw}(G)} n^{O(1)}$ and $4^{\mathrm{pw}(G)} n^{O(1)}$ respectively.*

Combining Theorems 5.3.1, 4.1.1 and 5.4.8 proves Theorem 5.1.1.b.

5.5 A General Algorithm

In this section we show how we can exploit bounded treewidth to compute the Tutte polynomial at any point in the plane, in FPT-time. For this we use a standard dynamic programming approach.

A linear (in the input size) time FPT-algorithm has previously been given by Noble [134]. This algorithm is double exponential in the treewidth, where the algorithm we give here has a running time of the form $2^{O(\mathrm{tw}\log(\mathrm{tw}))}n^{O(1)}$. We consider this an improvement for our purposes, since we are mainly interested in the dependence on the treewidth.

Theorem 5.5.1. *There is an algorithm that, given a graph G and a point (a,b), computes $T(G;a,b)$ in time $\mathrm{tw}^{O(\mathrm{tw})}\,n^{O(1)}$.*

Proof. Note that, in order to compute the Tutte polynomial, we only need to know the number $c_{i,j}$ of edgesets with i components and j edges, for $i,j=1,\ldots,n$. We can then compute

$$T(G;a,b)=\sum_{i,j=1}^{n} c_{i,j}(a-1)^{i-k(E)}(b-1)^{i+j-|V|},$$

in polynomial time.

We will now focus on computing the values of $c_{i,j}$. Let $(\mathbb{T},(B_x)_{x\in V(\mathbb{T})})$ be a rooted, nice tree decomposition with root r. We define $C_x(\pi,i,j)$ as the number of edge subsets of the graph covered by the subtree rooted[3] at bag B_x, with i components j edges and whose connected components give the partition π on B_x. At the leaves of the decomposition we have $B_x=$ and thus

$$C_x(\pi,i,j)=\begin{cases}1, & \text{if } \pi=\emptyset, i=j=0,\\ 0, & \text{otherwise.}\end{cases}$$

If x is an introduce node for vertex v with child y. For $N_\pi(v)$ the set of vertices in the same set of π, we have

$$C_x(\pi,i,j)=\begin{cases}\displaystyle\sum_{\emptyset\neq A\subseteq N_\pi(v)} C_y(\pi-v,i-|A|,j), & \text{if } N_\pi(v)\neq\emptyset,\\ C_y(\pi-v,i,j-1), & \text{otherwise.}\end{cases}$$

If x is a forget node for vertex v with child y, we have

$$C_x(\pi,i,j)=\sum_{\pi'\in 2^{B_y},\pi'|_{B_x}=\pi} C_y(\pi',i,j).$$

If x is a join node with children y and z, we have

$$C_x(\pi,i,j)=\sum_{k=0}^{i}\sum_{l=0}^{j}\sum_{\pi'\sqcup\pi''=\pi} C_y(\pi,k,l)C_z(\pi,i-k,j-l),$$

[3] In other words, all vertices that are in some bag y, such that any path in $\mathbb{T}$ from y to r must pass through x.

where $\pi' \sqcup \pi'' = \pi$ indicates that merging any overlapping sets in π' and π'' results in π. □

5.6 Conclusion

In this chapter we gave a classification of the complexity of evaluating the Tutte polynomial at integer points, parameterized by treewidth, pathwidth or cutwidth, into either computable

- in polynomial time,
- in $\mathrm{tw}^{O(\mathrm{tw})}\, n^{O(1)}$ time but not in $\mathrm{ctw}^{o(\mathrm{ctw})}\, n^{O(1)}$ time,
- in $q^{\mathrm{tw}} n^{O(1)}$ time but not in $2^{o(\mathrm{ctw})}$ (and for many points not even in $r^{\mathrm{ctw}} n^{O(1)}$ time for some constants $r < q$),

assuming the (Strong) Exponential Time Hypothesis.

This classification turned out to be somewhat surprising, especially considering the asymmetry between $H_0^x = \{(x,y) : x = 1\}$ and $H_0^y = \{(x,y) : y = 1\}$ that does not show up in other classifications such as the ones from [28, 106], but is hinted at in [61].

It would be interesting to see if our classification of the complexity of all points on $\mathbb{Z}^2$ can be extended to a classification of the complexity of all points on $\mathbb{R}^2$ (or even $\mathbb{C}^2$). Typically, evaluation at non-integer points can be reduced to integers points (leading to hardness for non-integer points), but we were not able to establish such a reduction without considerably increasing the width parameters.

It would also be interesting to see if a similar classification can be made when parameterizing by the vertex cover number instead of treewidth/pathwidth/cutwidth. We already know that the runtime of $2^n n^{O(1)}$ by Björklund et al. [14] for evaluating the Tutte polynomial cannot be strengthened to a general $2^{O(k)} n^{O(1)}$ time algorithm where k is the minimum vertex cover size of the input graph due to a result by Jaffke and Jansen [107], but this still leaves the complexity of evaluating at many other points open.

Finally it is still an open question what the complexity is of computing the Tutte polynomial modulo some prime p. Annan [5] discusses this in his PhD thesis and shows hardness for some of the points in $\mathbb{F}_p^2$. Goodall [96] has shown that $T(G;a,b)$ can also be computed in polynomial time for some points that are not special points, but over $\mathbb{F}_p^2$ behave like the complex valued special points $(-i,i),(i,-i),(j,j^2)$ and (j^2,j). An obvious conjecture is now that all other points, other that the special points and H_1, are $\#_p P$-hard, which one can think of as the modular equivalent of $\#P$-hardness.

Part III

Parameterized Logarithmic Space Complexity

6 Integer Multicommodity Flow Parameterized by Various Parameters

How could drops of water know themselves to be a river? Yet the river flows on.

Antoine de Saint-Exupery

6.1 Introduction

The Multicommodity Flow problem is the generalization of the textbook flow problem where, instead of just one commodity, multiple different commodities have to be transported through a network. The problem models important operations research questions and several variants of this problem exist (see e.g. [155]). In this chapter we consider the variant where for each commodity, a given amount of flow (the demand) has to be sent from the commodity's source to its sink, subject to a capacity constraint on the total amount of flow through each arc. The nature and computational complexity of the problem is highly influenced by (the structure of) the graph (bounded graph parameters, directed edges or undirected edges, etc.) and the capacities, demands, and flow value (integral or not, represented in unary or binary). When the flow values are allowed to be fractional, the problem can be trivially solved through a linear program (see e.g. [113, 120]).

In this chapter we focus on the parameterized complexity of the Integer Multicommodity Flow problem, where all the given capacities and demands are integers, and the output flow must also be integral. In doing so we encounter the notions of XNLP and XALP hardness for the first time (see Definition 2.5.3). The Integer Multicommodity Flow problem is widely studied and well known to be NP-hard even if all capacities are 1, on both directed and undirected graphs, even when there are only two commodities [76]. On directed graphs, it is NP-hard even for two commodities of demand 1 [83]. These strong hardness results have led to a wide range of heuristic solution methods being investigated as well as a substantial body of work on approximation algorithms. For surveys, see e.g. [9, 155, 156].

An important special case of Integer Multicommodity Flow and the main source of its computational hardness is the Edge Disjoint Paths problem. It can be readily seen that Integer Multicommodity Flow is equivalent to Edge Disjoint Paths when all capacities and demands are 1. Indeed, all aforementioned hardness results stem from this connection.

Parameter	unary capacities	binary capacities
pathwidth	XNLP-complete	para-NP-complete
treewidth	XALP-complete	para-NP-complete
weighted tree partition width	FPT (1)	FPT (1)
vertex cover	(2); in XP	(2); open

Table 6.1: Overview of our results for Integer 2-Commodity Flow. para-NP-complete means NP-complete for fixed value of parameter. (1) Capacities of arcs inside bags can be arbitrary, capacities of arcs between bags are bounded by weighted tree partition width. (2) Approximation, see Theorem 6.1.11; conjectured in FPT. For the undirected case, the same results hold, except that for the para-NP-completeness for the parameters pathwidth and treewidth, we need a third commodity.

Investigation of the parameterized complexity of Edge Disjoint Paths has recently been continued by considering structural parameterizations. Unfortunately, the problem is NP-hard for graphs of treewidth 2 [133] and even for graphs with a vertex cover of size 3 [79]. It is also W[1]-hard parameterized by the size of a vertex set whose removal leaves an independent set of vertices of degree 2 [89]. From an algorithmic perspective, Ganian and Ordyniak [89] showed that Edge Disjoint Paths is in XP parameterized by tree-cut width. Zhou et al. [160] give two XP algorithms for Edge Disjoint Paths for graphs of bounded treewidth: one for when the number of paths is small, and one for when a specific condition holds on the pairs of terminals. We note that these results imply an XP algorithm on graphs of bounded treewidth for a bounded number of commodities if the capacities are given in unary. Ganian et al. [90] give an FPT algorithm parameterized by the treewidth and degree of the graph. Friedrich et al. [87, 86] give approximation algorithms for multicommodity flow on graphs of bounded treewidth.

These results naturally motivate the question:

What can we say about the parameterized complexity of the general Integer Multicommodity Flow problem under structural parameterizations?

6.1.1 Results

We consider the Integer Multicommodity Flow problem for a small, fixed number of commodities. In particular, Integer ℓ-Commodity Flow is the variant in which there are ℓ commodities. Furthermore, we study the setting where some well-known structural parameter of the input graph, particularly its pathwidth or treewidth, is small.

The main results in this chapter show that Integer 2-Commodity Flow is unlikely to be fixed-parameter tractable parameterized by treewidth and or by pathwidth. However, instead of being satisfied with just a W[t]-hardness result for some t or any t, we seek stronger results using XNLP and XALP. An overview of the results can be found in Table 6.1.

We prove XNLP-completeness (and stronger) results for Integer ℓ-Commodity Flow. We distinguish how the capacities of arcs and edges are specified: these can be given in either unary or binary. First, we consider the unary case:

Theorem 6.1.1. *Integer 2-Commodity Flow with capacities given in unary, parameterized by pathwidth, is XNLP-complete.*

Theorem 6.1.2. *Undirected Integer 2-Commodity Flow with capacities given in unary, parameterized by pathwidth, is XNLP-complete.*

These hardness results follow by reduction from the XNLP-complete #Chained Multicolored Clique problem [22], a variant of the perhaps more familiar Multicolored Clique problem [78]. We follow a common strategy in such reductions, using vertex selection and edge verification gadgets. However, a major hurdle is to use flows to select vertices and verify the existence of edges to form the sought-after cliques. To pass this hurdle, we construct gadgets that use so-called Sidon sets (see Subsection 6.2.1) as flow values, combined with gadgets to check that a flow value indeed belongs to such a Sidon set.

For the parameter treewidth, we are able to show a slightly stronger result. It turns out that many problems that are XNLP-complete with pathwidth as parameter are XALP-complete with treewidth as parameter. We show that this phenomenon also holds for the studied Integer Multicommodity Flow problem:

Theorem 6.1.3. *Integer 2-Commodity Flow with capacities given in unary, parameterized by treewidth, is XALP-complete.*

The reduction is from the XALP-complete Tree-Chained Multicolored Clique problem [23] and follows similar ideas as the above reduction. Combining techniques of the proofs of Theorems 6.1.2 and 6.1.3 gives the following result.

Theorem 6.1.4. *Undirected Integer 2-Commodity Flow with capacities given in unary, parameterized by treewidth, is XALP-complete.*

Assuming the *Slice-wise Polynomial Space Conjecture* [137, 21], these results show that XP-algorithms for either directed or undirected Integer 2-Commodity Flow for graphs of small pathwidth or treewidth cannot use only $f(k)|x|^{O(1)}$ memory. Moreover, the XNLP- and XALP-hardness implies these problems are $W[t]$-hard for all positive integers t.

If the capacities are given in binary, then the problems become even harder.

Theorem 6.1.5. *Integer 2-Commodity Flow with capacities given in binary is NP-complete for graphs of pathwidth at most* 13.

Theorem 6.1.6. *Undirected Integer 3-Commodity Flow with capacities given in binary is NP-complete for graphs of pathwidth at most* 18.

Finally, we consider a variant of the Integer Multicommodity Flow problem where the flow must be *monochrome*, i.e. a flow is only valid when no edge carries more than one type of commodity. Then, we obtain hardness even for parameterization by the vertex cover number of the graph, for both variants of the problem.

Theorem 6.1.7. *Integer 2-Commodity Flow with Monochrome Edges is NP-hard for binary weights and vertex cover number 6, and W[1]-hard for unary weights when parameterized by the vertex cover number.*

Theorem 6.1.8. *Undirected Integer 2-Commodity Flow with Monochrome Edges is NP-hard for binary weights and vertex cover number 6, and W[1]-hard for unary weights when parameterized by the vertex cover number.*

To complement our hardness results, we prove two algorithmic results. Bodlaender et al. [17] had given FPT algorithms for several flow problems, using the recently defined notion of weighted tree partition width as parameter (see [17]). Weighted tree partition width can be seen as a variant of the notion of *tree partition width* for edge-weighted graphs, introduced by Seese [141] in 1985 under the name *strong treewidth*. See Section 2.3 for formal definitions of these parameters. We note that the known hardness for the vertex cover number [79] implies that Edge Disjoint Paths is NP-hard even for graphs of tree partition width 3. Here, we prove the following:

Theorem 6.1.9. *The Integer ℓ-Commodity Flow problem can be solved in time $O(2^{2^{\mathrm{tpw}^{3\ell\,\mathrm{tpw}}}} n)$, where* tpw *is the breadth of a given weighted tree partition of the input graph.*

Theorem 6.1.10. *The Undirected Integer ℓ-Commodity Flow problem can be solved in time $O(2^{2^{\mathrm{tpw}^{3\ell\,\mathrm{tpw}}}} n)$, where* tpw *is the breadth of a given weighted tree partition of the input graph.*

For the standard Integer 2-Commodity Flow problem with the vertex cover number of the input graph as parameter, we conjecture that this problem is in FPT. As a partial result, we can give the following approximation algorithms. Let $\mathrm{vc}(G)$ denote the vertex cover number of a graph G.

Theorem 6.1.11. *There exists a constant $C > 0$ such that the following holds. There is a polynomial-time algorithm that, given an instance of Integer 2-Commodity Flow on a graph G with demands d_1, d_2, either outputs that there is no flow that meets the demands or outputs a 2-commodity flow of value at least $d_i - C\,\mathrm{vc}^3$ for each commodity $i \in [2]$.*

Theorem 6.1.12. *There exists a constant $C > 0$ such that the following holds. There is a polynomial-time algorithm that, given an instance of Undirected Integer 2-Commodity Flow on a graph G with demands d_1, d_2, either outputs that there is no flow that meets the demands or outputs a 2-commodity flow of value at least $d_i - C\,\mathrm{vc}^3$ for commodity $i \in [2]$.*

6.2 Preliminaries

In this chapter, we consider both directed and undirected graphs. Graphs are directed unless explicitly stated otherwise.

6.2.1 Sidon Sets

A *Sidon set* is a set of positive integers $\{a_1, a_2, \ldots, a_n\}$ such that all pairs have a different sum, i.e., when $a_i + a_{i'} = a_j + a_{j'}$ then $\{i, i'\} = \{j, j'\}$. The notion of a Sidon set is equivalent to that of a Golomb ruler. In a Golomb ruler, pairs of different elements have unequal differences, i.e. if $i \neq i'$ and $j \neq j'$, then $|a_i - a_{i'}| = |a_j - a_{j'}|$, then $\{i, i'\} = \{j, j'\}$. A simple argument implies the following.

Theorem 6.2.1. *A Sidon set of n elements in* $[4n^2]$ *can be found in* $n\sqrt{n}\log^{O(1)}(n)$ *time and* $O(\log n)$ *space.*

Proof. We follow the argument of [25]. We note that, given a prime number p, Erdös and Turán [74] give an explicit construction of a Sidon set of size p:

$$\{2pk + (k^2 \mod p) | k \in \{0, 1, \ldots, p-1\}\}.$$

We use the following procedure to find a prime between n and $2n$ and then use the above expression to produce a sufficiently large Sidon set (removing any excess elements to get a set of exactly size n).

For each $\ell \in \{n, n+1, \ldots, 2n\}$ we check for $i = 1, \ldots, \sqrt{\ell}$ whether i divides ℓ. If no dividers are found this way we return $p = \ell$. Note that there must be at least one prime in this interval, by Bertrand's postulate [42, 3] and that the elements of the set are of value at most $4n^2$. Each division can be done in $\log^{O(1)}(n)$ time and the set can be constructed in $O(n\log(n))$ time and thus the whole procedure takes $n\sqrt{n}\log^{O(1)}(n)$ time. Each operation can be done using $O(\log(n))$ space and we only need to save p to memory to construct the set, hence $O(\log(n))$ space suffices. □

6.2.2 Multicommodity Flow Problems

We now formally define our flow problems. A *flow network* is a pair (G, c) of a directed (undirected) graph $G = (V, E)$ and a function $c : E \to \mathbb{N}_0$ that assigns to each arc (edge) a non-negative integer *capacity*.

For a positive integer ℓ, an *ℓ-commodity flow* in a flow network with sources $s_1, \ldots, s_\ell \in V$ and sinks $t_1, \ldots, t_\ell \in V$ is an ℓ-tuple of functions $f^1, \ldots, f^\ell : E \to \mathbb{R}_{\geq 0}$, that fulfils the following conditions:

- **Flow conservation.** For all $i \in [\ell]$, $v \notin \{s_i, t_i\}$, $\sum_{wv \in E} f^i(wv) = \sum_{vw \in E} f^i(vw)$.
- **Capacity.** For all $vw \in E$, $\sum_{i \in [\ell]} f^i(vw) \leq c(vw)$.

An ℓ-commodity flow is an *integer ℓ-commodity flow* if for all $i \in [c]$, $vw \in E$, $f^i(vw) \in \mathbb{N}_0$. The *value for commodity i* of an ℓ-commodity flow is equal to $\sum_{s_i w \in E} f^i(s_i w) - \sum_{w s_i \in E} f^i(w s_i)$. We shorten this to 'flow' when it is clear from context what the value of ℓ is and whether we are referring to an integer or non-integer flow.

The main problem considered in the chapter is as follows:

Integer ℓ-Commodity Flow
Input: A flow network $G = (V, E)$ with capacities c, sources $s_1, \ldots, s_\ell \in V$, sinks $t_1, \ldots, t_\ell \in V$, and demands $d_1, \ldots, d_\ell \in \mathbb{N}$.
Question: Does there exist an integer ℓ-commodity flow in G which has value d_i for each commodity $i \in [\ell]$?

The Integer Multicommodity Flow problem is the problem that consists of all instances of Integer ℓ-Commodity Flow for all non-negative integers ℓ.

For undirected graphs, flow still has direction, but the capacity constraint changes to:

- **Capacity.** For all $vw \in E$, $\sum_{i \in [\ell]} f^i(vw) + f^i(wv) \leq c(vw)$.

The undirected version of the Integer ℓ-Commodity Flow problem then is as follows:

Undirected Integer ℓ-Commodity Flow
Input: An *undirected* flow network $G = (V, E)$ with capacities c, sources $s_1, \ldots, s_\ell \in V$, sinks $t_1, \ldots, t_\ell \in V$, and demands $d_1, \ldots, d_\ell \in \mathbb{N}$.
Question: Does there exist an integer ℓ-commodity flow in G which has value d_i for each commodity $i \in [\ell]$?

Finally, we say that an ℓ-commodity flow $f^1, \ldots, f^\ell$ is *monochrome* if no arc (edge) has positive flow of more than one commodity. That is, if $f^i(e) > 0$ for some arc (or edge) e, then $f^{i'}(e) = 0$ for all $i' \in [\ell] \setminus \{i\}$. We can then immediately define monochrome versions of Integer ℓ-Commodity Flow and Undirected Integer ℓ-Commodity Flow in the expected way.

6.3 Hardness results

In this section, we give the proofs of our hardness results. The section is partitioned into three parts. We start by giving the results for the case of unary capacities, parameterized by pathwidth and parameterized by treewidth, both for the directed and undirected cases. This is followed by our results for binary capacities in these settings. Finally, we give the results for graphs of bounded vertex cover, for both the unary and binary case.

6.3.1 Unary Capacities

We prove our hardness results for Integer Multicommodity Flow with unary capacities, parameterized by pathwidth and parameterized by treewidth. We aim to reduce from Chained Multicolored Clique (for the parameter pathwidth) and Tree-Chained Multicolored Clique (for the parameter treewidth). We first introduce a number of gadgets: subgraphs that fulfill certain properties and that are used in the hardness constructions. After that, we give the hardness results for directed graphs, followed by reductions from the directed case to the undirected case.

Before we start describing the gadgets, it is good to know that all constructions will have disjoint sources and sinks for the different commodities. We will set the

demands for each commodity equal to the total capacity of the outgoing arcs from the sources, which is equal to the total capacity of the incoming arcs to the sinks. Thus, the flow over such arcs will be equal to their capacity.

Furthermore, throughout this section, our constructions will have two commodities. We name the commodities 1 and 2, with sources s_1, s_2 and sinks t_1, t_2, respectively.

Gadgets

We define two different types of (directed) gadgets. Given an integer a, the *a-Gate gadget* either can move 1 unit of flow from one commodity from left to right, or at most a units of flow from the other commodity from top to bottom, but not both. Hence, it models a form of choice. This gadget will grow in size with a, and thus will only be useful if the input values are given in unary. Given a set S of integers and a large integer L (larger than any number in S), the (S, L)-*Verifier* is used to check if the flow over an arc belongs to a number in S. The a-Gate gadget is used as a sub-gadget in this construction. In our reduction, later, we will use appropriately constructed sets S to select vertices or to check for the existence of edges. Both types of gadget have constant pathwidth, and thus constant treewidth.

When describing the gadgets and proving that their tree- or pathwidth is bounded, it is often convenient to think of them as puzzle pieces being placed in a bigger mold. Formally, a *(puzzle) piece* is a directed (multi-)graph H given with a set $B^- \subseteq V(H)$ of vertices that have in total a incoming arcs without tail (*entry arcs*) and a set $B^+ \subseteq V(H)$ of vertices that have in total b outgoing arcs without head (*exit arcs*). The sets B^- and B^+ are disjoint and we call the vertices of B^- and B^+ the *boundary vertices* of H. It is a *path piece* if H has a path decomposition such that all vertices of B^- are (also) in the first bag and all vertices of B^+ are (also) in the last bag.

Now let G be any directed graph or piece. We say that the piece (H, B^-, B^+, a, b) is a *valid* piece for $v \in V(G)$ if the in-degree of v is a and the out-degree of v is b. Then the *placement* of the valid piece for v in G replaces v by H such that the original incoming and outcoming arcs of v are identified with the entry and exit arcs (respectively) of H in any way that forms a bijection. This terminology enables the following convenient lemmas:

Lemma 6.3.1. *Let G be a path piece with boundary vertices B^-, B^+. Let $S \subseteq V(G)$. Suppose that the path decomposition $(\{X_i \mid i \in I\}, T = (I, F))$ of G has width $\mathrm{pw}(G)$ and, for every $v \in S$, all in-neighbors of v also appear in the first bag containing v. Moreover, for every $v \in S$, let $(H_v, B_v^-, B_v^+, a_v, b_v)$ be a valid path piece for v such that the assumed path decomposition of H_v has width $\mathrm{pw}(H_v)$. Let G' be obtained from G by the placement of the pieces of H_v in G for all $v \in S$. Then G' is a path piece such that the required path decomposition has width at most the maximum over all $i \in I$ of:*

$$|X_i \setminus S| + \max_{v \in S \cap X_i} \left\{ \mathrm{pw}(H_v) + 1 + \sum_{v' \in (S \cap X_i) \setminus \{v\}} \max\{|B_{v'}^-|, |B_{v'}^+|\} \right\} - 1.$$

Proof. We modify the path decomposition for G. We may assume that the path decomposition is such that for two consecutive bags $X_i, X_{i'}$, it holds that $|X_i \triangle X_{i'}| = 1$.

For every $v \in S$, let X_v be the first bag containing v (note that X_v will be the same for all v in the first bag of the path decomposition of G, but will otherwise be unique). Then, for each $v \in S$, iteratively, replace v in X_v by B_v^-. Then, create a number of copies of the $i \in I$ corresponding to X_v, where the number of copies is equal to the number of nodes of the assumed path decomposition of H_v. To each of these new bags, add the vertices in the bags of the assumed path decomposition of H_v in the natural order. We need special care again for the first bag of the path decomposition: here we expand the decomposition for one vertex after the other. Finally, we add B_v^+ to all further bags containing v. The claimed bound immediately follows from this construction. □

With some additional assumptions on the path decomposition, we find the following strengthening of Lemma 6.3.1.

Lemma 6.3.2. *Let G be a path piece with boundary vertices B^- and B^+. Let $S \subseteq V(G)$. Suppose that G has a path decomposition $(\{X_i \mid i \in I\}, T = (I, F))$ of width $\mathrm{pw}(G)$, the first bag contains at most one vertex from S, no bag contains more than two vertices from S, and, for every $v \in S$, every in-neighbor u of v is contained in the first bag containing v and if $u \in S$, not in any subsequent bags. Moreover, for every $v \in S$, let $(H_v, B_v^-, B_v^+, a_v, b_v)$ be a valid path piece for v such that the assumed path decomposition of H_v has width $\mathrm{pw}(H_v)$. Let G' be obtained from G by the placement of the pieces of H_v in G for all $v \in S$. Then G' is a path piece such that the required path decomposition has width at most the maximum over all $i \in I$ of:*

$$|X_i \setminus S| + \max\left\{ \max_{v \in S \cap X_i} \{\mathrm{pw}(H_v) + 1\}, \sum_{v \in S \cap X_i} \max\{|B_v^-|, |B_v^+|\} \right\} - 1.$$

Proof. We modify the path decomposition for G. We may assume that the path decomposition is such that for two consecutive bags $X_i, X_{i'}$, it holds that $|X_{i'} \setminus X_i| \leq 1$. For every $v \in S$, let X_v be the first bag containing v (note that X_v is unique by the previous assumption and the assumption that the first bag contains at most one vertex from S). Treat the bags of the path decomposition in order of the path order on T. Consider the vertex $v \in S$ for which X_v comes first in this order. Replace v by B_v^- in this bag. Then, create a number of copies of this bag equal to the number of bags of the assumed path decomposition of H_v, and insert these bags (joined with the vertices of $X_v \setminus \{v\}$) into the path decomposition after X_v. At the end, we have a bag containing $(X_v \setminus \{v\}) \cup B_v^+$, because $(H_v, B_v^+, B_v^-, a_v, b_v)$ is a path piece. We now continue along the path order of T and replace v by B_v^+ in every bag we encounter, until the first bag we encounter containing a vertex $v' \in S \setminus \{v\}$. In this bag, we still replace v by B_v^+, then replace v' by $B_{v'}^-$ and create a new subsequent bag where we remove B_v^+. This yields a bag containing $(X_{v'} \setminus \{v, v'\}) \cup B_{v'}^-$. Since v will not appear in any further bags by the assumptions of the lemma, this is safe. Then, we continue with the same treatment for v' as we did before with v, etc., all the way until we reach the end of T. The claimed bound immediately follows from this construction. □

We can prove a similar lemma with respect to tree decompositions. A piece (H, B^-, B^+, a, b) is a *tree piece* if H has a tree decomposition that has a bag containing $B^- \cup B^+$.

Remark 6.3.3. *Any path piece (H, B^-, B^+, a, b) with an assumed path decomposition of width $\mathrm{pw}(H)$ is also a tree piece with a tree decomposition of width at most $\mathrm{pw}(H) + |B^+|$ by adding B^+ to every bag.*

Lemma 6.3.4. *Let G be a tree piece with boundary vertices B^-, B^+ such that the assumed tree decomposition $(\{X_i \mid i \in I\}, T = (I, F))$ has width $\mathrm{tw}(G)$. Let $S \subseteq V(G)$. For every $v \in S$, let $(H_v, B_v^-, B_v^+, a_v, b_v)$ be a valid tree piece for v such that the assumed tree decomposition of H_v has width $\mathrm{tw}(H_v)$. Let G' be obtained from G by the placement of the pieces of H_v in G for all $v \in S$. Then G' is a tree piece such that the required tree decomposition has width at most:*

$$\max\left\{\max_{v\in S}\{\mathrm{tw}(H_v)\}, \max_{i\in I}\left\{|X_i \setminus S| + \max_{v\in S\cap X_i}\{|B_{v'}^- \cup B_{v'}^+|\}\right\} - 1\right\}.$$

Proof. We modify the tree decomposition for G. For each $v \in S$, replace v in all bags containing v by $B_v^- \cup B_v^+$. For each $v \in S$, add the assumed tree decomposition of H_v to T by adding an edge between the node of this tree decomposition whose bag contains $B^- \cup B^+$ and some node $i \in I$ for which v used to be in X_i. The claimed bound immediately follows from this construction. □

Remark 6.3.5. *The same lemmas hold (with simple modifications) in the case of directed or undirected graphs for directed or undirected path/tree pieces respectively.*

We now describe both gadgets in detail.

a-Gate Gadget

Let a be a positive integer. The *a-Gate gadget* gadget will allow a units of flow of commodity 2 through one direction, unless 1 unit of flow of commodity 1 flows through the other direction.

The gadget is constructed as follows. See Figure 6.1 for its schematic representation and for an example with $a = 4$. We build a directed path P with $2 \cdot a$ vertices. We add two additional vertices v and w. The vertex v has arcs towards the first, third, fifth, etc. vertices of the path, and w has arcs from the second, fourth, sixth, etc. vertices of the path. All these arcs and the arcs of the path have capacity 1. We add an incoming arc of capacity 1 to the leftmost vertex x of the path and an incoming arc of capacity a to v. We call these the *entry arcs* of the gadget. We add an outgoing arc of capacity 1 at the rightmost vertex y of the path and an outgoing arc with capacity a to w. We call these the *exit arcs* of the gadget.

We call v, w, x, y the *boundary vertices* of the gadget. Note that all arcs incoming to or outgoing from the gadget are incident on boundary vertices.

Observe that this gadget can be constructed in time polynomial in the given value of a if a is given in unary, as the gadget has size linear in a. We capture the functioning of the gadget in the following lemma.

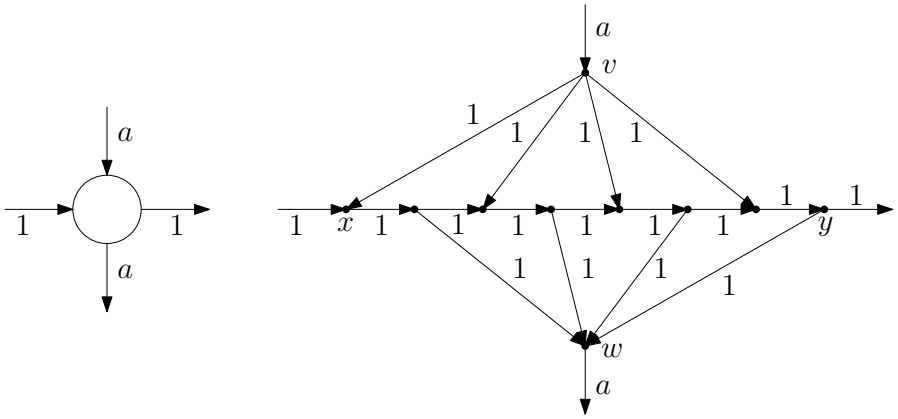


Figure 6.1: The a-Gate gadget. Left: the schematic representation of the gadget, with its entry and exit arcs. Right: the full construction for $a = 4$, with arcs labelled by their capacities.

Lemma 6.3.6. *Consider the a-Gate gadget for some integer a. Let f be some 2-commodity flow such that the entry arc at x and the exit arc at y only carry flow of commodity 1 and such that the entry arc at v and the exit arc at w only carry flow of commodity 2. Then:*

- *If v receives a units of commodity 2, then x receives no flow of commodity 1.*
- *If x receives 1 unit of commodity 1, then v receives no flow of commodity 2.*

Proof. Suppose that v receives a units of commodity 2. Then, every arc leaving v is used to capacity by commodity 2. Since the exit arc at y can only have flow of commodity 1, the flow of commodity 2 can only exit the gadget at w. This means that the arc leaving x is used to capacity by commodity 2. As such, x cannot receive any flow of commodity 1.

Suppose that x receives 1 unit of commodity 1. Since w can only receive flow of commodity 2, the flow of commodity 1 can only exit the gadget at y. This means that all the arcs in the path P are used to capacity by commodity 1. As such, w cannot receive any flow of commodity 2. □

Lemma 6.3.7. *For any integer a, the a-Gate gadget is a path piece such that the required path decomposition has width 3.*

Proof. The gadget is a piece by construction, with $B^- = \{v, x\}$, $B^+ = \{w, y\}$, and $a = b = 2$. To construct the path decomposition, begin with a path decomposition where the endpoints of each edge on the path from x to y are in a consecutive bags together. Add v, w to every bag. This is a valid path decomposition of width 3. □

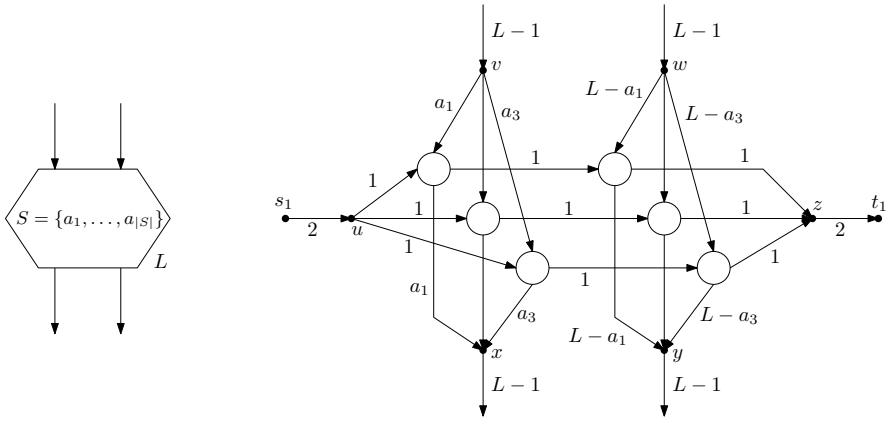


Figure 6.2: The (S, L)-Verifier gadget. Left: a schematic representation of the gadget, with its entry and exit arcs. The value on the bottom-right of the schematic representation denotes the sum of the incoming flows to the gadget. Right: the graph that realises the gadget for $S = \{a_1, a_2, a_3\}$, with arcs labelled by their capacities (the unlabelled arcs have capacities a_2 and $L - a_2$ respectively; their labels are omitted for clarity).

Verifier Gadget

Let L be a (typically large) integer. Let $S \subseteq [L-1]$ be a set of integers, where $S = \{a_1, \ldots, a_{|S|}\}$. An (S, L)-*Verifier* gadget is used to verify that the amount of flow through an edge is in S.

The gadget is constructed as follows. See Figure 6.2 for its schematic representation and for an example with $S = \{a_1, a_2, a_3\}$. We add six vertices u, v, w, x, y, z; these are the *boundary vertices* of this gadget. We add arcs s_1u and zt_1 of capacity $|S| - 1$. Then, v and w have incoming arcs of capacity $L-1$ (the *entry arcs* of this gadget) and x and y have outgoing arcs of capacity $L-1$ (the *exit arcs* of this gadget). Finally, we have $|S|$ rows of two Gate gadgets each. The ith row has an a_i-Gate gadget and an $(L - a_i)$-Gate gadget with the following arcs:

- an arc of capacity 1 from u to the first Gate gadget,
- an arc of capacity a_i from v to the first Gate gadget,
- an arc of capacity a_i from the first Gate gadget to x,
- an arc of capacity 1 from the first to the second Gate gadget,
- an arc of capacity $L - a_i$ from w to the second Gate gadget,
- an arc of capacity $L - a_i$ from the second Gate gadget to y,
- an arc of capacity 1 from the second Gate gadget to z.

Observe that this gadget can be constructed in time polynomial in the values of $a_1, \ldots, a_{|S|}$ if they are given in unary, as the Gate gadgets have size linear in the given value. We capture the functioning of the gadget in the following lemma.

Lemma 6.3.8. *Consider the (S, L)-Verifier gadget for some integer L and some $S \subseteq [L-1]$. Let f be some 2-commodity flow such that s_1 sends $|S|-1$ units of flow of commodity 1 to u, t_1 receives $|S|-1$ units of flow of commodity 1 from z. Suppose there is an integer $\alpha \in [L-1]$ such that v receives α units of flow of commodity 2 over its entry arc, w receives $L-\alpha$ units of flow of commodity 2 over its entry arc, and neither entry arc receives flow of commodity 1. Then:*

- *$\alpha \in S$,*
- *x sends α units of flow of commodity 2 over its exit arc,*
- *y sends $L-\alpha$ units of flow of commodity 2 over its exit arc.*

Conversely, if $\alpha \in S$, then there exists a 2-commodity flow that fulfills the conditions of the lemma.

Proof. Since u receives $|S|-1$ units of flow of commodity 1 and has $|S|$ outgoing arcs, u sends 1 unit of flow of commodity 1 over all but one of its outgoing arcs. Recall that v and w do not receive flow of commodity 1. If 1 unit of flow of commodity 1 is sent over say the ith outgoing arc of u, in order to arrive at z, it must go through the a_i-Gate gadget and the corresponding $(L-a_i)$-Gate gadget. The same amount of flow must also leave these gadgets and thus by Lemma 6.3.6, these gadgets cannot transfer flow of commodity 2. Thus, the flow of commodity 2 that v and w receive must go through the Gate gadgets of the row where u has not sent any flow to, say this is row j. x and y must send flow through arcs of capacity a_j and $L-a_j$, respectively. The capacities of the arcs and the gadgets enforce that $\alpha \leq a_j$ and $L-\alpha \leq L-a_j$, respectively. Thus, $\alpha = a_j$, and so $\alpha \in S$.

All the flow that the $(L-a_j)$-Gate gadget receives (which is only of commodity 2) must be sent to y, since t_1 is a sink. Hence, y sends $L-\alpha$ flow through its exit arc. The flow that the a_j-Gate gadget receives (which is only of commodity 2) must all be sent to x: we cannot send even 1 unit of flow over the horizontal arc to the second gadget, as the second gadget would then receive $L-\alpha+1$ units of flow of commodity 2. However, it cannot send flow of commodity 2 to z (as z only has an arc to t_1) and can send out at most $L-\alpha$ flow to y. Thus, x sends α units of flow over its exit arc.

For the converse, u sends 1 unit of flow of commodity 1 through all Gate gadgets of the values unequal to α, and α and $L-\alpha$ flow through the two other Gate gadgets. □

We note that in the later proofs, for a particular choice of L, we may also use Verifier gadgets with capacities $2L-1$, and incoming and outgoing flows adding up to $2L$ instead of L. We will explicitly indicate that, also in the schematic representation.

Lemma 6.3.9. *For any integer L and any $S \subseteq [L-1]$, the (S, L)-Verifier gadget is a path piece such that the required path decomposition (ignoring s_1 and t_1) has width 9.*

Proof. The gadget is a piece by construction, with $B^- = \{u, v, w\}$ and $B^+ = \{x, y, z\}$. If we treat the Gate gadgets as vertices, then we can immediately construct a path decomposition of width 7, by putting u, v, w, x, y, z in all bags and adding the two vertices corresponding to each row of Gate gadgets in successive bags. Since the Gate gadgets are each path pieces with a decomposition of width 3 by Lemma 6.3.7 and each have two entry and exit arcs, we obtain a path decomposition of the whole Verifier of width 9 by Lemma 6.3.2. □

Reductions for Directed Graphs

Using the gadgets we just proposed, we show our hardness results for Integer 2-Commodity Flow (i.e. the case of directed graphs) with pathwidth and with treewidth as parameter.

We note that our hardness construction will be built using only Verifier gadgets as subgadgets. The entry arcs and exit arcs of this gadget are meant to transport solely flow of commodity 2 per Lemma 6.3.8. Hence, in the remainder, it helps to think of only commodity 2 being transported along the edges, so that we may focus on the exact value of that flow to indicate which vertex is selected or whether two selected vertices are adjacent. We later make this more formal when we prove the correctness of the reduction.

Theorem 6.1.1. *Integer 2-Commodity Flow with capacities given in unary, parameterized by pathwidth, is XNLP-complete.*

Proof. Membership in XNLP can be seen as follows. Take a path decomposition of G, say with successive bags $(X_1, \ldots, X_r)$. One can build a dynamic programming table, where each entry is indexed by a node j with associated bag X_j and a function $f_j^i : X_j \to [-C, C]$, where C is some upper bound on the maximum value of the flow of any commodity (note that C is linear in the input size), for $i = 1, 2$ and $1 \leq j \leq r$. One should interpret f_j^i as mapping each vertex $v \in X_j$ to the net difference of flow of commodity i in- or outgoing on that vertex in a partial solution up to bag X_j. The content of the table is a Boolean representing whether there is a partial flow satisfying the requirements that f_j^i sets. Basic application of dynamic programming on (nice) path decompositions can solve the Integer 2-Commodity Flow problem with this table. This dynamic programming algorithm can be transformed to a non-deterministic algorithm by not building entire tables, but instead guessing one element for each table with positive Boolean. The guessed element of the table can be represented by $O(\mathrm{pw}(G) \log C)$ bits; in addition, we need $O(\log n)$ bits to know which bag of the path decomposition we are handling, and to look up relevant information of the graph. This yields a non-deterministic algorithm with $O(\mathrm{pw}(G) \log C + \log n)$ memory.

For the hardness, we use a reduction from Chained Multicolor Clique (see Theorem 2.5.7). Suppose we have an instance of Chained Multicolor Clique, with a graph $G = (V, E)$, coloring $c : V \to [k]$, and partition $V_1, \ldots, V_r$ of V.

Build a Sidon set with $|V|$ numbers by applying the algorithm of Theorem 6.2.1. Following the same theorem, the numbers are in $[4|V|^2]$. We set $L = 4|V|^2 + 1$. To

each vertex $v \in V$, we assign a unique element of the set S, denoted by $S(v)$. For any subset $V' \subseteq V$, let $S(V') = \{S(v) \mid v \in W\}$. For any subset $E' \subseteq E$, let $S(E') = \{S(u) + S(v) \mid uv \in E'\}$.

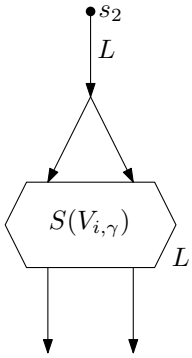


Figure 6.3: Vertex selector gadget used in the proof Theorem 6.1.1. The flow sent through the $(S(V_{i,\gamma}), L)$-Verifier will correspond to selecting a vertex from the class $V_{i,\gamma}$.

We now describe several (further) gadgets that we use to build the full construction. Let $V_{i,\gamma}$ be the vertices in V_i with color γ. Each set $V_{i,\gamma}$ is called a *class*. For each class $V_{i,\gamma}$, we use a *Vertex selector gadget* to select the vertex from $V_{i,\gamma}$ that should be in the solution to the Chained Multicolored Clique instance. This gadget (see Figure 6.3) consists of a single $(S(V_{i,\gamma}), L)$-Verifier gadget, where its entry arcs jointly start in a single vertex that in turn has a single arc from s_2 of capacity L. Intuitively, we select some $v \in V_{i,\gamma}$ if and only if the left branch receives $S(v)$ flow and the right branch receives $L - S(v)$ flow.

For each pair of incident classes, we construct an *Edge check gadget*. That is, we have an Edge check gadget for all classes $V_{i,\gamma}$ and $V_{i',\gamma'}$ with $|i - i'| \leq 1$, and $\{i, \gamma\} \neq \{i', \gamma'\}$. Let $E_{i,\gamma}^{i',\gamma'} \subseteq E$ denote the set of edges between $V_{i,\gamma}$ and $V_{i',\gamma'}$. An Edge check gadget will check if two incident classes have vertices selected that are adjacent. The gadget (see Figure 6.4) consists of a central $(S(E), 2L)$-Verifier gadget (note that we could also use an $(S(E_{i,\gamma}^{i',\gamma'}), 2L)$-Verifier gadget instead, but this is not necessary). Its entry arcs originate from two vertices that have as incoming arcs the exit arcs of an $(S(V_{i,\gamma}), L)$- and $(S(V_{i',\gamma'}), L)$-Verifier gadget. Its exit arcs head to two vertices that have as outgoing arcs the entry arcs of a different $(S(V_{i,\gamma}), L)$- and $(S(V_{i',\gamma'}), L)$-Verifier gadget. The gadget thus has four entry arcs and four exit arcs, corresponding to the entry arcs of the first two Verifier gadgets and the exit arcs of the last two Verifier gadgets.

Intuitively, if the entry arcs have flow (of commodity 2) of value $S(v)$, $L - S(v)$, $S(w)$, and $L - S(w)$ consecutively (refer to Figure 6.4), then there is a valid flow if and only if $vw \in E_{i,\gamma}^{i',\gamma'}$. Note that the sum $S(v) + S(w)$ is unique, because S is a Sidon set, and thus so is $2L - (S(v) + S(w))$. Hence, the only way for the flow to split up again and leave via the exit arcs is to split into $S(v)$, $S(w)$, $L - S(v)$, and $L - S(w)$; otherwise,

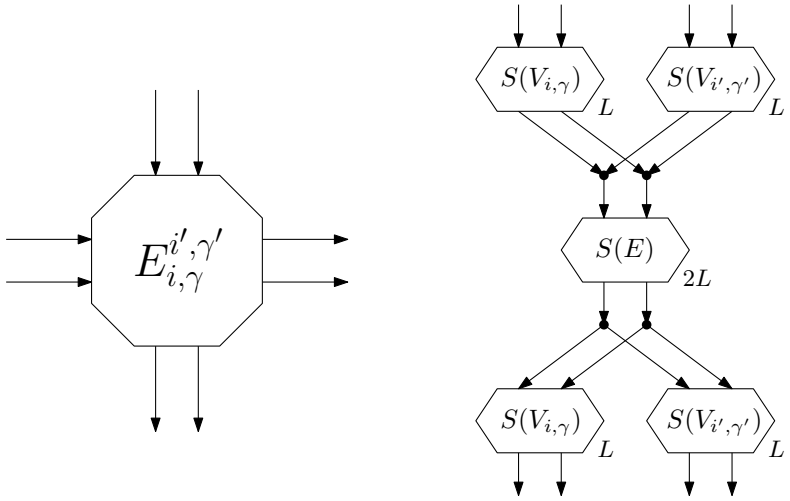


Figure 6.4: Edge check gadget for the combination $V_{i,\gamma}$ and $V_{i',\gamma'}$. Left: the schematic representation of the gadget. The flow corresponding to the $V_{i,\gamma}$ vertex enters on the left and exits on the right. The flow corresponding to the $V_{i',\gamma'}$ vertex enters from the top and exits at the bottom. Right: the realization of the gadget. The flow sent through the $(S(E), 2L)$-Verifier must correspond to a unique sum of two incident vertices from the classes $V_{i,\gamma}$ and $V_{i',\gamma'}$. The $(S(V_{i,\gamma}), L)$-Verifiers and $(S(V_{i',\gamma'}), L)$-Verifiers ensure the right combination is checked, and the outgoing flow is the same as the incoming flow. Chaining Edge check gadgets makes some $(S(V_{i,\gamma}), L)$-Verifiers redundant, but this does not matter for the argument.

it cannot pass the $(S(V_{i,\gamma}), L)$-Verifier or the $(S(V_{i',\gamma'}), L)$-Verifier at the bottom of the Edge check gadget. Hence, the exit arcs again have flow of values $S(v)$, $L - S(v)$, $S(w)$, and $L - S(w)$ consecutively, just like the entry arcs.

With these gadgets in hand, we now describe the global structure of the reduction. Throughout, it will be more helpful to look at the provided figures than to follow the formal description. For each class $V_{i,\gamma}$, we first create a Vertex selector gadget (as in Figure 6.3).

We then create Edge check gadgets to check, for any $i \in [r]$, that the selected vertices in $V_{i,\gamma}$ for all $\gamma \in [k]$ are adjacent in G. We create k rows of gadgets. Row $\gamma \in [k]$ has $k - \gamma$ Edge check gadgets, which correspond to checking that the vertices selected in $V_{i,\gamma}$ and $V_{i,\gamma'}$ are indeed adjacent (via edges in $E_{i,\gamma}^{i,\gamma'}$) for each $\gamma' \in [\gamma+1, k]$. The construction is as follows (see Figure 6.5). For any $\gamma \in [k], \gamma' \in [\gamma+1, k]$, the Edge check gadget for $E_{i,\gamma}^{i,\gamma'}$ has its left entry arcs unified with the bottom exit arcs of the Edge check gadget for $E_{i,\gamma-1}^{i,\gamma'-1}$ if $\gamma > 1$ and $\gamma' = \gamma + 1$ and with the right exit arcs of the Edge check gadget for $E_{i,\gamma}^{i,\gamma'-1}$ if $\gamma' > \gamma + 1$. The Edge check gadget for $E_{i,\gamma}^{i,\gamma'}$ has its top entry arcs unified with the bottom exit arcs of the Edge check gadget for $E_{i,\gamma-1}^{i,\gamma'}$

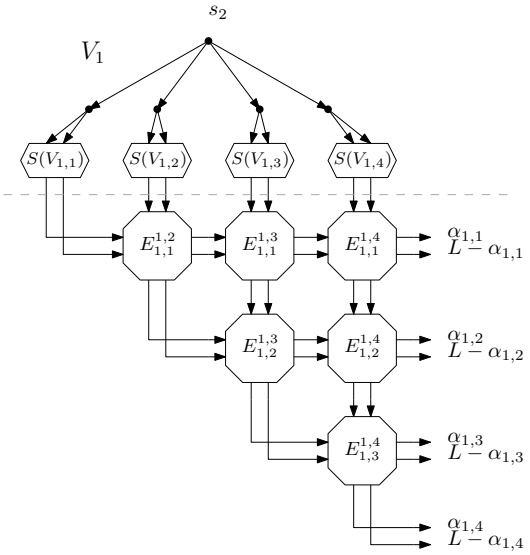


Figure 6.5: Illustration of the structure of Vertex selector and Edge check gadgets to enforce that the selected vertices from V_1 form a clique. In this example, $k = 4$. The $\alpha_{i,\gamma}$'s denote the amount of flow, having a value of $S(v)$ for some $v \in V_{i,\gamma}$, selected in the corresponding $(S(V_{i,\gamma}), L)$-Verifiers of the Vertex selector gadgets.

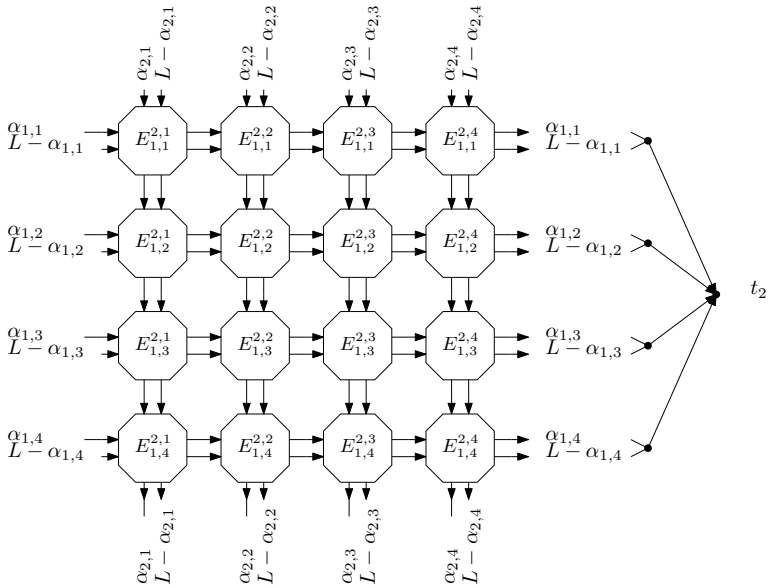


Figure 6.6: Illustration of the structure of the Edge check gadgets to enforce that the selected vertices from V_1 and from V_2 are complete to each other. In this example, $k = 4$. The $\alpha_{i,\gamma}$'s denote the amount of flow, having a value of $S(v)$ for some $v \in V_{i,\gamma}$, selected in the corresponding $(S(V_{i,\gamma}), L)$-Verifiers.

if $\gamma > 1$.

We call this the *Triangle gadget* for V_i. Note that it has $2k$ entry arcs and $2k$ exit arcs. The former can be interpreted to be partitioned into k columns, while the latter can be seen to be partitioned into k rows (see Figure 6.5).

Note that chaining Edge check gadgets makes some $(S(V_{i,\gamma}), L)$-Verifiers redundant, as two such verifiers of the same type will follow each other. However, for the sake of clarity and as it does not matter for the overall argument, we leave such redundancies in place.

Next, we create Edge check gadgets to check, for any $i \in [r-1]$, that the selected vertices in $V_{i,\gamma}$ and $V_{i+1,\gamma'}$ for all $\gamma, \gamma' \in [k]$ are indeed adjacent in G. We create a grid with k rows and k columns of gadgets, where each row corresponds to a combination of γ and γ'. The construction is as follows (see Figure 6.6). For each $\gamma, \gamma' \in [k]$, the Edge check gadget for $E_{i,\gamma}^{i+1,\gamma'}$ has its left entry arcs identified with the right exit arcs of the Edge check gadget for $E_{i,\gamma}^{i+1,\gamma'-1}$ if $\gamma' > 1$ and it has its top entry arcs identified with the bottom exit arcs of the Edge check gadget for $E_{i,\gamma-1}^{i+1,\gamma'}$ if $\gamma > 1$.

We call this the *Square gadget* for V_i and V_{i+1}. Note that it has $4k$ entry arcs and $4k$ exit arcs. The horizontal entry and exit arcs (see Figure 6.6) correspond to V_i, whereas the vertical entry and exit arcs correspond to V_{i+1}.

Then, we connect the Vertex selector and Triangle gadgets. For each V_i, we connect the Vertex selector gadgets for V_i to the Triangle gadget for V_i as follows (see Figure 6.5). The Edge check gadget for $E_{i,1}^{i,2}$ of the Triangle gadget has its left entry arcs unified with the exit arcs of the Vertex selector gadget of $V_{i,1}$. For all $\gamma \in [2,k]$, the Edge check gadget for $E_{i,\gamma-1}^{i,\gamma}$ has its top entry arcs unified with the exit arcs of the Vertex selector gadget of $V_{i,\gamma}$.

Finally, we connect the Triangle and Square gadgets (see Figure 6.7). The Square gadget for V_1 and V_2 has its horizontal entry arcs identified with the exit arcs of the Triangle gadget for V_1. Then, for the Square gadget for V_i and V_{i+1} for any $i \in [r-1]$, it has its vertical entry arcs identified with the exit arcs of the Triangle gadget for V_{i+1} and, if $i \in [r-2]$, it has its vertical exit arcs identified with the horizontal entry arcs of the Square gadget for V_{i+1} and V_{i+2}. Its $2k$ horizontal exit arcs are paired (one pair per color class) and directed to a vertex; these k vertices are then each connected by a single arc of capacity L to t_2 (cf. Figure 6.6). Finally, we also do the latter for the vertical exit arcs of the Square gadget for V_{r-1} and V_r.

We now set the demand for commodity 1 to the sum of the capacities of the outgoing arcs of s_1 (which is equal to the sum of the capacities of the incoming arcs of t_1). We set the demand for commodity 2 to the sum of the capacities of the outgoing arcs of s_2 (which is equal to the sum of the capacities of the incoming arcs of t_2). This completes the construction. We now prove the bound on the pathwidth, followed by the correctness of the reduction and a discussion of the time and space needed to produce it. To prove the pathwidth bound, we note that all constructed gadgets are path pieces, and thus we can apply Lemma 6.3.1.

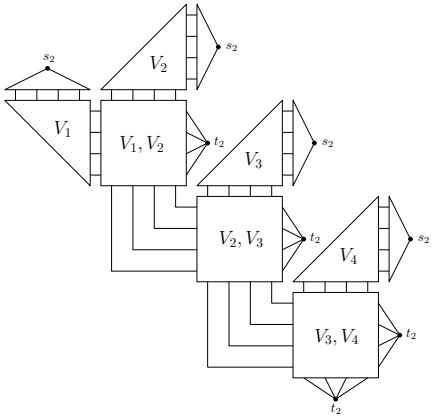


Figure 6.7: Overview of the complete structure of the pathwidth reduction for $r = 4$. Triangles represent a structure as in Figure 6.5, and squares a structure as in Figure 6.6. Directions are not drawn, but clear from Figure 6.5 and 6.6. The labels inside each block (say V_i or V_i, V_{i+1}) denote that flow corresponding to vertices of this set (i.e. V_i or V_i and V_{i+1}) is flowing in a block. Note that all points labelled s_2, t_2 are indeed the same vertex.

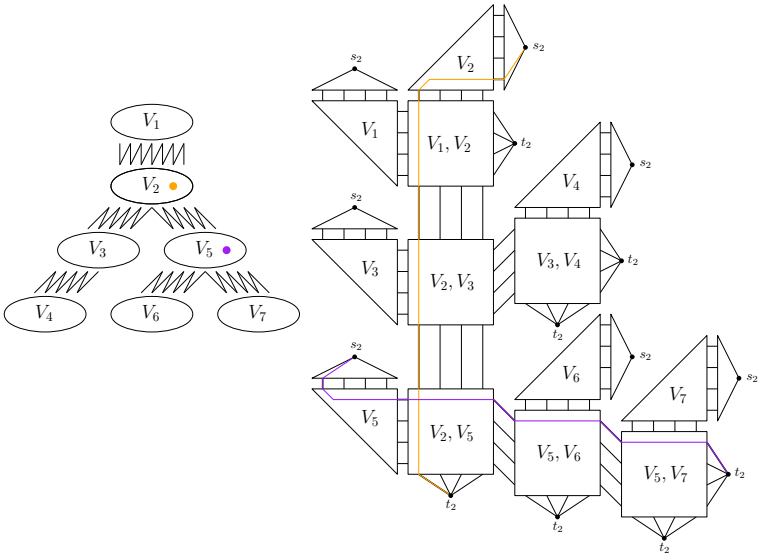


Figure 6.8: Overview of the complete structure of the treewidth reduction for $|I| = 7$. Left: the structure of the input tree partition. Right: the structure of the reduction. Triangles represent a structure as in Figure 6.5, and squares a structure as in Figure 6.6. Directions are not drawn, but clear from Figure 6.5 and 6.6. The labels inside each block (say V_i or V_i, V_{i+1}) denote that flow corresponding to vertices of this set (i.e. V_i or V_i and V_{i+1}) is flowing in a block. Note that all points labelled s_2, t_2 are indeed the same vertex. Flow paths corresponding to a selected vertex in V_2 (orange) and one in V_5 (purple) are drawn as an example.

Claim 6.3.10. *The constructed graph has pathwidth at most* $8k + O(1)$.

Proof. We construct a path decomposition as follows. First, we will ensure that s_1, t_1, s_2, t_2 are in every bag. Then, we make the following observations about the gadgets we construct. Notice that Vertex selector gadgets and Edge check gadgets have pathwidth $O(1)$, which directly follows from Lemma 6.3.1, and that Verifier gadgets have pathwidth $O(1)$, as proven in Lemma 6.3.9. Note that Vertex selector gadgets have two exit arcs, whereas Edge check gadgets have four entry arcs and four exit arcs.

The Triangle gadget is a subgraph of a $k \times k$ grid, while the Square gadget is a $k \times k$ grid. Note that a standard path decomposition of the grid has width k and satisfies the conditions of Lemma 6.3.1. Hence, following Lemma 6.3.1 and the above bounds for the Vertex selector and Edge check gadgets, the pathwidth of the Triangle gadget and the Square gadget is at most $4k + O(1)$. The Triangle gadget has $2k$ entry arcs and $2k$ exit arcs, whereas the Square gadget has $4k$ entry arcs and $4k$ exit arcs.

Finally, the full construction (treating Triangle and Square gadgets as vertices) has a path decomposition of width 2, as it is a caterpillar (see Figure 6.7), with at most two vertices corresponding to Triangle or Square gadgets per bag. Applying Lemma 6.3.1, we obtain a bound of $8k + O(1)$. ∎

We note that a slightly stronger bound of $4k + O(1)$ seems possible with a more refined analysis, but this bound will be sufficient for our purposes.

Claim 6.3.11. *The given* Chained Multicolor Clique *instance has a solution if and only if the constructed instance of* Integer 2-Commodity Flow *has a solution.*

Proof. For the forward direction, assume there exists a chained multicolor clique W in G. We construct a flow. We first consider commodity 2. Recall that one vertex is picked per $V_{i,\gamma}$ class by definition and thus W has size rk. If $v \in V_{i,\gamma} \cap W$ for some $i \in [r], \gamma \in [k]$, then in the Vertex selector gadget of $V_{i,\gamma}$, we send $S(v)$ units of flow of commodity 2 to the left and $L - S(v)$ units of flow to the right into the Verifier gadget (see Figure 6.3). In any Verifier gadget, we route the flow so that it takes the path with capacity equal to the flow (see Lemma 6.3.8). This flow is then routed through all Edge check gadgets of the Triangle and Square gadgets, in the manner presented above in the description of Edge check gadgets. Since W is a chained multicolor clique, the corresponding edge exist in E and the flow can indeed pass through the $(S(E), 2L)$-Verifier gadget of each Edge check gadget (again, see Lemma 6.2). For any $i \in [r-1]$, after passing through the Square gadget for V_i and V_{i+1}, the flow originating in the Vertex selector gadget corresponding to V_i is sent to t_2 through the horizontal exit arcs of the Square gadget (see Figure 6.6). Finally, the flow originating in the Vertex selector gadget corresponding to V_r is sent to t_2 through the vertical exit arcs of the Square gadget. Since W has size equal to rk and we send L units of flow through each Vertex selector gadget, all arcs from s_2 and to t_2 are used to capacity.

Next, we consider commodity 1. All flow of commodity 1 is routed through the unused gates in the Verifier gadgets, which is possible as we only use one vertical

path per gadget for the flow corresponding to a vertex or edge (see Figure 6.2). It follows that we use all arcs from s_1 and to t_1 to capacity.

For the other direction, suppose we have an integer 2-commodity flow f that meets the demands. Hence, there is a 2-commodity flow in the constructed graph with all arcs from s_1 and s_2 and to t_1 and t_2 used to capacity, meaning that their total flow is equal to their total capacity, with flow with the corresponding commodity: commodity 1 for s_1 and t_1, and commodity 2 for s_2 and t_2.

We first reason that the flow of commodity 1 behaves as expected in the Verifier gadgets. Notice that the constructed graph is a directed acyclic graph. Abstractly, it can also be seen as a directed acyclic graph where the vertices are Verifier gadgets. Hence, there exists a topological ordering on the Verifier gadgets. We refer to Figure 6.2 to recall the naming of the vertices u and z of a Verifier gadget. We prove by induction on the topological ordering that the flow of commodity 1 in f that enters over the arc s_1u leaves over the arc zt_1, while staying in the gadget for the entirety of its flow path. As the base case, consider the last Verifier gadget H in the ordering. The bottom of this gadget (refer to Figure 6.2) has arcs only going to t_2, by construction and the fact that it is the last Verifier gadget in the ordering. Hence, flow from $u \in H$ must all go to $z \in H$ and the flow on the arcs s_1u and zt_1 fully fills the capacities. Then, indeed, flow of commodity 1 behaves as in the precondition of Lemma 6.3.8, and this flow does not leave the gadget downwards. For the induction step, consider some Verifier gadget H in the ordering. By the induction hypothesis, all Verifier gadgets later in the ordering have the arcs to t_1 fully filled. But then the flow of commodity 1 in H can only go to $z \in H$ and then on to t_1. We get that the flow on the arcs s_1u and zt_1 in H fully fills the capacities, and does not leave the gadget downwards. Hence, flow of commodity 1 behaves as in the precondition of Lemma 6.3.8. By induction, all flow of commodity 1 behaves 'properly' for Lemma 6.3.8.

By applying Lemma 6.3.8 to every Verifier gadget, we get that the amount flow of commodity 2 always corresponds to some $\alpha \in S$ for the associated set S of the gadget, and the left and right exit arcs carry α and $L-\alpha$ units of flow of commodity 2 respectively. Now, the arc from s_2 in a Vertex selector gadget for $V_{i,\gamma}$ must have L flow of commodity 2 and this must be split in α and $L-\alpha$, with $\alpha \in S(V_{i,\gamma})$. Thus, there is a $v \in V_{i,\gamma}$ with $\alpha = S(v)$, which corresponds to placing v in the chained multicolored clique. In any Edge check gadget, the flow of $\alpha \in S(V_{i,\gamma})$, $L-\alpha$ and $\beta \in S(V_{i',\gamma'})$,$L-\beta$ combines to a unique sum $\alpha+\beta$ and $2L-(\alpha+\beta)$, and assures that the edge between the corresponding vertices is present. As reasoned before, the flow must split back up into α, $L-\alpha$ and $\beta, L-\beta$ by the unique sum due to the fact that S is a Sidon set. We get that the chosen vertices indeed form a chained multicolor clique, as the Edge check gadgets in the Triangle and Square gadgets enforce that all edges between selected vertices are present as should be. ■

Finally, we claim that the constructed graph with its capacities can be built using $O(f(k)+\log n)$ space, for some computable function f. First, Sidon sets can be built in logarithmic space (cf. Theorem 6.2.1). Second, we use the (for log-space reductions standard) technique of not storing intermediate results, but recomputing parts whenever needed. E.g., each time we need the ith number of the Sidon set, we

recompute the set and keep the ith number. Viewing the computation as a recursive program, we have constant recursion depth, while each level uses $O(f(k)+\log n)$ space, for some computable function f. The result now follows. □

To show the XALP-hardness of the Integer 2-Commodity Flow parameterized by treewidth, we reduce from Tree-Chained Multicolor Clique in a similar, but more involved manner.

Theorem 6.1.3. *Integer 2-Commodity Flow with capacities given in unary, parameterized by treewidth, is XALP-complete.*

Proof. Membership in XALP can be seen as follows. Take a tree decomposition of G, say $(\{X_i \mid i \in I\}, T)$; assume that T is binary. One can build a dynamic programming table, where each entry is indexed by a node $j \in I$ with associated bag X_j and a function $f_j^i : X_j \to [-C, C]$, where C is some upper bound on the maximum flow (note that C is linear in the input size), for $i = 1, 2$ and $1 \leq j \leq r$. One should interpret f_j^i as mapping each vertex $v \in X_j$ to the net difference of flow of commodity i in- or outgoing on that vertex in a partial solution in the subtree with bag X_j as root. The content of the table is a boolean representing when there is a partial flow satisfying the requirements that f_j^i sets. Basic application of dynamic programming on (nice) tree decompositions can solve the Integer 2-Commodity Flow problem with this table.

Now, we transform this to a non-deterministic algorithm with a stack as follows. We basically guess an entry of each table, similar to the path decomposition case of Theorem 6.1.1, and use the stack to handle nodes with two children. Recursively, we traverse the tree T. For a leaf bag, guess an entry of the table. For a bag with one child, guess an entry of the table given the guessed entry of the child. For a bag with two children, recursively get a guessed entry of the table of the left child. Put that entry on the stack. Then, recursively get a guessed entry of the table of the right child. Get the top element of the stack, and combine the two guessed entries. We need $O(\log n)$ bits to store the position in T we currently are and to look up information on G. We need $O(f(k)\log C)$ bits to denote a table entry and at each point, we have $O(1)$ such table entries in memory. This shows membership in XALP.

For the hardness, we use a reduction from Tree Chained Multicolor Clique (see Theorem 2.5.7). Suppose we have an instance of Tree Chained Multicolor Clique, with a graph $G = (V, E)$, a tree partition $(\{V_i \mid i \in I\}, T = (I, F))$ with T a tree of maximum degree 3, and a function $c\colon V \to [k]$.

As in the proof of Theorem 6.1.1, we build a Sidon set S of $|V|$ integers in the interval $[4|V|^2]$ using Theorem 6.2.1. To each vertex $v \in V$, we assign a unique element of the set S, denoted by $S(v)$. For any subset $W \subseteq V$, let $S(W) = \{S(v) \mid v \in W\}$ and for any subset $E' \subseteq E$, let $S(E') = \{S(u) + S(v) \mid uv \in E'\}$. Also, let $L = 4|V|^2 + 1$.

We now create a flow network, similar to the network in the proof of Theorem 6.1.1. In particular, we recall the different gadgets that we created in that construction: the Vertex selector gadget (see Figure 6.3), the Edge check gadget (see Figure 6.4), the Triangle gadget (see Figure 6.5), and the Square gadget (see Figure 6.6). Their structure, functionalities, and properties will be exactly the same as before.

Root the tree T at an arbitrary leaf r. We may assume that T has at least two nodes. We now create gadgets in the following manner, essentially mimicking the structure of T (see Figure 6.8). Start with r. Let r' be its child in T. Create a Triangle gadget for V_r and a Square gadget for V_r and $V_{r'}$. Identify the exit arcs of the Triangle gadget with the horizontal entry arcs of the Square gadget.

Now, for every node i of T, starting with r' and traversing the tree down in DFS order, do the following. Let p be the parent of i. Suppose the depth of p is odd. Create a Triangle gadget for V_i. Identify the exit arcs of this new Triangle gadget with the horizontal entry arcs of the (already constructed) Square gadget for V_p and V_i. If i has a child in T, let d be the first child of i in DFS order. Create a Square gadget for V_i and V_d and identify the horizontal exit arcs of the Square gadget of V_p and V_i with the horizontal entry arcs of the Square gadget of V_i and V_d. If i does not have a second child, then the $2k$ horizontal exit arcs of the Square gadget of V_i and V_d are paired (one pair per color class) and directed to a vertex; these k vertices are then each connected by a single arc of capacity L to t_2 (cf. Figure 6.6). If i has a second child d', create a Square gadget for V_i and $V_{d'}$, identify the horizontal exit arcs of the Square gadget of V_i and V_d with the horizontal entry arcs of the Square gadget of V_i and $V_{d'}$. Then the $2k$ horizontal exit arcs of the Square gadget of V_i and $V_{d'}$ are paired (one pair per color class) and directed to a vertex; these k vertices are then each connected by a single arc of capacity L to t_2 (cf. Figure 6.6).

If the depth of p is even, then we do the same as above, but with vertical entry and exit arcs instead of the horizontal ones.

We now set the demand for commodity 1 to the sum of the capacities of the outgoing arcs of s_1 (which is equal to the sum of the capacities of the incoming arcs of t_1). We set the demand for commodity 2 to the sum of the capacities of the outgoing arcs of s_2 (which is equal to the sum of the capacities of the incoming arcs of t_2). This completes the construction. We now prove the bound on the pathwidth, followed by the correctness of the reduction and a discussion of the time and space needed to produce it. To prove the pathwidth bound, we note that all constructed gadgets are pieces, and thus we can apply Lemma 6.3.4.

Claim 6.3.12. *The constructed graph has treewidth at most* $16k + O(1)$.

Proof. We construct a tree decomposition as follows. First, we will ensure that s_1, t_1, s_2, t_2 are in every bag. Following the proof of Claim 6.3.10, the pathwidth (and thus the treewidth) of the Triangle gadget and the Square gadget is at most $4k + O(1)$. Recall that the Triangle gadget has $2k$ entry arcs and $2k$ exit arcs, whereas the Square gadget has $4k$ entry arcs and $4k$ exit arcs. The full construction (treating the Triangle and Square gadgets as vertices) has a tree decomposition of width 1, since it is tree (see also Figure 6.8). Applying Lemma 6.3.4 and Remark 6.3.3, we obtain a bound of $16k + O(1)$. ∎

We note that a slightly stronger bound of $4k + O(1)$ seems possible with a more refined analysis, but this bound will be sufficient for our purposes.

Claim 6.3.13. *The given* Tree Chained Multicolor Clique *instance has a solution if and only if the constructed instance of* Integer 2-Commodity Flow *has a solution.*

Proof. If G is a YES-instance of Tree-Chained Multicolor Clique, then using a similar construction of flows as in Theorem 6.1.1, we can see that the constructed graph has a flow with all arcs from s_1 and s_2 and to t_1 and t_2 are used to capacity. Minor modifications are needed to route the flow corresponding through the tree structure of the Square gadgets in the constructed instance here. Examples of such flow paths are illustrated in Figure 6.8 for the orange and purple vertices.

Conversely, suppose we have an integer 2-commodity flow that meets the demands. Hence, there is a 2-commodity flow in the constructed graph with all arcs from s_1 and s_2 and to t_1 and t_2 used to capacity; that is, their total flow is equal to their total capacity, with flow with the corresponding commodity: commodity 1 for s_1 and t_1, and commodity 2 for s_2 and t_2. Like in Theorem 6.1.1, all flow of commodity 1 does not leave the Verifier gadget it enters, as the constructed graph again is a acyclic. Then, s_2 sends L units of flow of commodity 2 to each Vertex selector gadget. This flow now splits into α and $L-\alpha$, where $\alpha \in S(V_{i,l})$, as Lemma 6.3.8 can be applied. Therefore, there is some $v \in V_{i,l}$ such that $\alpha = S(v)$. We select v into the multicoulored clique. Each Vertex selector gadget in turn sends α and $L-\alpha$ flow respectively through the two input arcs of the edge check gadget incident on it. Since there is a flow passing through each Edge check gadget, we know that there exists an edge between the pair of the selected vertices. From the construction, we then see that the selected vertex form a tree chained multicolor clique. Therefore, G is a YES-instance of Tree-Chained Multicolor Clique. ■

As the constructed graph with its capacities can be built using $O(f(k)+\log|V|)$ space for some computable function f, the result follows. (See also the discussion at the end of the proof of Theorem 6.1.1.) □

Reductions for Undirected Graphs

We now reduce from the case of directed graphs to the case of undirected graphs in a general manner, by modification of a transformation by Even et al. [76, Theorem 4]. In this way, both our hardness results (for parameter pathwidth and for parameter treewidth) can be translated to undirected graphs.

Lemma 6.3.14. *Let G be a directed graph of an Integer 2-Commodity Flow instance with capacities given in unary. Then in logarithmic space, we can construct an equivalent instance of Undirected Integer 2-Commodity Flow with an undirected graph G' with* $\mathrm{pw}(G') \leq \mathrm{pw}(G) + O(1)$, $\mathrm{tw}(G') \leq \mathrm{tw}(G) + O(1)$, *and unit capacities.*

Proof. We consider the transformation given by Even et al. [76, Theorem 4] that shows the NP-completeness of Undirected Integer 2-Commodity Flow and argue that we can modify it to obtain a parameterized transformation from Integer 2-Commodity Flow to Undirected Integer 2-Commodity Flow with capacities given in unary, and with path- or treewidth as parameter. In particular, the transformation increases the path- or treewidth of a graph by at most a constant.

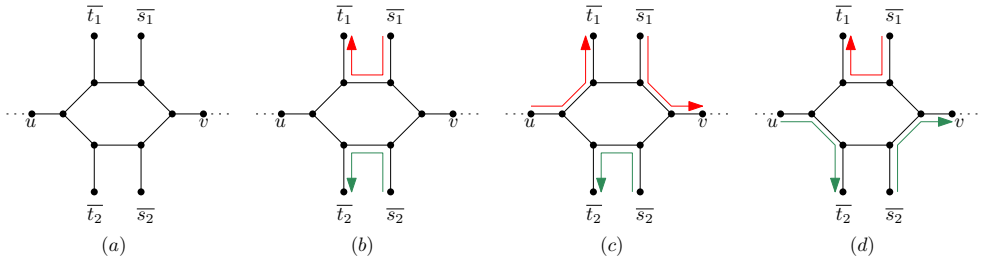


Figure 6.9: The transformation of Lemma 6.3.14 from directed to undirected graphs. Every arc uv with capacity c is replaced by c parallel copies of gadget (a), where $\overline{t_1}, \overline{s_1}, \overline{t_2}, \overline{s_2}$ are the same for every gadget, for all arcs. All capacities are 1. The remaining figures illustrate that the gadget either transports no flow (b), a flow of commodity 1 (c), or a flow of commodity 2 (d).

Given a directed graph $G = (V, E)$, demands d_1 and d_2, and capacity function $c : E \to \mathbb{N}_0$, we construct the instance G', d_1' and d_2', and $c' : E(G') \to \{0, 1\}$ as follows. To the graph G, we add four new vertices $\overline{s_1}, \overline{s_2}, \overline{t_1}, \overline{t_2}$ as new sources and sinks. We connect $\overline{s_i}$ to s_i and $\overline{t_i}$ to t_i by d_i parallel undirected edges of capacity 1, for each $i \in \{1, 2\}$. Next, for each arc $uv \in E$ of capacity p, we create p parallel undirected edges between u and v of capacity 1 each. Then, we replace each of these p undirected edges by the following Diamond gadget. We create a cycle C on six vertices $x_1^{uv}, \ldots, x_6^{uv}$, numbered in cyclic order, which we make adjacent to u, $\overline{t_1}$, $\overline{s_1}$, v, $\overline{s_2}$, and $\overline{t_2}$ respectively by an edge of capacity 1 (see Figure 6.9(a)).

This is the graph G' and c' is as just described. In G', the demands on the two commodities are $d_1' = d_1 + e^*$ and $d_2' = d_2 + e^*$, where e^* is the number of edge gadgets in G' (i.e. the sum of all capacities in c). While G' is technically a multi-graph, any parallel edges e can be subdivided once, and the resulting edges e_1 and e_2 given the same capacity as e. By abuse of notation, we still call this graph G'.

Claim 6.3.15. *The pathwidth of G' is* $\mathrm{pw}(G) + O(1)$ *and the treewidth is* $\mathrm{tw}(G) + O(1)$.

Proof. We note that each of the Diamond gadgets has pathwidth and treewidth 2 and forms a path piece and a tree piece. We add $\overline{s_1}, \overline{s_2}, \overline{t_1}, \overline{t_2}$ as well as s_1, s_2, t_1, t_2 to every bag. Hence, using Lemma 6.3.1 and 6.3.4 and the above description, the claim follows. ∎

Claim 6.3.16. *The demands d_1 and d_2 are met in G if and only if the demands d_1' and d_2' are met in G'.*

Proof. Suppose that the demands in the directed graph G are met by some flow. Then, first we send one unit of flow of each commodity in each edge gadget as shown in Figure 6.9(b). If uv is used to flow one unit of commodity 1 in G, then we change the direction of flow in the edge gadget as in Figure 6.9(c). If one unit of commodity 2 flows through uv, we change the flow through the edge gadget as in Figure 6.9(d). Hence, in addition to the d_1 units of flow of commodity 1 and d_2 units

of flow of commodity 2, e^* units of flow of each type of commodity flows through G'. Therefore, the demands d'_1 and d'_2 are met.

Conversely, suppose that the demands of each commodity are met in the undirected graph G' by some flow. The pattern of flow through each edge gadget could be as in one of the three flows in Figure 6.9. If the flow pattern is as in Figure 6.9(b), then the corresponding flows through the arc uv in G are set as $f^1(uv) = f^2(uv) = 0$. If it is in accordance with Figure 6.9(c), then the corresponding flows in G are set as $f^1(uv) = 1$ and $f^2(uv) = 0$. If the flow pattern is as in Figure 6.9(d), then the corresponding flows in G are set as $f^1(uv) = 0$ and $f^2(uv) = 1$. There are no other options, as every edge incident to any of $\overline{s_1}, \overline{s_2}, \overline{t_1}, \overline{t_2}$ must have 1 unit of flow of that commodity, otherwise the demands cannot be met. Since the capacity of each edge of the Diamond gadget is 1, the three options (b), (c), (d) in Figure 6.9 model exactly the possibilities of sending 1 unit of flow over each edge incident to one of $\overline{s_1}, \overline{s_2}, \overline{t_1}, \overline{t_2}$. Therefore, the flow through G is at least $d_1 + d_2$ and the demands of each commodity are met. ■

The construction can be done in logarithmic space: while scanning G, we can output G'. This completes the proof. □

Theorem 6.1.2. *Undirected Integer 2-Commodity Flow with capacities given in unary, parameterized by pathwidth, is XNLP-complete.*

Proof. The proof of membership in XNLP follows in the same way as the membership of Integer 2-Commodity Flow as described in the proof of Theorem 6.1.1. For the hardness, apply the reduction of Lemma 6.3.14 to the construction of Theorem 6.1.1. □

Theorem 6.1.4. *Undirected Integer 2-Commodity Flow with capacities given in unary, parameterized by treewidth, is XALP-complete.*

Proof. The proof of membership in XALP follows in the same way as the membership of Integer 2-Commodity Flow as described in the proof of Theorem 6.1.3. For the hardness, apply the reduction of Lemma 6.3.14 to the construction of Theorem 6.1.3. □

6.3.2 Binary Capacities

We prove our hardness results for Integer Multicommodity Flow with binary capacities, parameterized by pathwidth. This immediately implies the same results for the parameter treewidth; we do not obtain separate (stronger) results for this case here. Our previous reduction strategy relied heavily on a-Gate gadgets, which have size linear in a, and thus only work in the case a unary representation of the capacities is given.

For the case of binary capacities, we can prove stronger results by reducing from Partition. However, we need a completely new chain of gadgets and constructions. Therefore, we first introduce a number of new gadgets. After that, we give the hardness results for directed graphs, followed by reductions from the directed case to the undirected case.

As before, throughout the section, all constructions will have disjoint sources and sinks for the different commodities. We will set the demands for each commodity equal to the total capacity of the outgoing arcs from the sources, which is equal to the total capacity of the incoming arcs to the sinks. Thus, the flow over such arcs will be equal to their capacity.

In contrast to the previous, our constructions will have two or three commodities. We name the commodities 1, 2, and 3, with sources s_1, s_2, s_3 and sinks t_1, t_2, t_3, respectively. We only need the third commodity for the undirected case.

Gadgets

We define three different types of (directed) gadgets. Since we use binary capacities, our goal is to double flow in an effective manner. For a given integer a, the *a-Doubler gadget* receives a flow and sends out $2a$ flow of the same commodity. This gadget is obtained by combining two other gadgets: the a-Switch and the Doubling a-Switch. The *a-Switch gadget* changes the type of flow; that is, it receives a flow from one commodity, but sends out a flow from the other commodity. The *Doubling a-Switch* is similar, but sends out $2a$ flow. All three types of gadgets have constant size, even in the binary setting.

We now describe the three gadgets in detail.

a-Switch Gadget Let a be any positive integer. The first gadget is called an *a-Switch*. This gadget turns a units of flow of one type of commodity (in the remainder, of commodity 2) into an equal amount of flow of the other commodity (in the remainder, of commodity 1).

The gadget is constructed as follows (see Figure 6.10). We create six vertices $v_1, \ldots, v_6$. We add an entry arc incoming to v_2 (the left entry arc) and an entry arc incoming to v_3 (the right entry arc). We add an exit arc outgoing from v_4 (the bottom exit arc) and an exit arc outgoing from v_5 (the top exit arc). We add arcs along the paths $v_2v_4v_6t_2$, $x_2v_3v_5v_6$, $s_1v_1v_2$, and v_1v_3. All arcs have capacity a. We call v_2, v_3, v_4, v_5 the boundary vertices of the gadget.

We note that, technically, we could also count the arc incoming on v_1 and the arc outgoing from v_6 as entry and exit arcs, but since they are coming from s_1 and going to t_2 respectively, we ignore this aspect.

Lemma 6.3.17. *Consider the a-Switch gadget for some integer a. Let f be some 2-commodity flow such that the arc outgoing from s_1 carries a units of flow commodity 1 and the arc incoming to t_2 carries a units of flow of commodity 2.*

1. *If the left entry arc carries a units of flow of commodity 2 and the right entry arc carries 0 units of flow of commodity 2, then the top exit arc carries a units of flow commodity 1 and the bottom exit arc carries 0 units of flow of commodity 1.*

2. *If left entry arc carries 0 units of flow of commodity 2 and the right entry arc carries a units of flow of commodity 2, then the top exit arc carries 0 units of flow of commodity 1 and the bottom exit arc carries a units of flow of commodity 1.*

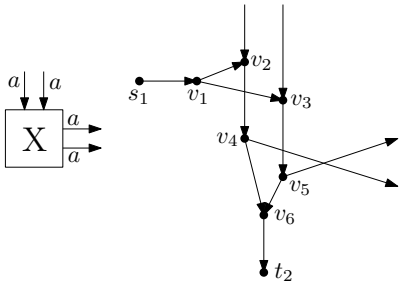


Figure 6.10: The a-Switch gadget. Left: the schematic representation of the gadget, with its entry and exit arcs. Right: the graph that realises the gadget. All arcs have capacity a.

Proof. Suppose the left entry arc carries a units of flow of commodity 2 and the right entry arc carries 0 units of flow of commodity 2. We must send a units of commodity 2 over the path $v_2v_4v_6t_2$, as this is the only way a units of commodity 2 can be sent over the arc v_6t_2 (recall that this arc must be used to capacity and this flow cannot come from s_1). Then all arcs along the path have been used to capacity. By a similar argument, the a units of flow of commodity 1 from s_1 to v_1 must go to v_3, and then via v_5 through the top exit arc.

The other case is symmetric. □

Lemma 6.3.18. *For any integer a, the a-Switch is a path piece such that the required path decomposition (ignoring the sources and sinks) has width 2.*

Proof. The gadget is a piece by construction, with $B^- = \{v_2, v_3\}$ and $B^+ = \{v_4, v_5\}$. To construct the path decomposition, start with a bag containing v_2, v_3. We can now add v_1, and in the next bag remove v_1 and add v_4. Then remove v_2 and add v_5. In the final bag, remove v_3 and add v_6. Each bag contains at most 3 vertices. □

Doubling a-Switch Gadget Let a be any positive integer. The second gadget is called a *Doubling a-Switch*. This gadget turns a units of flow of one type of commodity (in the remainder, of commodity 1) into a $2a$ units of flow of the other commodity (in the remainder, of commodity 2).

The gadget is constructed as follows (see Figure 6.11). We create fourteen vertices $v_1, \ldots, v_{14}$. We add an entry arc incoming to v_4 (the left entry arc) and an entry arc incoming to v_9 (the right entry arc), each of capacity a. We add an exit arc outgoing from v_{13} (the bottom exit arc) and an exit arc outgoing from v_{14} (the top exit arc), each of capacity $2a$. We add arcs with capacity a along the paths $v_4v_5v_6v_7v_8t_1$, $v_9v_{10}v_{11}v_{12}v_8$. We also add arcs v_2v_4, v_2v_6, v_3v_9, v_3v_{11}, v_5v_{13}, v_7v_{13}, $v_{10}v_{14}$ and $v_{12}v_{14}$ with capacity a. Finally, we add the arcs s_2v_1, v_1v_2, v_1v_3 with capacity $2a$. The vertices v_4, v_9, v_{13}, v_{14} are the boundary vertices of the gadget.

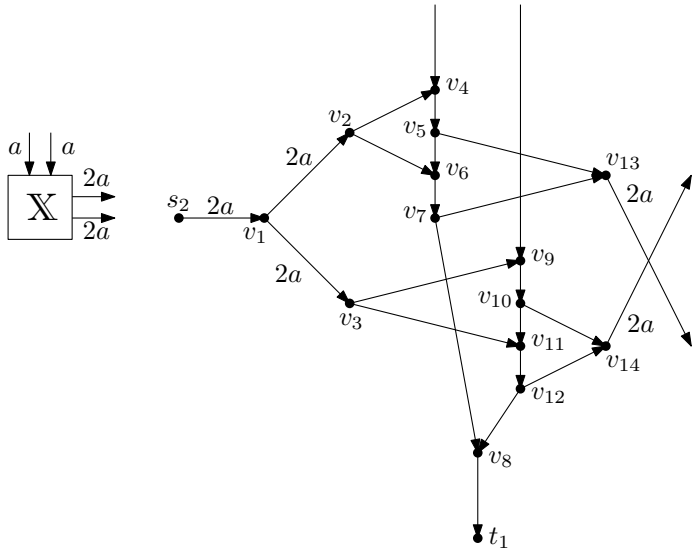


Figure 6.11: The Doubling a-Switch gadget. Left: the schematic representation of the gadget, with its entry and exit arcs. Right: the graph that realises the gadget. The arcs are labelled with their capacities; all unlabelled arcs have capacity a.

Lemma 6.3.19. *Consider the Doubling a-Switch gadget for some integer a. Let f be some 2-commodity flow such that the arc outgoing from s_2 carries $2a$ units of flow of commodity 1 and the arc incoming to t_1 carries a units of flow of commodity 2 .*

1. *If the left entry arc carries a units of flow of commodity 1 and the right entry arc carries 0 units of flow of commodity 1, then the top exit arc carries $2a$ units of flow of commodity 2, and the bottom exit arc carries 0 units of flow of commodity 2.*

2. *If the left entry arc carries 0 units of flow of commodity 1 and the right entry arc carries a units of flow of commodity 1, then the top exit arc carries 0 units of flow of commodity 2 and the bottom exit arc carries $2a$ units of flow of commodity 2.*

Proof. Suppose the left entry arc carries a units of flow of commodity 1 and the right entry arc carries 0 units of flow of commodity 1. We must send a units of flow of commodity 1 over the path $v_4v_5v_6v_7v_8t_1$, as this is the only way a units of commodity 1 can be sent over the arc v_8t_1. Then all arcs along the path have been used to capacity. This implies that the flow from s_2 to v_1 must go to v_3, and then via v_9v_{10} and $v_{11}v_{12}$ to v_{14} after which it must go through the top exit arc.

The other case is symmetric. □

Lemma 6.3.20. *For any integer a, the Doubling a-Switch is a path piece such that the required path decomposition (ignoring sources and sinks) has width 5.*

Proof. The gadget is a piece by construction, with $B^- = \{v_4, v_9\}$ and $B^+ = \{v_{13}, v_{14}\}$. To construct the path decomposition, start with a bag containing v_4, v_9, v_1, v_2, v_3, fol-

lowed by a bag containing v_4, v_9, v_2, v_3. From there, we create bags where we subsequently add v_5, remove v_4, add v_6 and v_{13}, remove v_5, add v_7, remove v_2 and v_6, and add v_8. The bag then contains v_3, v_8, v_9, v_{13}. Then we create bags where we subsequently add v_{10}, remove v_9, add v_{11} and v_{14}, remove v_{10}, and add v_{12}. This forms the required path decomposition. All bags contain at most six vertices. □

***a*-Doubler Gadget** Let a be any positive integer. We can combine an a-Switch gadget with a Doubling a-Switch gadget to get an *a-Doubler* gadget. The first gadget changes the commodity of the flow, where the second gadget changes the commodity back with double the amount of flow.

We construct this gadget as follows (see Figure 6.12). Create an a-Switch gadget and an Doubling a-Switch gadget (refer to Figure 6.10 and 6.11). Identify the top exit arc of the a-Switch gadget with the left entry arc of the Doubling a-Switch gadget. Then identify the bottom exit arc of the a-Switch gadget with the right entry arc of the Doubling a-Switch gadget.

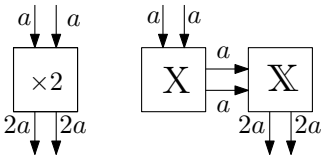


Figure 6.12: The a-Doubler gadget. Left: the schematic representation of the gadget, with its entry and exit arcs. Right: the graph that realises the gadget. The arcs are labelled by their capacities.

Note that the a-Doubler gadget has two entry arcs (the left and right entry arcs) and two exit arcs (the left and right exit arcs), corresponding to the left and right entry arcs of the a-Switch gadget and the top and bottom exit arcs of the Doubling a-Switch gadget respectively.

Lemma 6.3.21. *Consider the a-Doubler gadget for some integer a. Let f be some 2-commodity flow. Then:*

1. *If the left entry arc carries a units of flow of commodity 2 and the right entry arc carries 0 units of flow of commodity 2, then the left exit arc carries $2a$ units of flow of commodity 2 and the right exit arc carries 0 units of flow of commodity 2.*

2. *If the left entry arc carries 0 units of flow of commodity 2 and the right entry arc carries a units of flow of commodity 2, then the left exit arc carries 0 units of flow of commodity 2 and the right exit arc carries $2a$ units of flow of commodity 2.*

Proof. The lemma follows immediately by combining Lemma 6.3.17 and 6.3.19. □

Lemma 6.3.22. *For any integer a, the a-Doubler is a path piece such that the required path decomposition (ignoring sources and sinks) has width* 5.

Proof. Recall from Lemma 6.3.18 that the a-Switch is a path piece of pathwidth at most 2 with two entry arcs and two exit arcs. Recall from Lemma 6.3.20 that the Doubling a-Switch is a path piece of pathwidth at most 5 with two entry arcs and two exit arcs. Note that the structure of the a-Doubler gadget trivially satisfies the preconditions of Lemma 6.3.2. Hence, the lemma then follows by applying Lemma 6.3.2. □

Reduction for Directed Graphs

With the gadgets in hand, we can prove our hardness result for Integer Multicommodity Flow (i.e. the case of directed graphs) for parameter pathwidth.

Theorem 6.1.5. *Integer 2-Commodity Flow with capacities given in binary is NP-complete for graphs of pathwidth at most* 13.

Proof. Membership in NP is trivial. To show NP-hardness, we transform from Partition. Recall that the Partition problem asks, given positive integers $a_1, \ldots, a_n$, to decide if there is a subset $S \subseteq [n]$ with $\sum_{i \in S} a_i = B$, where $B = \sum_{i=1}^{n} a_i/2$. This problem is well known to be NP-complete [114].

So consider an instance of Partition with given integers $a_1, \ldots, a_n$. Create the sources s_1, s_2 and the sinks t_1, t_2. Create two vertices b_1, b_2, both with an arc of capacity B to t_2.

For each a_i, we build a Binary gadget that either sends a_i units of flow to b_1 or a_i units of flow to a vertex b_2, in each case of commodity 2. This will indicate whether or not a_i is in the solution set to the Partition instance. This gadget is constructed as follows (see Figure 6.13 for the case when $a_i = 13$). Consider the binary representation $a_i^p, \ldots, a_i^0$ of a_i. That is, $a_i = \sum_{j=0}^{p} 2^j a_i^j$, with $a_i^j \in \{0,1\}$. For each $j \in [p]$ such that $a_i^j = 1$, we create a column of chained Doubler gadgets. For each $j' < j$, create a $2^{j'}$-Doubler gadget and identify its entry arcs with the exit arcs of the $2^{j'-1}$-Doubler gadget (see Figure 6.13). Then the left exit arc of the (final) 2^{j-1}-Doubler gadget is directed to b_1, while the right exit arc is directed to b_2.

Note that the Binary gadget for a_i still has $2\sum_{j=0}^{p} a_i^j$ open entry arcs (of the 1-Doubler gadget of each column). These naturally partition into $\sum_{j=0}^{p} a_i^j$ left entry arcs and $\sum_{j=0}^{p} a_i^j$ right entry arcs. These all have capacity 1. We now connect these arcs. All further arcs in the construction will have capacity 1.

Create two directed paths P_i^1, P_i^2 of $2\sum_{j=0}^{p} a_i^j$ vertices each (see Figure 6.13). We consider the vertices of each of these paths in consecutive pairs, one pair for each a_i^j that is equal to 1. For each $j \in [p]$ such that $a_i^j = 1$, create a vertex v_i^j with an arc from s_2, an arc to the first vertex of the pair on P_i^1 corresponding to a_i^j, and an arc to the first vertex of the pair on P_i^2 corresponding to a_i^j. Then, add an arc from the second

vertex of the pair on P_i^1 corresponding to a_i^j to the left entry arc of the 1-Doubler gadget of the jth column of gadgets and an arc from the second vertex of the pair on P_i^2 corresponding to a_i^j to the right entry arc of the 1-Doubler gadget of the jth column of gadgets. Finally, create a vertex u_i with an arc to the first vertex of P_i^1 and to the first vertex of P_i^2 and create a vertex w_i with an arc from the last vertex of P_i^1 and the last vertex of P_i^2. This completes the description of the Binary gadget.

We now chain the Binary gadgets. For each $i \in [n-1]$, add an arc from w_i to u_{i+1}. Add an arc from s_1 to u_1 and from w_n to t_1. These arcs all have capacity 1.

We now set the demand for commodity 1 to the sum of the capacities of the outgoing arcs of s_1 (which is equal to the sum of the capacities of the incoming arcs of t_1). We set the demand for commodity 2 to the sum of the capacities of the outgoing arcs of s_2 (which is equal to the sum of the capacities of the incoming arcs of t_2). This completes the construction.

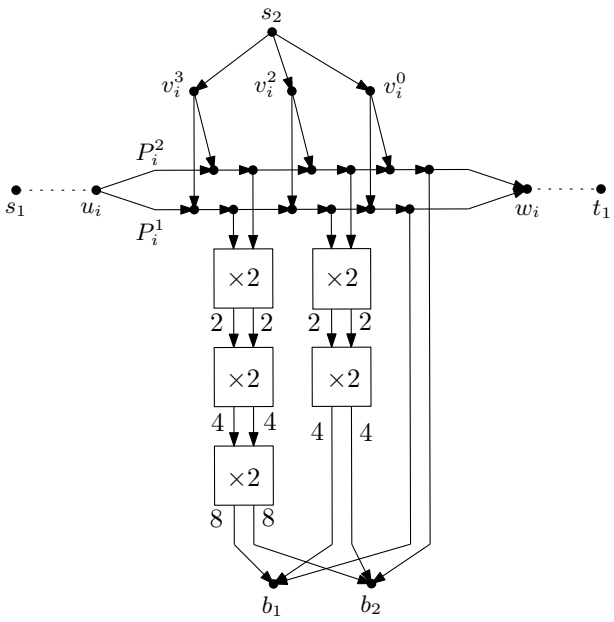


Figure 6.13: Example of the Binary gadget for $a_i = 13$ and its associated paths P_i^1 and P_i^2 and the ends u_i and w_i. Since $13 = 2^3 + 2^2 + 2^0$, we have a column with three Doubler gadgets, a column with two Doubler gadgets, and one with no Doubler gadgets. The vertices v_i^3, v_i^2 and v_i^0 are also drawn. Arcs are labelled by their capacities, but unlabelled arcs have capacity 1. If 1 unit of flow of commodity 1 is sent from s_1 to t_1, then it must pick one of P_i^1, P_i^2 to go through. Hence, the gadget ensures that either 13 units of flow of commodity 2 are sent to b_1 through the left entry and exit arcs of the Doubler gadgets, or 13 units of flow of commodity 2 are sent to b_2 through the left entry and exit arcs of the Doubler gadgets.

Claim 6.3.23. *The constructed graph has pathwidth at most* 13.

Proof. We construct a path decomposition as follows. Add s_1, s_2, t_1, t_2, b_1 and b_2 to every bag. Then, we construct a path decomposition for the Binary gadget and its associated paths P_i^1 and P_i^2. We create the trivial path decompositions for P_i^1 and P_i^2 and union each of their bags, so that we 'move' through the two paths simultaneously. When the first vertex of the pair corresponding to a_i^j (where $a_i^j = 1$) is introduced in a bag, we add a subsequent copy of the bag to which we add v_i^j and another subsequent copy without it. Then, when the second vertex of the pair corresponding to a_i^j (where $a_i^j = 1$) is introduced in a bag, we add bags for the Doubler gadgets of the column corresponding to a_i^j. Since each Doubler gadget has two entry and exit arcs and is a path piece with a path decomposition of width 5 by Lemma 6.3.22, it follows from Lemma 6.3.2 (recalling Remark 6.3.5) that each column has pathwidth 5. Combining this with the other vertices we add to each bag (s_1, s_2, t_1, t_2, b_1 and b_2) and to each bag for each column (both second vertices of the pair corresponding to the column), the total width of the path decomposition is 13. ■

Claim 6.3.24. *The given* Partition *instance has a solution if and only if the constructed instance of* Integer 2-Commodity Flow *has a solution.*

Proof. Let $S \subseteq [n]$ be a solution to the Partition instance. We will find a corresponding solution to the constructed Integer 2-Commodity Flow instance. For each $i \in [n]$, we do the following. If $i \in S$, then we send flow of commodity 2 from s_2 to b_1, through left entry and exit arcs of the Doubler gadgets in the Binary gadget corresponding to a_i. To reach this left side of the Doubler gadgets, the flow passes through vertices and arcs of P_i^1. We can thus send flow of commodity 1 from u_i to w_i via P_i^2. Otherwise, if $i \notin S$, we send flow of commodity 2 from s_2 to b_2, through right entry and exit arcs of the Doubler gadgets in the Binary gadget corresponding to a_i. To reach this right side of the Doubler gadgets, the flow passes through vertices and arcs of P_i^2. We can thus send flow of commodity 1 from u_i to w_i via P_i^1.

Now note that by the properties of the Doubler gadget, proved in Lemma 6.3.21, b_1 will receive a_i units of flow of commodity 2 if $i \in S$ and b_2 will receive a_i units of flow of commodity 2 if $i \notin S$. Since S is a solution to Partition, both b_1 and b_2 receive B units of flow of commodity 2, which they can then pass on to t_2. Moreover, we observe that we can send 1 unit of flow from s_1 to t_1 via the paths P_i^1 and P_i^2, when $i \notin S$ and $i \in S$ respectively.

In the other direction, suppose we have an integer 2-commodity flow that meets the demands. That is, there is a 2-commodity flow in the constructed graph with all arcs from s_1 and s_2 and to t_1 and t_2 used to capacity; that is, their total flow is equal to their total capacity, with flow with the corresponding commodity: commodity 1 for s_1 and t_1, and commodity 2 for s_2 and t_2. Hence, the arc $w_n t_1$ is used to capacity, so 1 unit flow of commodity 1 flows over this arc. Because of the direction of the arcs, this flow can only come from u_n, and so from w_{n-1}. By induction, this flow must come over the arc $s_1 u_1$. We see that the flow of commodity 1 starting at u_1 takes a path which is a union of $P_i^{j_i}$ paths, for $i \in [n]$ and $j_i \in \{1, 2\}$. In particular, this flow does not 'leak' into any Doubler gadget, uses all the arcs $w_i u_{i+1}$ for all $i \in [n-1]$

to capacity, and uses a complete path $P_i^{j_i}$ up to capacity for each $i \in [n]$, $j_i \in \{1,2\}$. Consider the Binary gadget corresponding to a_i. By the above argument, we must have an 1 unit of flow of commodity 1 going through one of the two paths P_i^1 or P_i^2, also using the arc $w_i u_{i+1}$ to capacity. Suppose this is P_i^1. This means that any flow from s_2 to t_2, in this Binary gadget, has to utilise P_i^2, the right side of the Doubler gadgets, and end up at b_2. We can apply Lemma 6.3.21 to every Doubler gadget. As flow of commodity 2 is carried over the right entry and exit arcs and no flow flows over the left entry and exit arcs, the total flow value reaching b_2 has to be equal to a_i. The same argument holds with respect to P_i^2 and b_1. Let $S \subseteq [n]$ be the set of indices i for which the flow of commodity 2 through the Binary gadget corresponding to a_i arrives at b_1. Since the edge $b_1 t_2$ has capacity B and since b_1 has received $\sum_{i \in S} a_i$ units of flow of commodity 2, we find that $\sum_{i \in S} a_i \leq B$. Similarly, $\sum_{i \notin S} a_i \leq B$. Since $\sum_{i \in [n]} a_i = 2B$, we conclude that S is a valid solution to the Partition instance. ■

Finally, as each a-Doubler has constant size, the gadget for a_i has size $O(\log^2(a_i))$, which is polynomial in the input size. Hence, the construction as a whole has size polynomial in the input size. Moreover, it can clearly be computed in polynomial time. □

Reduction for Undirected Graphs

We now reduce from the case of directed graphs to the case of undirected graphs in a general manner. We define a new gadget that is similar to the gadget we used in Section 6.3.1. However, we note that there we required a copies of the gadget if the capacity of an arc is a, which is not feasible in the case of binary capacities. Also note that increasing the capacities of the gadget by Even et al. [76, Theorem 4], here Figure 6.9, invalidates the gadget, as any under-capacity edge would allow flow in the other direction. Hence, we need a different gadget, before we can give our hardness result for undirected graphs.

We first define a subgadget called a *Directed twin flow edge* gadget, seen in Figure 6.14, which we construct as follows. Given two vertices u and v, two commodities i_1 and i_2, and an integer (capacity) c, we add vertices w_1, w_2, w_3, and w_4. We then add edges uw_1, w_1w_2, w_2w_3, w_3w_4, w_4v, $t_{i_1}w_1$, $s_{i_1}w_2$, $t_{i_2}w_3$ and $s_{i_2}w_4$, all with capacity $2c$. We will refer to the vertex u as the *tail vertex* of the gadget and to v as the *head vertex*. Furthermore, we say that the gadget is *labelled* by the two commodities whose source and sink are attached in the gadget.

The gadget only allows flow of the two given commodities through, in only one direction and only in equal amounts. The following lemma captures this behaviour.

Lemma 6.3.25. *Let G be an undirected graph as part of an instance of Undirected Integer ℓ-Commodity Flow for some $\ell \geq 2$. Let $u, v \in V(G)$ and suppose there is a Directed twin flow edge gadget H in G for commodities i_1 and i_2 with u as the tail-vertex of H and v the head-vertex of H. Let f be an ℓ-flow in G that fills all edges incident on each source and sink to capacity with flow of the corresponding commodity. Then:*

- *no flow of any commodity other than i_1 or i_2 can travel from u to v, through H,*

- *the amount of flow of commodities i_1 and i_2 travelling from u to v through H is equal,*
- *no flow can travel through H from v to u.*

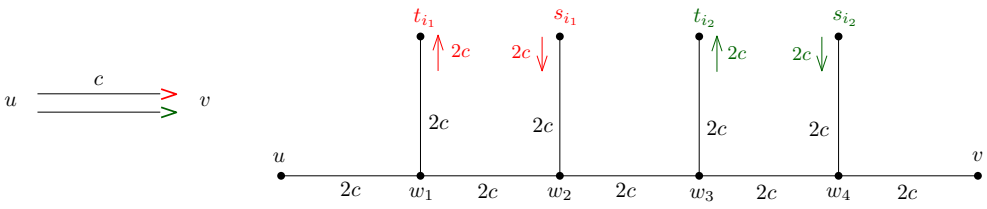


Figure 6.14: The Directed twin flow edge gadget of capacity c. Left: the schematic representation of the gadget. Right: the gadget itself. The arcs are labelled by their capacities. The colors are used to indicate the respective commodities.

Proof. For simplicity, we will assume that the flow f does not go back on itself, i.e. no edge has nonzero flow of the same commodity in both directions.

If in total a units of flow of any commodity other than i_1 or i_2 travel from u to w_3 through H, then by the capacity of w_1w_2 and the fact that $2c$ units of flow of commodity i_1 enter the gadget at w_2, $b \geq a$ units of flow of commodity i_1 must travel from w_2 via w_3 to w_4. Hence, by a similar argument, at least $d \geq a+b$ units of flow of commodity i_2 leave the gadget at w_4. Therefore, the flow of commodity i_2 travelling from u to w_3 must be at least d units. However this implies that $b \geq a+d \geq 2a+b > b$ if $a \neq 0$. We find that $a = 0$.

We then note that if we have a_1 units of flow of commodity i_1 entering the gadget at w_1, we can only have at most a total of a_1 units of the other commodities flowing from w_1 to w_2. Indeed, note that $2c$ units of flow of commodity i_1 must enter the gadget from s_{i_1} to w_2 and leave the gadget via w_1 to t_{i_1}. Hence, $2c - a_1$ units of flow of commodity i_1 would travel from w_2 to w_1. Similarly, if we have a_2 units of flow of commodity i_2 leaving the gadget at w_4, we can only have at most a total of a_2 units of the other commodities flowing from w_3 to w_4. Hence, the amount of flow of commodities i_1 and i_2 travelling from u to v through H is equal.

We also find that the edges w_1w_2 and w_3w_4 are always used to capacity and thus we cannot send any flow from v to u through H, which proves the last item of the lemma. □

We now create a *Directed edge gadget* that effectively functions as a directed edge. To enable this gadget, we need an additional commodity to activate the gadget. So suppose we have $\ell+1$ commodities, with commodity $\ell+1$ being this extra commodity. The gadget is constructed as follows (see Figure 6.15). Given two vertices u and v and an integer (capacity) c, we wish to simulate the arc $e = uv$. Create a new vertex v_e. For each $i \in [\ell]$, we then attach a Directed twin flow edge gadget from u to v_e labelled by commodities i and $\ell+1$, with capacity c. Finally, we add an edge v_ev

with capacity $2c$. We will refer to the vertex u as the *tail vertex* of the gadget, to v as the *head vertex* of the gadget and to v_e as the *central vertex*.

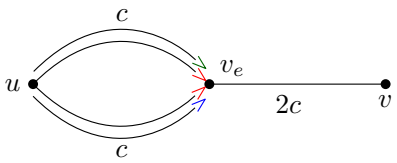


Figure 6.15: Directed edge gadget, for $\ell = 2$ and capacity c.

Lemma 6.3.26. *Let G be an undirected graph that is part of an instance of* Undirected Integer $\ell+1$-Commodity Flow *for some $\ell \geq 2$. Let $u, v \in V(G)$ and suppose there is a Directed edge gadget H in G with u as the tail-vertex of H and v the head-vertex of H. Let f be an $(\ell+1)$-flow in G. Then:*

- *in total, no more than c units of flow of commodities $1,\ldots,\ell$ can travel from u to v, through H,*
- *the amount of flow of commodity $\ell+1$ travelling from u to v through H is equal to the sum of all other commodities travelling from u to v through H,*
- *no flow can travel through H from v to u.*

Proof. Let $H_1,\ldots,H_\ell$ be the Directed twin flow edge gadgets in the Directed edge gadget corresponding to commodities $1,\ldots,\ell$ respectively and let v_e be the central vertex of the gadget. By the last item of Lemma 6.3.25, no flow can travel from v_e to u through any of the H_i and thus no flow can travel from v to u through H.

By the first two items of Lemma 6.3.25, the amount of flow travelling from u to v_e through H_i is equal to some a_i for commodities i and $\ell+1$, and 0 for any other commodities. We find that the amount of flow travelling from u to v_e of commodity $\ell+1$ is equal to the sum the other commodities and thus the amount of flow of commodity $\ell+1$ travelling from u to v through H is equal to the sum of all other commodities travelling from u to v through H.

The first item now holds trivially, since the edge $v_e v$ has capacity $2c$. □

This construction now allows us to extend any reductions from some problem Π to Integer ℓ-Commodity Flow to a reduction from Π to Undirected Integer $\ell+1$-Commodity Flow. We now show such an extension only increases the pathwidth by a constant.

Lemma 6.3.27. *Let G be a directed graph of an* Integer ℓ-Commodity Flow *instance with a path decomposition of width* $\mathrm{pw}(G)$*, such that each bag contains the sources and sinks of commodities $1,\ldots,\ell$. Then in polynomial time, we can construct an equivalent instance of* Undirected Integer $\ell+1$-Commodity Flow *of pathwidth at most* $\mathrm{pw}(G)+5$.

Proof. For each $i \in [\ell]$, let d_i denote the demand for commodity i for the given instance of Integer ℓ-Commodity Flow. To start, we create a source $s_{\ell+1}$ and sink $t_{\ell+1}$

for the extra commodity, with demand $d'_{\ell+1} = \sum_{i=1}^{\ell} d_i$. Then, for $i \in [\ell]$, we connect $s_{\ell+1}$ to s_i by an edge with capacity d_i, and connect t_i to $t_{\ell+1}$ by an edge with capacity d_i. Then, replace each arc uv in G by a Directed edge gadget. Call the resulting graph G'. Set the remaining demands $d'_i = d_i$ for $i \in [\ell]$. This completes the construction.

We note that G' can be constructed in polynomial time from G. The fact that the given instance of Integer ℓ-Commodity Flow and the constructed instance of Undirected Integer $\ell+1$-Commodity Flow are equivalent follows immediately from Lemma 6.3.26 and the construction of G'. Indeed, by the setting of the demand $d'_{\ell+1}$ and the capacities of the edges incident on $s_{\ell+1}$, every source s_i of G receives d_i units of flow of commodity $\ell + 1$. Hence, every flow of commodity i is and can be accompanied by an equal amount of flow of commodity $\ell + 1$. Then, following Lemma 6.3.26, the direction of uv is maintained by the transformation.

To prove the upper bound on the pathwidth of G', consider some arc $e = uv$ in the directed graph G. In the path decomposition of G, there must be some bag X that contains both u and v. Create 3ℓ copies of the bag X that we insert after X. To each of these copies, add the vertex v_e of the Directed edge gadget (this covers in particular the edge $v_e v$ of the gadget). For each commodity $i \in [\ell]$, add to consecutive bags the pairs (w_1, w_2), (w_2, w_3), and (w_3, w_4) of the Directed twin flow edge gadget H_i in the Directed edge gadget. These three bags for each $i \in [\ell]$ handle the path decomposition for the Directed twin flow edge of commodity i. So, in total, the 3ℓ bags handle the path decomposition for the entire gadget.

We do this for every arc in G, where the copies we make of bags are copies only of bags in the path decomposition of G. After we have done this for every arc, we add the source and sink of the extra commodity $\ell + 1$ to every bag. We see that the maximum number of vertices in any bag increases by at most 5. □

By combining Lemma 6.3.27 and Theorem 6.1.5, we obtain the following.

Theorem 6.1.6. *Undirected Integer 3-Commodity Flow with capacities given in binary is NP-complete for graphs of pathwidth at most 18.*

Proof. By Theorem 6.1.5, Integer 2-Commodity Flow with capacities given in binary is NP-complete for graphs of pathwidth at most 13. By inspection of the proof, we see that it has a path decomposition of width 13 such that each bag contains both sources and both sinks. Then the reduction of Lemma 6.3.27 immediately implies the theorem. □

6.3.3 Parameterization by Vertex Cover — At Most One Commodity Per Edge

The last hardness result we will discuss concerns a parameterization by vertex cover. In this case, we will add the additional constraint that the edges must be *monochrome*, that is, for each edge, only one commodity can have positive flow. Later, in Section 6.4.2, we will see that it is possible to approximately solve the problem without this constraint, in polynomial time.

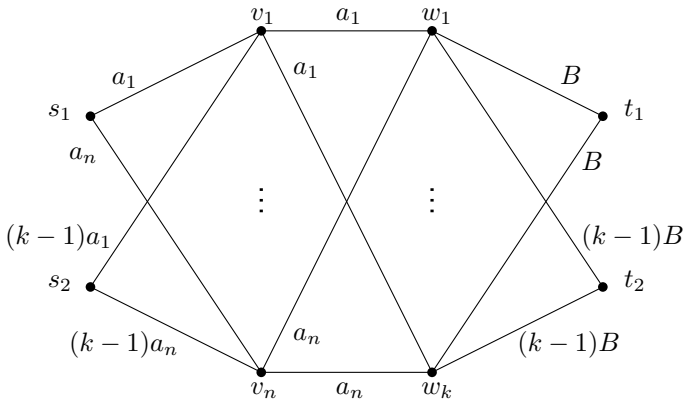


Figure 6.16: Construction for the reduction from Bin Packing to 2-Commodity Flow.

Theorem 6.1.8. *Undirected Integer 2-Commodity Flow with Monochrome Edges is NP-hard for binary weights and vertex cover number 6, and W[1]-hard for unary weights when parameterized by the vertex cover number.*

Proof. We reduce from Bin Packing. Recall that in Bin Packing, we are given integers $a_1, \ldots, a_n$, an integer B, and an integer k. We are asked to decide whether the integers $a_1, \ldots, a_n$ can be partitioned into at most k bins, such that the sum of the numbers assigned to each bin does not exceed B. We note that Subset Sum can be seen as a special case of bin packing by setting $B = (\sum_{j=1}^{n} a_j)/2$ and adding an additional weight $a_0 = B - T$ for T the target value. Subset Sum, and thus Bin Packing for binary weights and $k = 2$, is NP-hard [91]. Bin Packing for unary weights when parameterized by k is W[1]-hard [110]. In both cases, we may assume that $\sum_{j=1}^{n} a_j = kB$: In the unary case, we can add $kB - \sum_{j=1}^{n} a_j$ additional integers that are all equal to 1. In the binary case we have $k = 2$ and we can reduce to the Subset Sum, as mentioned previously.

We first describe our construction. Suppose we are given an instance $((a_j)_1^n, B, k)$ of Bin Packing. We construct an equivalent instance of Undirected Integer 2-Commodity Flow with Monochrome Edges that has a vertex cover of size $k+4$ (see also Figure 6.16). We create a graph G with the following vertices and edges:

$$V(G) = \{s_1, s_2, t_1, t_2, v_1, \ldots v_n, w_1, \ldots, w_k\}$$
$$E(G) = \{s_i v_j : i \in \{1,2\}, j \in [n]\} \cup \{w_i v_j : i \in [k], j \in [n]\} \cup \{w_i t_j : i \in [k], j \in \{1,2\}\}.$$

We set the capacities of the edges as follows:

$$\begin{aligned}
c(s_1 v_j) &= a_j && \forall j \in [n] \\
c(s_2 v_j) &= (k-1)a_j && \forall j \in [n] \\
c(w_i v_j) &= a_j && \forall i \in [k], j \in [n] \\
c(w_i t_1) &= B && \forall i \in [k] \\
c(w_i t_2) &= (k-1)B && \forall i \in [k].
\end{aligned}$$

We set the demands as $d_1 = kB$ and $d_2 = (k-1)kB$. This completes the reduction.

Suppose that the instance of Bin Packing is a YES-instance. Thus, there is a partition $I_1, \ldots, I_k$ of $\{1, \ldots, n\}$ such that $\sum_{j \in I_i} a_j \leq B$ for each $i \in [k]$. As $\sum_{j=1}^{n} a_j = kB$, actually $\sum_{j \in I_i} a_j = B$ for each $i \in [k]$. For each $i \in [k]$ and each $j \in I_i$, route a_j units of flow of commodity 1 from s_1 to v_j to w_i to t_1. Hence, edges incident on s_1 and t_1 only transport flow of commodity 1. For $i \in [k]$, since $\sum_{j \in I_i} a_j = B$, the flow over each edge $w_i t_1$ does not exceed B. Also, route $(k-1)a_j$ units of flow of commodity 2 from s_2 to v_j. Then, split this flow into a_j units of flow to each vertex in $\{w_1, \ldots, w_k\} \setminus \{w_i\}$ and from these vertices to t_2. Hence, edges incident on s_2 and t_2 only transport flow of commodity 2. For $i \in [k]$, $\sum_{i \in [k] \setminus \{i\}} \sum_{j \in I_{i'}} a_j \leq (k-1)B$ since $\sum_{j \in I_i} a_j = B$, and thus the flow over each each edge $w_i t_2$ does not exceed $(k-1)B$. Finally, note that edges between $\{v_1, \ldots, v_n\}$ and $\{w_1, \ldots, w_k\}$ only transport flow of a single commodity, because $I_1, \ldots, I_k$ is a partition of $\{1, \ldots, n\}$. Hence, the instance of Undirected Integer 2-Commodity Flow with Monochrome Edges is a YES-instance.

Conversely, suppose that the instance of Undirected Integer 2-Commodity Flow with Monochrome Edges is a YES-instance and consider a valid flow. We analyse the structure of this flow. The cut $(\{s_1\}, V(G) \setminus \{s_1\})$ has capacity $\sum_{i=1}^{n} a_i = kB = d_1$. Hence, for each $j \in [n]$, we must have a_j units of flow of commodity 1 from s_1 to v_j. Similarly, the cut $(\{s_2\}, V(G) \setminus \{s_2\})$ has capacity $\sum_{i=1}^{n} (k-1)a_i = kB = d_2$. Hence, for each $j \in [n]$, we must have $(k-1)a_j$ units of flow of commodity 2 from s_2 to v_j. Since the total capacity of the remaining edges connected to v_j (the edges $v_j w_i$) is equal to ka_j and since each edge can only be used by one commodity, we find that exactly one of these edges transports a_j units of flow of commodity 1 to some w_i and the other $k-1$ edges each transport a_j units of flow of commodity 2 to the other $w_{i'}$ with $i' \neq i$. Finally, any unit of commodity 1 in w_i must be sent to t_1 and any unit of commodity 2 must be set to t_2, since by construction the total capacity of the cut $(V(G) \setminus \{t_1, t_2\}, \{t_1, t_2\})$ is $k \sum_{j=1}^{n} a_j = d_1 + d_2$, so we cannot have any flow going in the reverse direction.

We now construct sets $I_1, \ldots, I_k$. Now observe that each v_j sends a_j units of flow of commodity 1 to exactly one neighbor w_i. Then add j to I_i. Then $I_1, \ldots, I_k$ forms a partition of $\{1, \ldots, n\}$. Moreover, each w_i can send at most B units of flow of commodity 1 to t_1 and t_1 does not receive units of flow of commodity 1 from vertices outside the set $\{v_1, \ldots, v_n\}$. Hence, $\sum_{j \in I_i} a_j \leq B$ for all $i \in [k]$. Hence, the instance of Bin Packing is a YES-instance.

Finally, note that $|V(G)| = O(n)$, that $V(G) \setminus \{v_1, \ldots, v_n\}$ forms a vertex cover of size $k+4$, and that we can construct the instance in time $n^{O(1)}$. We conclude that since Bin Packing is NP-hard for binary weights and $k = 2$ [91], Undirected Integer 2-Commodity Flow with Monochrome Edges is NP-hard for binary weights and vertex cover size 6. We also conclude that since Bin Packing is W[1]-hard for unary weights when parameterized by k [110], Undirected Integer 2-Commodity Flow with Monochrome Edges is W[1]-hard for unary weights, when parameterized by the vertex cover size. □

The reduction to the directed case can be readily seen.

Theorem 6.1.7. *Integer 2-Commodity Flow with Monochrome Edges is NP-hard for binary weights and vertex cover number* 6*, and W[1]-hard for unary weights when parameterized by the vertex cover number.*

Proof. The proof immediately follows from the proof of Theorem 6.1.8, by directing each edge from left to right (direction as in Figure 6.16). □

6.4 Algorithms

In this section, we complement our hardness results with two algorithmic results.

6.4.1 Parameterization by Weighted Tree Partition Width

We first give an FPT-algorithm for Integer ℓ-Commodity Flow parameterized by weighted tree partition width. This algorithm assumes that a tree partition of the input graph is given. There is an algorithm by Bodlaender et al. [20] that for any graph G and integer k, runs in time $\text{poly}(k) \cdot n^2$ and either outputs a tree partition of G of width $\text{poly}(k)$ or outputs that G has no tree partition of width at most k. By some simple tricks, this can be expanded to approximate weighted tree partition width as well, at the expense of a slightly worse polynomial in k. An approximately optimal tree partition of this form would be sufficient as input to our algorithm.

Theorem 6.1.9. *The Integer ℓ-Commodity Flow problem can be solved in time* $O(2^{2^{\text{tpw}^{3\ell}\text{tpw}}} n)$*, where* tpw *is the breadth of a given weighted tree partition of the input graph.*

Proof. We will describe a dynamic-programming algorithm on a given tree partition $(\mathbb{T}, (B_x)_{x \in V(\mathbb{T})})$. Let $r \in V(\mathbb{T})$ be some node, that we will designate as the root of the tree $\mathbb{T}$. For convenience, we first attach a node to every leaf, with an empty bag.

We will create a table τ, where every entry is indexed by a node x of the tree partition and a collection $\mathbf{f}_x$ of functions f_x^i, one function for every commodity $i \in [\ell]$. We will refer to $\mathbf{f}_x$ as a flow profile and use the superscript i to refer to the flow function for commodity i in the profile. The function

$$f_x^i : B_{p(x)} \to [-\text{tpw}, \text{tpw}],$$

where $p(x)$ is the parent node of x, indicates for every $v \in B_{p(x)}$ the net difference between the amount of flow of commodity i that v receives from (indicated by a positive value) or sends to (indicated by a negative value) the vertices in the bag B_x, in the current partial solution. That is, $f_x^i(v)$ models the value of $\sum_{u \in B_x}(f^i(uv) - f^i(vu))$, where f denotes the current partial solution. Notice that this sum has value in $[-\text{tpw}, \text{tpw}]$, as the sum over all capacities of edges between bags B_x and $B_{p(x)}$ is at most tpw. The content of each table entry will be a boolean that indicates whether there exists a partial flow on the graph considered up to x that is consistent with the indices of the table entry.

We will build the table τ, starting at the leaves of the tree, for which we assumed the corresponding bags to be empty sets, and working towards the root. If x is a

leaf in the tree partition, we set $\tau[x,\{\emptyset,\dots,\emptyset\}] = \text{True}$, where we denote by $\emptyset$, the unique function with the empty set as domain. Otherwise, x is some node with children $y_1,\dots,y_t$. We will group these children y_i in equivalence classes ξ, defined by the equivalence relation $y \sim y'$ if and only if $\tau[y,\mathbf{f}_x] = \tau[y',\mathbf{f}]$ for every flow profile $\mathbf{f}$. Note that there are at most $2^{\text{tpw}^{\ell(2\,\text{tpw}+1)}}$ such equivalence classes, with at most $\text{tpw}^{\ell(2\,\text{tpw}+1)}$ possible flow profiles $\mathbf{f}_{y_j} = (f^1_{y_j},\dots,f^\ell_{y_j})$ for every child y_j of x.

We will now describe an integer linear program that determines the value of $\tau[x,\mathbf{f}_x]$ for a given flow profile $\mathbf{f}_x$. We define a variable $X_{\xi,\mathbf{g}}$ as the number of sets in class ξ whose in- and outflow we choose to match flow profile $\mathbf{g}$[1]. We also define a variable Y^i_e for each edge inside the bag B_x or between B_x and its parent bag, which indicates the flow of commodity i on this edge. We will denote by $N^{in}(v)$ and $N^{out}(v)$ the set of in-neighbors and out-neighbors of v, respectively, restricted to $B_x \cup B_{p(x)}$. We now add constraints for the following properties, for every commodity $i \in [\ell]$. Flow conservation for all vertices v in the bag B_x, that are not a sink/source for commodity i:

$$\sum_{u\in N^{in}(v)} Y_{uv} + \sum_{\xi,\mathbf{g}} X_{\xi,\mathbf{g}} \cdot g^i(v) = \sum_{u\in N^{out}(v)} Y_{uv}.$$

The flow of commodity i from a source s_i (if $s_i \in B_x$):

$$-\sum_{u\in N^{in}(s_i)} Y_{us_i} + \sum_{u\in N^{out}(s_i)} Y_{us_i} - \sum_{\xi,\mathbf{g}} X_{\xi,\mathbf{g}} \cdot g^i(s_i) = d_i$$

The flow of commodity i to a sink t_i (if $t_i \in B_x$):

$$\sum_{u\in N^{in}(t_i)} Y_{ut_i} - \sum_{u\in N^{out}(t_i)} Y_{ut_i} + \sum_{\xi,\mathbf{g}} X_{\xi,\mathbf{g}} \cdot g^i(t_i) = d_i$$

The desired flow to a vertex v in the parent bag:

$$\sum_{u\in N^{in}(v)\setminus B_{p(x)}} Y_{uv} - \sum_{u\in N^{out}(v)\setminus B_{p(x)}} Y_{uv} = f^i_x(v).$$

Edge capacities and non-negative flow:

$$0 \le \sum_{i=1}^{\ell} Y^i_e \le c(e)$$

The number of flow profiles of each type from each class matches the number of bags in that class:

$$\sum_{g:B_x\to[-\text{tpw},\text{tpw}]} X_{\xi,\mathbf{g}} = |\xi|.$$

[1] Throughout the proof, if we sum over pairs $\xi,\mathbf{g}$, we only sum over flow profiles that are valid for bags in ξ. Alternatively, we can set any invalid $X_{\xi,\mathbf{g}}$ to 0 beforehand.

$X_{\xi,\mathbf{g}}$ must be a non-negative integer:

$$X_{\xi,\mathbf{g}} \in \mathbb{N}_0$$

We then use an algorithm of Frank and Tardos [84, Theorem 5.3] to solve the ILP in time $N^{2.5N+o(N)}$, where N is the number of variables in the ILP. This number is dominated by the number of variables $X_{\xi,\mathbf{g}}$, of which there are $O(2^{\mathrm{tpw}^{\ell(2\,\mathrm{tpw}+2)}})$. We thus find a running time of

$$2^{\mathrm{tpw}^{\ell(2\,\mathrm{tpw}+2)}(2.5\cdot 2^{\mathrm{tpw}^{\ell(2\,\mathrm{tpw}+2)}}+o(2^{\mathrm{tpw}^{\ell(2\,\mathrm{tpw}+2)}}))} = 2^{2.5\cdot 2^{\mathrm{tpw}^{\ell(2\,\mathrm{tpw}+2)}}+o(2^{\mathrm{tpw}^{\ell(2\,\mathrm{tpw}+2)}})}.$$

If the ILP has a feasible solution, we set $\tau[x,\mathbf{f}_x] = \text{True}$; otherwise, we set $\tau[x,\mathbf{f}_x] = \text{False}$. We solve $\mathrm{tpw}^{\ell(2\,\mathrm{tpw}+1)}$ such ILP's per bag in the decomposition and thus find a total running time of

$$\mathrm{tpw}^{\ell(2\,\mathrm{tpw}+1)}\, 2^{2.5\cdot 2^{\mathrm{tpw}^{\ell(2\,\mathrm{tpw}+2)}}+o(2^{\mathrm{tpw}^{\ell(2\,\mathrm{tpw}+2)}})} = O(2^{2^{\mathrm{tpw}^{3\ell b}}}).$$

Once we reach the root bag, we use a similar ILP to compute the flow on the root bag, finding a final solution. Since the number of bags is $O(n)$, we find a total running time of $O(2^{2^{\mathrm{tpw}^{3\ell b}}} n)$. □

Note that with some minor changes to the ILP (flow variables can be negative and there is no distinction between in/out edges), this proof also works in the undirected case. Thus, we also find the following result.

Theorem 6.1.10. *The* Undirected Integer ℓ-Commodity Flow *problem can be solved in time* $O(2^{2^{\mathrm{tpw}^{3\ell\,\mathrm{tpw}}}} n)$, *where* tpw *is the breadth of a given weighted tree partition of the input graph.*

Proof. Adjusting the ILP so that flow variables can be negative and there is no distinction between in/out edges, we can see the proof of Theorem 6.1.9 extends to the undirected case. □

6.4.2 Parameterization by Vertex Cover

In this section, we give an approximation algorithm for the Integer 2-Commodity Flow problem, parameterized by the vertex cover number of a graph. Since we consider a decision problem, the use of the term 'approximation' requires some explanation. In our case, we still give a Boolean output, but for the question of whether there is a flow with values in some range around the demands. We make this more precise with the following theorem:

Theorem 6.1.11. *There exists a constant* $C > 0$ *such that the following holds. There is a polynomial-time algorithm that, given an instance of* Integer 2-Commodity Flow *on a graph* G *with demands* d_1, d_2, *either outputs that there is no flow that meets the demands or outputs a 2-commodity flow of value at least* $d_i - C\,\mathrm{vc}^3$ *for each commodity* $i \in [2]$.

Proof. Let X be a vertex cover of the graph G. We can assume that one of size $\mathrm{vc}(G)$ is given as part of the input or we can compute a 2-approximation in polynomial time by the folklore algorithm (attributed to Gavril and to Yannakakis; see e.g. [46]). In the latter case, the proof still holds, but the (hidden) constant is worse by a factor 8.

We first compute, in polynomial time, a solution to the LP relaxation of the problem. This gives us a fractional flow f. If this flow does not meet the demands, then we can immediately answer that there exists no integral flow that meets the demands. We will now argue that we can transform this flow such that the total value remains the same, but the number of arcs that is assigned a non-integral value is $O(|X|^3)$.

We say that an arc e is a *mono-fractional arc* if there is exactly one commodity $i \in [2]$ such that $f^i(e) \notin \mathbb{N}_0$. We say that arc e is a *bi-fractional arc* if for both commodities $i \in [2]$, we have $f^i(e) \notin \mathbb{N}_0$. An arc is *fractional* if it is mono-fractional or bi-fractional. An arc that is not fractional is called *integral*.

We will show that a fractional flow f can be transformed to a flow with the same values for both commodities, such that there are at most $O(|X|^3)$ fractional arcs. The case analysis is tedious, with many cases that are similar but handled slightly differently. In each case, small changes are made to the flow, but all vertices in X will have the same inflow and outflow.

Note that if v is incident to an arc with fractional flow for commodity i, then there must be another arc with v as endpoint with fractional flow for commodity i, due to the flow conservation laws. We can have an incoming and an outgoing arc, two (or more) incoming arcs, or two (or more) outgoing arcs. With arcs that can be mono-fractional or bi-fractional, this gives in total twelve cases. These are illustrated in Figure 6.17.

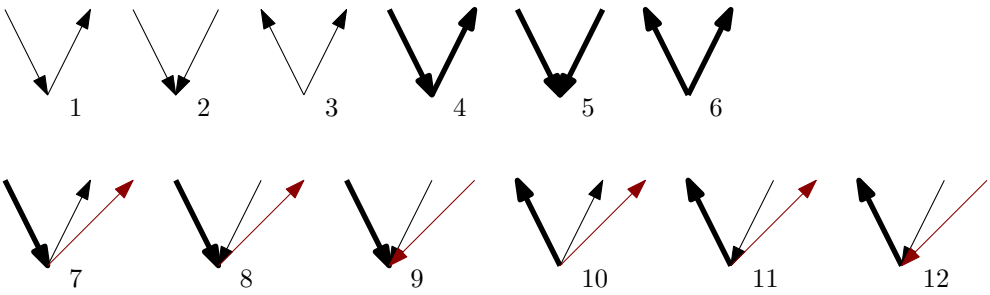


Figure 6.17: Different cases for vertices incident to fractional arcs. Fat edges are bi-fractional; different colored non-fat edges are for different commodities.

Let $Y = V \setminus X$. For vertices $v \in Y$, we distinguish the following cases

1. v has an incoming and an outgoing mono-fractional arc for the same commodity.

2. v has two incoming mono-fractional arcs for the same commodity.

3. v has two outgoing mono-fractional arcs for the same commodity.

4. v has an incoming and an outgoing bi-fractional arc.

5. v has two incoming bi-fractional arcs.

6. v has two outgoing bi-fractional arcs.

7. v has an incoming bi-fractional arc, and for each commodity, an outgoing mono-fractional arc.

8. v has an incoming bi-fractional arc, for one commodity an incoming mono-fractional arc, and for the other commodity an outgoing mono-fractional arc.

9. v has an incoming bi-fractional arc, and for each commodity, an incoming mono-fractional arc.

10. v has an outgoing bi-fractional arc, and for each commodity, an outgoing mono-fractional arc.

11. v has an outgoing bi-fractional arc, for one commodity an incoming mono-fractional arc, and for the other commodity an outgoing mono-fractional arc.

12. v has an outgoing bi-fractional arc, and for each commodity, an incoming mono-fractional arc.

For each of the twelve cases, we have a rule. Each time, we have two 'similar' vertices in Y, and give a transformation of an optimal fractional flow to another optimal fractional flow with fewer fractional values. Some of these rules can be derived by a symmetry argument from earlier rules, which still leaves seven cases, each with a relatively simple proof.

For each arc uv and commodity $i \in [2]$, write the fractional part of this flow as $g^i(uv) = f^i(uv) - \lfloor (f^i(uv) \rfloor$. Recall that all arcs have integral capacities, so a mono-fractional arc has residual capacity, i.e., it is possible to increase the flow over the arc by a positive amount.

For each of the twelve cases, we have a rule that changes an optimal fractional flow to another optimal fractional flow with fewer fractional values. Each time, we increase over some arcs the flow of specified commodities by γ and decrease over some other arcs the flow of those same commodities by γ. Each of the changed values was fractional to start with; in each case, we set γ to the smallest value such that at least one edge hits an integer value.

Claim 6.4.1 (Case 1 Rule)**.** *Let f be an optimal fractional solution to the given instance of* Integer 2-Commodity Flow*. Suppose there are vertices $u, v \in X$, and $y, z \in Y$, with uy, yv, uz, zv mono-fractional for the same resource $i \in [2]$. Then, in polynomial time, we can compute optimal fractional solution g with less fractional arcs than f.*

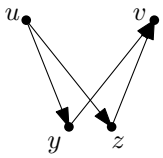


Figure 6.18: The subgraph for the Case 1 Rule

Proof. Recall that all arcs have integral capacities. Write $\gamma = \min\{1-g^i(uy), 1-g^i(yv), g^i(uz), g^i(zv)\}$. Note that we can increase the flow of commodity i over arcs uy and yv by γ and simultaneously decrease the flow of commodity i over arcs uz and zv by γ and obtain a flow with the same values. By doing this, the number of fractional arcs decreases by at least one. ■

Now, when we apply the step of Claim 6.4.1 exhaustively, then we have at most $O(|X|^2)$ vertices where Case 1 applies: if we have more than $2|X|^2$ such vertices, we can find a pair of vertices in X and a commodity for which we can apply the rule from Claim 6.4.1.

For each of the other cases, we can use a similar argument. For Cases 4 – 12, we change the flow for both commodities. For Cases 7 – 12, we have that the resulting rules leave $O(|X|^3)$ vertices of the specific types.

Claim 6.4.2 (Case 2 Rule)**.** *Let f be an optimal fractional solution to the given instance of* Integer 2-Commodity Flow*. Suppose there are vertices $u, v \in X$, and $y, z \in Y$, with uy, uz, vy, vz mono-fractional for the same resource $i \in [2]$. Then, in polynomial time, we can compute optimal fractional solution g with less fractional arcs than f.*

Proof. Set $\gamma = \min\{1 - g^i(uy), g^i(uz), g^i(vy), 1 - g^i(vz)\}$. Increase the flow (for commodity i) over arcs uy and vz by γ, and decrease the flow over arcs uz and vy by γ. All vertices have the same inflow and outflow, so we still have a correct flow, but with a smaller number of fractional arcs. ■

Case 3 is similar to Case 2 but with reversed directions. The proof is almost identical.

Claim 6.4.3 (Case 3 Rule)**.** *Let f be an optimal fractional solution to the given instance of* Integer 2-Commodity Flow*. Suppose there are vertices $u, v \in X$, and $y, z \in Y$, with yu, zu, yv, zv mono-fractional for the same resource $i \in [2]$. Then, in polynomial time, we can compute optimal fractional solution g with less fractional arcs than f.*

The next three rules deal with vertices in Y incident to two bi-fractional edges (Cases 4–6).

Claim 6.4.4 (Case 4 Rule)**.** *Let f be an optimal fractional solution to the given instance of* Integer 2-Commodity Flow*. Suppose there are vertices $u, v \in X$, and $y, z \in Y$, with uy, yv, uz, zv bi-fractional. Then, in polynomial time, we can compute optimal fractional solution g with the same or fewer fractional arcs than f, and less bi-fractional arcs than f.*

Proof. Set $\gamma = \min\{1 - g^1(uy), g^2(uy), g^1(yv), 1 - g^2(yv), g^1(uz), 1 - g^2(uz), 1 - g^1(zv), g^2(zv)\}$. Change the flow as follows: increase by γ the flow of commodity 1 over arcs uy and yv, and the flow of commodity 2 over arcs uz and zv and decrease by γ the flow of commodity 2 over arcs uy and yv, and the flow of commodity 1 over arcs uz and zv. One can check that each arc sends the same amount of total flow, and each vertex receives and sends the same amount of flow of each commodity. The new flow has at least one fewer fractional value. ■

Claim 6.4.5 (Case 5 Rule)**.** *Let f be an optimal fractional solution to the given instance of* Integer 2-Commodity Flow*. Suppose there are vertices $u, v \in X$, and $y, z \in Y$, with uy, vy, uz, vz bi-fractional. Then, in polynomial time, we can compute optimal fractional solution g with the same or fewer fractional arcs than f, and less bi-fractional arcs than f.*

Proof. Set $\gamma = \min\{1 - g^1(uy), g^2(uy), g^1(vy), 1 - g^2(vy), g^1(uz), 1 - g^2(uz), 1 - g^1(vz), g^2(vz)\}$. The flow with less fractional values is obtained by increasing by γ the flow of commodity 1 over arcs uy and vz and of commodity 2 over arcs vy and uz, and decreasing by γ the flow of commodity 2 over arcs uy and vz and of commodity 1 over arcs vy and uz. ■

Using symmetry, we also have the following rule.

Claim 6.4.6 (Case 6 Rule)**.** *Let f be an optimal fractional solution to the given instance of* Integer 2-Commodity Flow*. Suppose there are vertices $u, v \in X$, and $y, z \in Y$, with yu, yv, zu, zv bi-fractional. Then, in polynomial time, we can compute optimal fractional solution g with the same or fewer fractional arcs than f, and less bi-fractional arcs than f.*

Claim 6.4.7. *Let f be an optimal fractional solution to the given instance of* Integer 2-Commodity Flow*. Suppose none of the rules of Cases 1 – 6 applies. There are $O(|X|^2)$ vertices in Y that are incident to at least two mono-fractional arcs for the same commodity, or at least two bi-fractional arcs.*

Proof. If we have $9|X|^2 + 1$ vertices in Y that are incident to at least two mono-fractional arcs for the same commodity, or at least two bi-fractional arcs, then we have either a commodity i with for one of Cases 1 – 3 at least $|X|^2 + 1$ vertices in Y of that case, or for one of the Cases 4 – 6, at least $|X|^2 + 1$ vertices in Y of that case. In each of these situations, we find a pair of vertices in X with two incident vertices in Y where we can apply a rule. ■

Since the neighborhood of any vertex is contained in X, it follows that the number of fractional arcs incident to a vertex of Cases 1 – 6 can be bounded by $O(|X|^3)$. We next give rules that deal with pairs of vertices for Cases 7 – 12.

Claim 6.4.8 (Case 7 Rule)**.** *Let f be an optimal fractional solution to the given instance of* Integer 2-Commodity Flow*. Suppose there are vertices $u, v, w \in X$, and $y, z \in Y$, with uy, uz bi-fractional, yv, zv mono-fractional for commodity 1, and yw, zw mono-fractional for commodity 2. Then, in polynomial time, we can compute optimal fractional solution g with fewer fractional values.*

Proof. Let $\gamma = \min\{1-g^1(uy), g^2(uy), 1-g^1(yv), g^2(yw), g^1(uz), 1-g^2(uz), g^1(zv), 1-g^2(zw)\}$. Increase by γ the flow of commodity 1 over arcs uy and yv, and of commodity 2 over arcs uz and zw; decrease by γ the flow of commodity 2 over arcs uy and yw, and of commodity 1 over arcs uz and zv. Again, we have an optimal flow, but at least one fractional value became integral, without creating new fractional values. ■

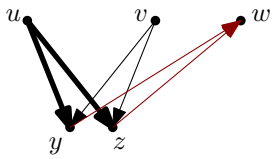


Figure 6.19: The subgraph for the Case 8 Rule

Claim 6.4.9 (Case 8 Rule)**.** *Let f be an optimal fractional solution to the given instance of* Integer 2-Commodity Flow*. Suppose there are vertices $u, v, w \in X$, and $y, z \in Y$, with uy, uz bi-fractional, vy, vz mono-fractional for commodity i, and yw, zw mono-fractional for commodity $3-i$. Then, in polynomial time, we can compute optimal fractional solution g with fewer fractional values.*

Proof. Without loss of generality, assume $i = 1$; otherwise switch the commodities. Let $\gamma = \min\{1-g^1(uy), g^2(uy), g^1(vy), g^2(yw), g^1(uz), 1-g^2(uz), 1-g^1(vz), 1-g^2(zw)\}$. Increase by γ the flow of commodity 1 over arcs uy and vz, and of commodity 2 over arcs uz and zw; decrease by γ the flow of commodity 2 over arcs uy and yw, and of commodity 1 over arcs vy and uz. One can again check that this flow again fulfils all conditions, but we have at least one fewer fractional value. ■

Claim 6.4.10 (Case 9 Rule)**.** *Let f be an optimal fractional solution to the given instance of* Integer 2-Commodity Flow*. Suppose there are vertices $u, v, w \in X$, and $y, z \in Y$, with uy, uz bi-fractional, yv, zv mono-fractional for commodity 1, and yw, zw mono-fractional for commodity 2. Then, in polynomial time, we can compute optimal fractional solution g with fewer fractional values.*

Proof. Let $\gamma = \min\{1-g^1(uy), g^2(uy), 1-g^1(yv), g^2(yw), g^1(uz), 1-g^2(uz), g^1(zv), 1-g^2(zw)\}$. We increase by γ the flow of commodity 1 over arcs uy and yv, and of commodity 2 over arcs uz and zw, and decrease by γ the flow of commodity 2 over arcs uy and yw and of commodity 1 over arcs uz and zv. This gives the desired flow. ■

For Cases 10, 11, and 12, we have similar rules. By reversing all directions of edges, we can observe that these are symmetrical to the rules for Cases 7, 8, and 9. We skip the details here.

Claim 6.4.11. *Let f be an optimal fractional solution to the given instance of* Integer 2-Commodity Flow*. We can find in polynomial time an optimal fractional solution g with $O(|X|^3)$ fractional edges.*

Proof. Apply the rules for Cases 1 – 12 exhaustively, until none apply.

Note that each vertex in Y incident to four or more fractional edges has either at least two mono-fractional edges for the same resource, or at least two bi-fractional edges. By Claim 6.4.7, there are $O(|X|^2)$ such vertices. Also, Claim 6.4.7 shows we have $O(|X|^2)$ vertices incident to two fractional edges. The number of fractional edges incident to these vertices is bounded by $O(|X|^3)$.

It remains to bound the number of vertices that are incident to exactly three fractional edges, and that do not fit in Cases 1–6. Such vertices necessarily belong to one of the cases 7 – 12. Cases 8 and 11 have two subcases, with the roles of the two resources switched. If for any one of these eight cases, there are $|X|^3 + 1$ vertices for which the case applies, then there is a pair of vertices of the same case that has the same neighbors in X among their fractional arcs, and one of the rules can be applied. It follows that there are at most $8|X|^3$ vertices in Y that are incident to exactly three fractional edges, and do not belong to Cases 1–6.

Thus, the total number of fractional edges is bounded by $O(|X|^3)$. ■

Let g be the flow obtained as described above (Claim 6.4.11). We now compute an integer flow h from g as follows.

For each commodity $i \in [2]$, do the following. Take the (standard, 1-commodity) flow network G with source s_i, sink t_i, and for each arc e, capacity $\lfloor g^i(e) \rfloor$, and compute with the Ford-Fulkerson algorithm (or a similar flow algorithm) an optimal $s_i - t_i$ flow. Let the resulting flow be h^i.

We claim that h^1 and h^2 form together the desired integer 2-commodity flow where both commodities differ an additive term of $O(k^3)$ from the optimal flow.

The network for commodity i with capacities $g^i(e)$ has a fractional $s_i - t_i$-flow of optimal value, say α_i, namely the flow g^i. As there are $O(k^3)$ edges with $g^i(e)$ fractional, rounding down these values decreases the total of all capacities by $O(k^3)$, so the network for commodity i with capacities $\lfloor g^i(e) \rfloor$ has an optimal value for fractional flows $\alpha_i - O(k^3)$, but as here, all capacities are integers, this equals the optimal value for integer flows, and a flow with such optimal integer value is found by the Ford-Fulkerson algorithm. The result now follows. □

Note that this algorithm also works for the undirected case, if we use an undirected LP and interpret the directions of the arcs in the various cases as the (net) direction of flow. Thus we also find the following result.

Theorem 6.1.12. *There exists a constant $C > 0$ such that the following holds. There is a polynomial-time algorithm that, given an instance of* Undirected Integer 2-Commodity Flow *on a graph G with demands d_1, d_2, either outputs that there is no flow that meets the demands or outputs a 2-commodity flow of value at least $d_i - C\,\mathrm{vc}^3$ for commodity $i \in [2]$.*

Proof. We adjust the algorithm of Theorem 6.1.11 to use an undirected LP and interpret the directions of the arcs in the various cases as the (net) direction of flow. Then, the result holds for undirected graphs. □

6.5 Conclusion

In this chapter we have discussed the complexity of the Integer Multicommodity Flow problem (see figure 6.2). We have considered the parameters pathwidth, treewidth, weighted tree partition width and vertex cover. We found different hardness results depending on whether the capacities in the instance are given in unary of binary. We find (nearly) the same hardness results for directed and undirected graphs.

Parameter	unary capacities	binary capacities
pathwidth	XNLP-complete	para-NP-complete
treewidth	XALP-complete	para-NP-complete
weighted tree partition width	FPT (1)	FPT (1)
vertex cover	(2); in XP	(2); open

Table 6.2: Overview of our results for Integer 2-Commodity Flow. para-NP-complete means NP-complete for fixed value of parameter. (1) Capacities of arcs inside bags can be arbitrary, capacities of arcs between bags are bounded by weighted tree partition width. (2) Approximation, see Theorem 6.1.11; conjectured in FPT. For the undirected case, the same results hold, except that for the para-NP-completeness for the parameters pathwidth and treewidth, we need a third commodity.

The case of unary capacities, with the parameter pathwidth (treewidth) gives an interesting example of XNLP-completeness (XALP-completeness), even when there are only two commodities. The XNLP- and XALP-completeness imply that XP algorithms for the problems are unlikely also to only use $O(f(k)n^{O(1)})$ space by the Slicewise Polynomial Space Conjecture (see Conjecture 2.5.5). Moreover, the XNLP- and XALP-completeness results imply that the problems are W[t]-hard via Lemma 2.5.6. In Chapter 7 we will further examine the class #XLP (#XALP), which is the counting equivalent of XNLP (XALP) and thus forms a bridge between the themes of this chapter and those of Part II.

We end the chapter with some open problems. A number of cases for undirected graphs remain unresolved. We conjecture that for several such cases, the complexity results will be analogous to the directed case. A notable open case is Undirected Integer 2-Commodity Flow, which we conjecture is NP-complete for graphs with a pathwidth bound, but Theorem 6.1.6 only gives the result with three commodities.

We also conjecture that Integer 2-Commodity Flow is fixed-parameter tractable with the vertex cover number as parameter, possibly by using a dynamic programming algorithm that only needs to investigate solutions that are 'close' to the approximate solution found by Theorem 6.1.11.

Finally, we believe that the problem may be interesting to investigate on certain graph classes, for example planar graphs of bounded treewidth or in general on graphs of treewidth or pathwidth below the bounds given by our hardness results.

7 Problems Complete for #XLP and #XALP

Not everything that can be counted counts, and not everything that counts can be counted.

William Bruce Cameron

7.1 Introduction

In this chapter we combine one of the themes of Part II with the theme of this part, by considering the classes #XLP and #XALP. These classes can be thought of as the counting equivalents of XNLP and XALP respectively. As is the case with XNLP and XALP, #XLP and #XALP form the natural homes for some problems for which previously only hardness was known and not completeness. One can typically expect the counting version of an XNLP-complete problem to be #XLP-complete, although the reverse does not always need to be the case.

It is worth noting that Bodlaender et al. [19] studied a closely related class called XLPP, which relates to XNLP and #XLP in the same way as PP relates to NP and #P.

One of the new insights in this chapter is the introduction of the notion of an adorned DAG and the associated problems of #Path-Like SAD Branchings and #SAD Branchings (see Section 7.3). These problems are in some sense a rephrasing of the definitions of #XLP and #XALP, but in a more manageable form. This makes them a very convenient starting point for our chain of reductions.

The remaining results in this chapter can loosely be grouped into three categories: multicolored pattern problems, satisfiability-like problems and Holant problems. Of the latter we only consider one instance, namely the #AntiFactor$_1$ problem (see Section 7.6). Other problems, like the #Perfect Matchings problem[1] also fall under this framework and could potentially be of interest in the context of #XLP and #XALP. We mostly use the satisfiability problems as intermediate problems and will not spend too much time discussing them.

The final category deserves a more extensive introduction, as it contains a large part of the technical contributions of this work. Over the last decades, several classes of counting and decision problems concerning pattern matching for fixed graph classes $\mathcal{H}$ were studied. These problems have the following form: Given a graph $H \in \mathcal{H}$ from the fixed class $\mathcal{H}$ and a general graph G with no restrictions, decide

[1] The #Perfect Matchings problem asks for the number of subsets of the edges, that hit every vertex exactly once.

whether H "occurs" in G, according to some formal criterion of occurrence. As main examples, the problems $\mathrm{Hom}(\mathcal{H})$, $\mathrm{Sub}(\mathcal{H})$, and $\mathrm{Ind}(\mathcal{H})$ respectively ask to determine the existence of a homomorphism from H to G, a subgraph of G isomorphic to H, or an induced subgraph. Variants of the problems for counting and counting modulo fixed primes have also been studied, also in the presence of colors. In the latter setting, the case that we will consider is that of multicolored patterns, where each vertex of the pattern has a unique color.

These problems have been studied under various complexity assumptions. Firstly, it is known that assuming P ≠ NP does not suffice for full classifications [43, Corollary 10]. Under FPT ≠ W[1] however, full classifications for the induced subgraph and homomorphism decision problem are known when parameterizing by pattern size [43]. The subgraph decision problem remains open, however, a full classification is known for the counting version [54] and other related problems called *graph motif parameters* [52].

Very recently, Curticapean [51] classified the counting versions of $\mathrm{Hom}(\mathcal{H})$, $\mathrm{Sub}(\mathcal{H})$, and $\mathrm{Ind}(\mathcal{H})$ under FP ≠ #P, an *a priori* weaker assumption than P ≠ NP, under the provision that $\mathcal{H}$ comes with a polynomial-time algorithm that enumerates sufficiently complicated patterns from $\mathcal{H}$. More concretely, if a graph class $\mathcal{H}$ contains n-vertex graphs with $f(n) \times f(n)$ grids as subgraphs, for *any* unbounded function f, then it is known [59] that the multicolored homomorphism decision and counting problems are W[1]- and #W[1]-hard. If f is even lower-bounded by a power function, i.e. $f(n) = \Omega(n^\alpha)$ for $\alpha > 0$, then it was shown [51] that the problem is #P-hard.

We will see that the complexity class #XLP allows us to investigate an intermediate setting, where the grid is of size $f(n) \times p(n)$ for some power function $p(n) = \Omega(n^\alpha)$ and an unbounded function f, i.e., slim yet long. The known parameterized reductions show that such problems are #W[1]-hard, while the #P-hardness proofs do not kick in for such patterns. Still, one is led to believe that W[1] is not the right upper bound for such problems, as the pattern size is not bounded by the parameter. Indeed, we can find an appropriate home for the slim-but-long grids in the class #P. To this end, we adapt a reduction by Curticapean [51] that was used to prove #P-hardness into the setting of #XLP.

7.1.1 Results

We introduce the complexity classes #XLP and #XALP. We show completeness for a number of counting problems. As a starting point, we introduce a pair of new problems called #SAD Branchings and #Path-Like SAD Branchings, which will function as both canonically hard problems and a convenient way to prove membership. See Figure 7.1 for a schematic overview of our reductions.

Theorem 7.1.1.

a. *#SAD Branchings is #XALP-complete.*

b. *#Path-Like SAD Branchings is #XLP-complete.*

Based on this result we show completeness for a number of other problems, using a combination of existing reductions and new or altered reductions. The first pair of

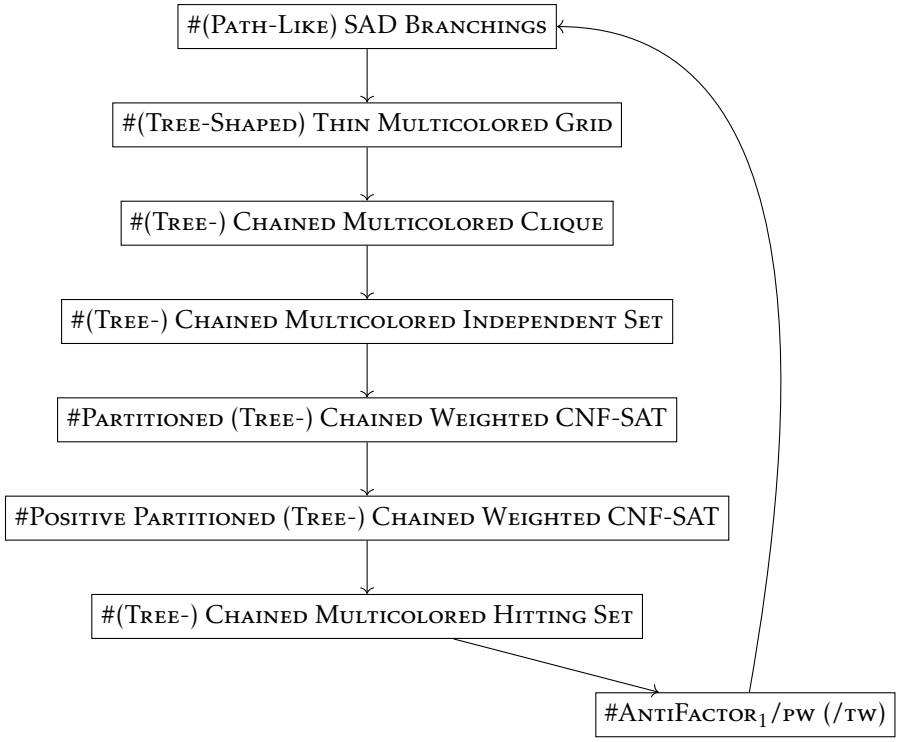


Figure 7.1: The reduction tree of our various results. Each arrow points towards the problem we reduce to.

problems is that of #Thin Multicolored Grids and #Tree-Shaped Thin Multicolored Grids.

Theorem 7.1.2.

a. *#Thin Multicolored Grids is #XLP-complete.*

b. *#Tree-Shaped Thin Multicolored Grids is #XALP-complete.*

From this we can easily show hardness for #Chained Multicolored Clique and #Tree-Chained Multicolored Clique, along with the closely related problems of #Chained Multicolored Independent Set and #Tree-Chained Multicolored Independent Set.

Theorem 7.1.3.

a. *#Chained Multicolored Clique and #Chained Multicolored Independent Set are #XLP-complete.*

b. *#Tree-Chained Multicolored Clique and #Tree-Chained Multicolored Independent Set are #XALP-complete.*

After that we turn our attention to #Partitioned Chained Weighted CNF-SAT and #Partitioned Tree-Chained Weighted CNF-SAT. We show hardness for these problems via a reduction from #Chained Multicolored Independent Set and #Tree-Chained Multicolored Independent Set.

Theorem 7.1.4.

a. *#Partitioned Chained Weighted CNF-SAT is #XLP-complete.*

b. *#Partitioned Tree-Chained Weighted CNF-SAT is #XALP-complete.*

We can trivially extend the results of Theorem 7.1.4 to #Chained Multicolored Hitting Set, #Positive Partitioned Chained Weighted CNF-SAT and their tree-chained variants.

Theorem 7.1.5.

a. *#Positive Partitioned Chained Weighted CNF-SAT and #Chained Multicolored Hitting Set are #XLP-complete.*

b. *#Positive Partitioned Tree-Chained Weighted CNF-SAT and #Tree-Chained Multicolored Hitting Set are #XALP-complete.*

From this, the final result follows by a slight variation on an existing reduction by Marx et al. [128].

Theorem 7.1.6.

a. *#AntiFactor$_1$/pw is #XLP-complete.*

b. *#AntiFactor$_1$/tw is #XALP-complete.*

Membership for #AntiFactor$_1$/pw and #AntiFactor$_1$/tw follow from a reduction back to #Path-Like SAD Branchings and #SAD Branchings respectively. These membership results then propagate through the existing chains of reduction to provide the membership results in Theorems 7.1.2 to 7.1.5.

7.2 Preliminaries

7.2.1 Turing Machines.

In previous chapters we were able to refer to Turing machines, without needing to examine them in too much detail. However, in this chapter we give new complexity classes and thus need to show completeness for a problem, directly from the definition of the class. This requires us to work directly with the inner workings of a Turing machine and as such we now give a more detailed description.

Throughout, we let $\llcorner$ denote a special character, called the *blank* character. A *k-tape Turing machine* is a tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$, where

- Q is a finite set of *states*,
- Σ is a finite set with $\llcorner \notin \Sigma$ called the *input alphabet*,
- $\Gamma \supseteq \{\llcorner\} \cup \Sigma$ is the *tape alphabet*,

- $q_0, q_{\text{acc}}, q_{\text{rej}} \in Q$ are the starting, accepting, and rejecting states, respectively,
- and $\delta\colon Q \times \Gamma^k \to 2^{Q\times(\Gamma\times\{-1,0,1\})^k}$ is the *transition function*.

A *configuration* of M is a tuple in $Q \times (\Gamma^* \times \mathbb{N})^k$. We say that a configuration $C_1 = (q_1, (x_{1,1}, p_{1,1}), \ldots, (x_{1,k}, p_{1,k}))$ *yields* a configuration $C_2 = (q_2, (x_{2,1}, p_{2,1}), \ldots, (x_{2,k}, p_{2,k}))$ if the following holds. For all $i \in [k]$, let $a_i = x_{1,i}(p_{1,i})$, i.e. the $p_{1,i}$-th element in string $x_{1,i}$. There exists $(q_2, (b_1, D_1), \ldots, (b_k, D_k)) \in \delta(q_1, a_1, \ldots, a_k)$ such that:

- For all $i \in [k]$: $p_{2,i} = p_{1,i} + D_i$.
- For all $i \in [k]$ and all $j \neq p_{1,i}$: $x_{2,i}(j) = x_{1,i}(j)$; and $x_{2,i}(p_{1,i}) = b_i$.

Let $x \in \Sigma^*$. The *starting configuration of M on input x* is the configuration $(q_0, (x, 0), (\text{␣}, 0), \ldots, (\text{␣}, 0))$. Any configuration $(q_{\text{acc}}, \ldots)$ is called an *accepting configuration* and every configuration $(q_{\text{rej}}, \ldots)$ is called a *rejecting configuration*. A configuration that is either accepting or rejecting is called a *final configuration*. For a set $\mathcal{C}$ of configurations of M and $i \in [k]$, the *space usage of M on its i-th tape in $\mathcal{C}$* is $\max_{C\in\mathcal{C}}\{p_i \mid C = (q, (x_1, p_1), \ldots, (x_k, p_k)\}$.

We say that *M has read-only access to its i-th tape* if for all $(q, a_1, \ldots, a_k) \in Q \times \Gamma^k$ and all $(q', (b_1, D_1), \ldots, (b_k, D_k)) \in \delta(q, a_1, \ldots, a_k)$, we have that $a_i = b_i$. We say that *M has write-only access to its i-th tape* if for all $(q, a_1, \ldots, a_k) \in Q \times \Gamma^k$ and all $(q', (b_1, D_1), \ldots, (b_k, D_k)) \in \delta(q, a_1, \ldots, a_k)$, we have that $a_i = \text{␣}$, $b_i \in \Sigma$, and $D_i = 1$.

A *non-deterministic k-tape Turing machine* is a k-tape Turing machine M, where we say that *M accepts a string $x \in \Sigma^*$* (where Σ is the input alphabet) if there is a sequence of configurations $C_0, C_1, \ldots, C_m$ such that C_0 is the starting configuration of M on input x, C_m is an accepting configuration, and for all $i \in [m]$, C_{i-1} yields C_i.

A *deterministic k-tape Turing machine* is a non-deterministic k-tape Turing machine such that $|\delta(q, a_1, \ldots, a_k)| = 1$ for all $q \in Q$ and $a_1, \ldots, a_k \in \Gamma$.

An *alternating k-tape Turing machine* is a tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}, \kappa)$ where $M' = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ is a k-tape Turing machine and $\kappa\colon Q \setminus \{q_{\text{acc}}, q_{\text{rej}}\} \to \{\vee, \wedge\}$ labels each non-final state as *non-deterministic* ($\vee$) or *co-non-deterministic* ($\wedge$). A configuration of M is a configuration of M'. For $x \in \Sigma^*$ a *computation tree of M on input x* is a rooted tree T whose vertices are configurations of M such that: (1) The root of T is the starting configuration of M on input x. (2) Each $C = (q, \ldots) \in V(T)$ such that $\kappa(q) = \vee$ has precisely one child C' in T where C yields C' in M. (3) Each $C = (q, \ldots) \in V(T)$ such that $\kappa(q) = \wedge$ has all configurations C' such that C yields C' in M as children in T. We say that *M accepts a string $x \in \Sigma^*$* if there is a computation tree of M on input x such that all leaves are accepting configurations.

7.2.2 #XLP and #XALP

We now recall the following definitions from Chapter 2 for #XLP and #XALP and the associated Turing machines. These classes form the central objects of this chapter.

Definition 2.5.1. *An* XNLP machine *is a non-deterministic Turing machine such that for a given parameterized instance (I, k), where $|I| = n$:*

(i) For a Yes *instance, at least one computation path accepts.*

(ii) For a No *instance, all computation paths reject.*

(iii) There is some computable function f and some constant c such that the machine runs in at most $f(k)n^c$ time and uses at most $f(k)\log n$ space.

(iv) The machine has read-only access to an input tape of size n and write-only access to an output tape.

Definition 2.5.2. *An* XALP machine *is an alternating Turing machine M such that for each instance (I,k), where $|I| = n$, items items i, ii and iv of Definition 2.5.1 are satisfied (where computation trees take the role of computation paths), and:*

(iii') There is some computable function f and some constant c such that M uses at most $f(k)\log n$ space and such that the computation tree of M on input (I,k) has size at most $f(k)n^c$.

Using these notions we can define the classes #XLP and #XALP.

Definition 2.5.8.

(i) The class #XLP *consists of all problems that ask for the number of accepting paths of some XNLP machine.*

(ii) The class #XALP *consists of all problems that ask for the number of accepting trees in some XALP machine.*

7.2.3 Polynomial Interpolation in Logspace

In previous chapters we have used interpolation to retrieve the coefficients of a univariate polynomial from a collection of evaluations. For a multivariate polynomial $p \in \mathbb{Z}[x_1,\dots,x_s]$ of maximum degree d this is still possible, but requires points to be sufficiently spread out in each direction. For example, we can compute the $(d+1)^s$ coefficients of p from its evaluations on the grid $[d+1]^s$: This can be achieved by solving a linear system of $(d+1)^s$ equations with a Vandermonde system matrix of full rank. However, this method *a priori* requires super-logarithmic space, which will prove to be a problem for the applications in this chapter. Luckily, for our purposes, rather than computing the coefficients of p, it suffices to evaluate $p(\xi)$ at an arbitrary point $\xi \in \mathbb{Q}^s$. This can be achieved with Lagrange interpolation in logarithmic space. Let us define

$$L_{(j_1,\dots,j_s)}(x_1,\dots,x_s) = \prod_{\kappa\in[s]} \prod_{\substack{i\in[d+1]\\ i\neq j_\kappa}} \frac{x_\kappa - i}{j_\kappa - i}$$

and observe that for $\mathbf{j},\mathbf{k} \in [d+1]^s$ we have

$$L_{\mathbf{j}}(\mathbf{k}) = \begin{cases} 1 & \text{if } \mathbf{j} = \mathbf{k}, \\ 0 & \text{otherwise.} \end{cases}$$

We thus have

$$p(x_1,\ldots,x_s) = \sum_{\mathbf{j}\in[d+1]^s} p(\mathbf{j})L_{\mathbf{j}}(x_1,\ldots,x_s)$$

and conclude the following useful lemma (also mentioned in Chapter 2).

Lemma 2.7.2. *Let $R \subseteq \mathbb{Q}$ be a finite set. Let $p \in \mathbb{Z}[x_1,\ldots,x_s]$ for $s = O(1)$ be a polynomial of maximum degree d. If there is a $d^{O(1)}$ time- and $O(\log d)$ space-bounded algorithm that computes $p(\mathbf{j})$ on input $\mathbf{j} \in [d+1]^s$, then there also exists such an algorithm for computing $p(\xi)$ at any input $\xi \in R^s$.*

7.2.4 Holant Problems

In Section 7.6 we use a framework for graph problems called the *Holant Framework*. Holant problems are a class of problems, introduced by Valiant [152], defined as follows.

Definition 7.2.1. *A* signature graph *is an edge-weighted graph Ω, with at each vertex $v \in V(\Omega)$ a* signature $f_v : \{0,1\}^{\delta(v)} \to \mathbb{Q}$.

The Holant *of Ω is defined as follows.*

$$\text{Holant}(\Omega) := \sum_{x\in\{0,1\}^{E(\omega)}} \left(\prod_{\substack{e\in E(\Omega)\\ x(e)=1}} w(e)\right)\left(\prod_{v\in V(\Omega)} f_v(x|_{\delta(v)})\right).$$

Many graph problems relating to subsets of the edgeset of a graph can be phrased as a Holant problem and we will see an example of this in #AntiFactor$_1$.

7.3 Branching in DAGs

Many dynamic programming algorithms on linear structures can be rephrased in terms of deciding the existence of paths in specific directed acyclic graphs: Given an input instance I for a problem, the instance is transformed into a directed acyclic graph $D = D(I)$, with unique source and sink vertices $s, t \in V(D)$, such that solutions for I correspond to s-t-paths in D. For counting problems, it should be possible to recover the number of solutions for I efficiently from the number of s-t-paths in D. For tree-like structures, trees should play the role of paths, and there may not be a unique sink.

7.3.1 Adorned DAGs

We capture the above considerations through the notion of an *adorned DAG*.

Definition 7.3.1. *An* adorned DAG *is a tuple $\overline{D} = (D, w, B, T, b)$, where*

- *D is a DAG with $V(D) = [0,n]$ for some number $n \in \mathbb{N}$,*
- *$w : E(D) \to \mathbb{Z}_{\geq 0}$ assigns non-negative integer weights to the edges of D,*

- $B, T \subseteq V(D)$ *are referred to as branch and terminal vertices respectively, with* $B \cap T = \emptyset$,
- *the out-degree of every* $v \in B$ *is exactly* 2.

We call $\overline{D}$ *path-like if* $B = \emptyset$. *A branching in* $\overline{D}$ *is a directed tree* A *in* D, *rooted at* $0 \in V(D)$ *and edges directed towards children, such that*

- *each branch vertex* $v \in B$ *has two children* w_0 *and* w_1 *in* A,
- *every terminal vertex* $v \in V(A)$ *is a leaf,*
- *each non-branch, non-terminal vertex* $v \in V(D) \setminus (B \cup T)$ *has one child in* A.

The weight of A *is defined as* $w(A) = \prod_{e \in A} w(e)$. *The depth of* A *is the maximum length of a path from* 0 *to a leaf. Note that any branching in a path-like adorned DAG is a path.*

In the Definition, the weight of T is defined as the *product* of occurring edge-weights rather than its sum. This is because edge-weights will be used to encode independent choices that contribute towards a solution. To determine the overall number of choices, these numbers should be multiplied.

Intuitively a branching is formed by starting at the root vertex 0 and walking along the arcs of the DAG in the following manner. If the current vertex is a terminal vertex (or has no out-neighbors), we stop. If the current vertex is a branch vertex, we duplicate and walk to both out-neighbors. Otherwise we choose one of the out-neighbors of the current vertex (in a non-deterministic manner) and walk to that vertex. The process is considered a success if all terminal vertices are visited (by some duplicate).

There is a natural analogue between branchings in adorned DAGs and computation trees in alternating Turing machines. One can think of each vertex in the DAG as a configuration and the root 0 as the starting configuration. Arcs in the DAG represent sequences of computation steps on the Turing machine. If a vertex has multiple out-neighbors, it represents a (co-)non-deterministic step in the computation, with branch vertices being co-non-deterministic and non-branch vertices being non-deterministic.

7.3.2 Succinct Representations of Adorned DAGs

In the setting of space-bounded computation, it may not be possible to store an explicit representation of the adorned DAG $\overline{D}$ derived from an instance I within the prescribed memory bounds. However, for many problems, an implicit representation suffices: Adorned DAGs $\overline{D}$ can be encoded via Turing machines M, possibly much more succinctly than via an explicit encoding. In the following, let $\langle\rangle : \mathbb{N}^* \to \{0,1\}^*$ be an injective pairing function that is logspace-computable, as well as its inverse, and which maps a variable number of arguments from $\mathbb{N}$ into a bitstring.

Definition 7.3.2. *A deterministic Turing machine* M encodes *an adorned DAG* $\overline{D} = (D, w, b, B, T)$ *if it outputs the following on input* $\langle i, q, r\rangle$ *with* $i, r \in \mathbb{N}$ *and* $q \in \{0,1\}$:

- *If $q = 0$, then M outputs $\langle z, d\rangle$ such that:*
  - *The number $z \in \{0,1,2\}$ indicates whether i is contained in B (1), T (2), or neither (0), and*
  - *d is the out-degree of i.*
- *If $q = 1$, then M outputs $\langle j, w(ij)\rangle$, where j is the r-th out-neighbor of i, in ascending order.*

The intuition here is as follows. M can be seen as an oracle that provides local information about the dag $\overline{D}$. i is the vertex we want to ask a question about. The bit q indicates whether we want to ask M for information about i itself ($q = 0$), or one of its out-neighbors ($q = 1$). r is only relevant if $q = 1$ and indicates the out-neighbor of i we want to know about. Note that this gives us enough information to walk through $\overline{D}$ along the direction of the arcs and to branch when we want to.

7.3.3 Completeness of Counting Branchings in DAGs

We now turn to a pair of counting problems for Succinct Adorned DAGs, that will form our canonical #XLP- and #XALP-complete problems. Recall that the depth of a branching is the length of the longest path from the root to any of the leaves of the branching.

#Succinct Adorned DAG Branchings (#SAD Branchings)
Input: Integer d (in unary), a $k \log n$ space- and $f(k) \cdot n^c$ time-bounded deterministic Turing machine M, an input x of length n for M such that $M(x, \cdot)$ encodes an adorned DAG $\overline{D}$.
Parameter: k
Question: The number of branchings of depth at most d in $\overline{D}$.

If the adorned dag $\overline{D}$ is additionally required to be path-like, we call the resulting problem #Path-Like SAD Branchings for short. Note that for the path-like variant, the branchings are all paths.

We can think of this problem as just counting branchings in DAGs, but in a setting where the DAG is provided implicitly by the input x and the Turing machine M. For example x might consist of a graph G with some tree decomposition. The vertices of the DAG might correspond to colorings of the bags of the tree decomposition and M might relay information about compatible colorings of neighboring bags. In this example we see that there is sometimes a direct link between vertices in an adorned DAG and states in a dynamic programming algorithm.

In essence #SAD Branchings problem is just a rephrasing of counting accepting computation trees of an alternating Turing machine (see also the discussion in Subsection 7.3.1), with bounded depth, i.e. running time. This rephrasing is will turn out to be quite useful, as it translates the (co-)non-deterministic aspects of the setting into the language of graphs. Since many of our problems are graph problems, this makes #SAD Branchings and #Path-Like SAD Branchings much easier to work with.

Lemma 7.3.3. *#SAD Branchings is in #XALP and #Path-Like SAD Branchings is in #XLP.*

Proof. We give a proof for #SAD Branchings and note that the same proof works for #Path-Like SAD Branchings if we leave out the parts about branching vertices. Given the Turing machine M that encodes the DAG $\overline{D}$, we repeat the following computation. We first write down the ID of the current vertex v in the DAG, starting with the root vertex. We then run M to find the children of v.

If v is a terminal vertex, we terminate.

If v is a non-branch, non-terminal vertex, choose a child u in a non-deterministic fashion. If the edge uv has weight w, we simulate this as follows. We create a binary tree rooted at v, with w leaves, consisting of non-branching vertices. We then connect each leaf to u. Note that there are w paths from v to u in this tree and thus we replaced a branching of weight $w \cdot C$ by w branchings of weight C, for some C. We then write down the ID of u, replacing that of v, and repeat the process.

If v is a branch vertex, we branch the computation to all children of v, again simulating the edge-weights with arbitrary branching.

We iterate until all branches of the computation have reached a terminal vertex.

It is easy to see that a computation tree in this process corresponds with a branching of $\overline{D}$. Since we only need to save vertex ID's and the internal storage of M to memory, we only need $O(f(k)\log n)$ space. □

We proceed to prove Theorem 7.1.1 by showing that these problems are also hard for their respective classes.

Theorem 7.1.1.

a. *#SAD Branchings is #XALP-complete.*

b. *#Path-Like SAD Branchings is #XLP-complete.*

Proof. Membership for both cases is shown in Lemma 7.3.3. We prove hardness for item (a) and observe that for item (b), it follows from essentially the same argument, ignoring co-non-deterministic steps of XALP-machines.

Let Π be a parameterized counting problem contained in #XALP, and let M_Π be its XALP-machine. Let f be a function and c a constant such that on inputs I with parameter k, where $n = |I|$, M_Π uses at most $f(k)\log n$ space and each computation tree of M_Π on such inputs has size at most $f(k)n^c$.

Now, fix any input I to M_Π, and let $n = |I|$ and $k = k(I)$ the associated parameter throughout. We construct an instance $(d, M, x, \overline{D})$ of #SAD Branchings. We let $d = f(k)n^c$, $x = I$, and we describe M and $\overline{D}$ in the following.

We first describe the adorned DAG $\overline{D}$. As $\overline{D}$ is unweighted in this reduction, it suffices to consider $\overline{D}$ as a triple (D, B, T) of a DAG D, a set $B \subseteq V(D)$ of branching nodes, and a set $T \subseteq V(D)$ of terminals. (As the branching nodes correspond to co-non-deterministic steps in M_Π, they can be left out in the proof of item (b).)

The vertices of D essentially correspond to the configurations of M_Π on input I. Note that since the input tape of M_Π is read-only, it suffices to store the position of the tape head alone, for which there are only n choices. Let Γ_Π be the tape alphabet of M_Π. Each such configuration is a tuple (q, p_1, p_2, s) where q is a state of M_Π, $p_1 \in [n]$

is the position of the first tape head, $p_2 \in [f(k)\log n]$ is the position of the second tape head, and s is a string in Γ_Π^* of length at most $f(k)\log n$. We choose an integer m large enough such that each configuration of M_Π on input I can be encoded as a binary string of length precisely $\lfloor \log m \rfloor + 1$. In particular, for each configuration C of M_Π on input I, there will be an integer $i \leq m$ such that the binary expansion of i using $\lfloor \log m \rfloor + 1$ bits will be a binary encoding of C. For each configuration (q, p_1, p_2, s), a constant number of bits suffices to encode q, $\mathcal{O}(\log n)$ bits suffice to encode p_1, $\log(f(k)\log n)$ bits suffice for p_2, and $\mathcal{O}(\log(|\Gamma_\Pi|) \cdot f(k)\log n)$ bits suffice for s. Therefore, it suffices to take some $m \in n^{\mathcal{O}(f(k))}$. The vertex set of D is $[m]$. For each vertex $i \in [m]$ of D, consider its binary expansion s_i using precisely $\lfloor \log m \rfloor + 1$ bits. It is of one of the following types.

1. If s_i is the all-zeroes string, then $i = 0$ corresponds to the starting configuration of M_Π on input I.
2. If s_i is the binary encoding of a non-starting configuration C of M_Π on input I, then i corresponds to C.
3. Otherwise, i is a garbage vertex.

Note that with access to I and a description of M_Π, a Turing machine can check if some integer $i \in [m]$ corresponds to a configuration of M_Π on input I in polynomial time. Since $\log m \in \mathcal{O}(f(k)\log n)$, the space usage of such a machine can be bounded by $f'(k)\log n$ for some function f', if we assume that I is written to the read-only input tape of the machine.

In D, all garbage vertices will be isolated. For all $i, j \in [n]$ such that i corresponds to a configuration C_i and j corresponds to a configuration C_j such that C_i yields C_j, there is an arc from i to j in D. Note that this indeed yields a DAG, since otherwise M_Π would loop on input I, which is impossible since an XALP-machine must terminate in at most $f(k)n^c$ time. Recall that a co-non-deterministic configuration is a configuration whose state is labeled co-non-deterministic. We let:

$$B = \{i \in V(D) \mid i \text{ corresponds to a co-non-deterministic configuration}\}.$$
$$T = \{i \in V(D) \mid i \text{ corresponds to an accepting configuration}\}.$$

Finally, we describe M, a Turing machine that encodes $\overline{D}$. By the definition of the #SAD Branchings problem, we may assume that I is written on the input tape of M. As a first step, we hard-code a description of M_Π into M by writing it to its work tape. Note that the description of M_Π has constant size, therefore this does not occur any prohibitive overhead on the space usage. Throughout the following, we can freely assume that M has access to both I and M_Π.

Let $\langle i, q, r \rangle$ be an input to M. If $q = 0$, we check if i corresponds to a configuration C_i of M_Π on input I; if not, we return $\langle 0, 0 \rangle$. Otherwise, we check if the state of C_i is the accepting state of M_Π; if yes, we set $z = 2$. Otherwise, if C_i is co-non-deterministic, we set $z = 1$; and otherwise $z = 0$. Using our access to M_Π and I, we can check how many configurations C_j exist such that $C_i = (q, p_1, p_2, s)$ yields $C_j = (q', p_1', p_2', s')$. The number of possibilities for C_j we have to check is constant:

there is a constant number of states in M_Π, we know that $p_1' \in \{p_1 - 1, p_1, p_1 + 1\}$, $p_2' \in \{p_2 - 1, p_2, p_2 + 1\}$, and s' can be obtained from s by (possibly) replacing the character at position p_2, again a constant number of choices. By construction, the number δ of configurations C_j such that C_i yields C_j is the out-degree of vertex i in D. We return $\langle z, \delta \rangle$. All of these operations can be done using $\mathcal{O}(\log m)$ space and $(\log m)^{\mathcal{O}(1)}$ steps. Since $\log m \leq f'(k) \log n$ for some function f', this satisfies the required bounds on the space and time usage of M.

If $q = 1$, then again, we first check if i corresponds to some configuration C_i or not. If not, then we report an error (as in this case, i has no out-neighbors). Otherwise, we enumerate all configurations C_j such that C_i yields C_j in ascending order, meaning $C_{j_1}, C_{j_2}, \ldots$ such that $j_1 < j_2 < \ldots$, using I and the description of M_Π, and output $\langle j_r \rangle$. If we find that there are less than r such configurations, we report an error. Once more, these steps can be done within the required space- and time-bounds.

It is clear from the construction that there is a bijective correspondence between accepting computation trees in M_Π on input I and the branchings in $\overline{D}$. □

7.4 Multicolored Pattern Problems

In this section we show hardness for the #Thin Multicolored Grids and #Tree-Shaped Thin Multicolored Grids problems. As a corollary we also find hardness for #Chained Multicolored Clique, #Tree-Chained Multicolored Clique, #Chained Multicolored Independent Set and #Tree-Chained Multicolored Independent Set. As mentioned in the results section, #XLP- and #XALP-membership follows from the entire sequence of reductions. It thus suffices to show hardness.

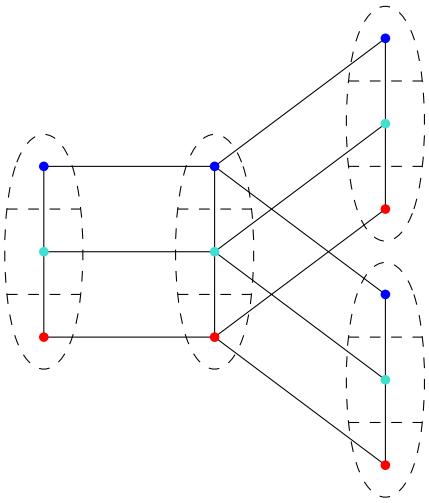


Figure 7.2: An example of a tree-shaped multicolored thin grid. The partition sets V_t and their partition sets $V_{t,i}$ are indicated with dashed lines. The colors have no additional meaning beyond indication the different sets $V_{t,1}, V_{t,2}, V_{t,3}$.

For the proof of Lemma 7.4.2, we will make use of the following lemma.

Lemma 7.4.1 (Bodlaender, Groenland, Jacob, Pilipczuk, Pilipczuk [22])**.** *For every XALP-machine M which on inputs of size n and parameter k uses at most $f(k)\log n$ space and has computation trees of size at most $f(k)n^c$, for some computable function f and constant c, there is an XALP-machine M^* that decides the same language using $\mathcal{O}(f(k)\log n)$ space, and such that each computation tree T of M^* on an input of length n and parameter k has the following property: T is obtained from a binary tree of height $\mathcal{O}(\log n) + f(k)$ by subdividing each edge the same number of times, which is at most $f(k)n^c$.*

We now formally state the problem we will discuss in this section.

#Tree-Shaped Thin Multicolored Grids
Input: Graph G, binary tree T, a partition $\{V_t\}_{t\in V(T)}$ of $V(G)$, and for each $t \in V(T)$, a partition $V_{t,1}, \dots, V_{t,k}$ of V_t such that

$$E(G) \subseteq \{uv : u \in V_{t,i}, v \in V_{t,i+1}, t \in V(T), i \in [k-1]\}$$
$$\cup \{uv : u \in V_{t,i}, v \in V_{t',i}, tt' \in E(T), i \in [k]\}.$$

Parameter: k
Question: How many sets $W \subseteq V(G)$ are there, such that for all $(t,i) \in V(T) \times [k]$: $|W \cap V_{t,i}| = 1$; and for all $st \in E(T)$, $G[W \cap (V_s \cup V_t)]$ is a $2 \times k$-grid?

The special case of the previous problem where T is required to be a path is called #Thin Multicolored Grids. Note that in this case, we can make the path structure implicit by assuming that we are given a partition $V_1, \dots, V_r$ of the vertex set, instead of an explicit path.

Lemma 7.4.2.

a. *#Tree-Shaped Thin Multicolored Grids is #XALP-hard.*

b. *#Thin Multicolored Grids is #XLP-hard.*

Proof. We prove item a and note that the proof of item b can be done analogously, starting from an XNLP-machine instead of an XALP-machine and going via #Path-Like SAD Branchings instead of #SAD Branchings.

We prove item a. Let Π be a parameterized counting problem contained in XALP, and let M_Π be its XALP-machine that has the special properties described in Lemma 7.4.1. We furthermore make the standard assumption that the tape alphabet of M_Π is binary. Let I be an input to M_Π with $|I| = n$ and associated parameter $k = k(I)$. Let $\mathbb{T}$ be the binary tree that dictates the shape of computation trees of M_Π on input I. Let $\mathbb{T}'$ be the tree of height $\mathcal{O}(\log n) + f(k)$ such that $\mathbb{T}$ is obtained from $\mathbb{T}'$ by subdividing each of its edges $s \leq f(k)n^c$ times. Now, we run the reduction from the proof of 7.1.1.a to obtain an instance $\mathcal{I} = (f(k)n^c, M, I, \overline{D})$ of #SAD Branchings with adorned DAG $\overline{D} = (D, B, T)$. (Throughout the following, we disregard the garbage vertices in D, as they are isolated and therefore have no influence on any solution to $\mathcal{I}$.)

As the structure of $\overline{D}$ models the computation of M_Π on input I we know that there is a partition[2] $\{V_t\}_{t\in V(\mathbb{T})}$ of $V(D)$ such that:

(i) $V_{\mathbf{r}} = \{0\}$ where $\mathbf{r}$ is the root of $\mathbb{T}$.

(ii) For each $t \in V(\mathbb{T})$ with one child: $V_t \subseteq V(D) \setminus (B \cup T)$.

(iii) For each $t \in V(\mathbb{T})$ with two children: $V_t \subseteq B$.

(iv) For each leaf $t \in V(\mathbb{T})$: $V_t \subseteq T$.

(v) $E(D) \subseteq \bigcup_{st\in E(\mathbb{T})} V_s \times V_t$.

We use this structure to create our instance of #Tree-Shaped Thin Multicolored Grids. However, for each $t \in V(\mathbb{T})$, the size of V_t could be as big as $n^{\Omega(f(k))}$, the number of configurations of M_Π on input I. We therefore use the same compression technique as in [22] to give representations of these sets of small enough size, i.e. $f'(k)n^{c'}$.

We now follow the proof from [22] in a near-verbatim way. Let v be a vertex in D, corresponding to a configuration (q, p_1, p_2, s) of M_Π on input I. Recall that q is a state of M_Π, p_1 the position of the input tape head, p_2 the position of the work tape head, and s the work tape content, i.e., a binary string of length at most $f(k)\log n$. For each $t \in V(\mathbb{T})$, we construct a set of vertices X_t partitioned into $X_{t,1}, \ldots, X_{t,f(k)}$ such that each configuration can be encoded as a choice of one vertex from each part of the partition. For each $j \in [f(k)]$, the vertices in $X_{t,j}$ encode the content of the j-th block of $\log n$ bits of the work tape content. To this end, each $X_{t,j}$ contains, for all states q of M_Π, for all $p_1 \in [n]$, for all $p_2 \in [f(k)\log n]$, and all $w \in \{0, 1, \sqcup\}^{\log n}$ a vertex $v_{q,p_1,p_2,w} = u_{q,p_1,p_2,w}$ (for convenience, we give this vertex two names) that can correspond to any configuration (q, p_1, p_2, s), where the j-th block of $\log n$ consecutive bits of s is equal to w. (We assume that s always has length precisely $f(k)\log n$; if fewer cells are used we fill up the remaining positions with blanks.)

The vertex set of the graph G is $V(G) = \bigcup_{t\in V(\mathbb{T}), j\in[f(k)]} X_{t,j}$. We add two types of edges. First, between consecutive blocks of each X_t, $t \in V(\mathbb{T})$, to ensure that each valid choice in a solution corresponds to one configuration (rather than a mix of some): For each $t \in V(\mathbb{T})$ and $j \in [f(k) - 1]$, we add an edge between $u_{q,p_1,p_2,w} \in X_{t,j}$ and $v_{q',p_1',p_2',w'} \in X_{t,j+1}$, if $q = q'$, $p_1 = p_1'$, and $p_2 = p_2'$. Next, for each edge $tt' \in E(\mathbb{T})$ (where t is the parent and t' the child), and each $j \in [f(k)]$, we add the following edges between $X_{t,j}$ and $X_{t',j}$. Let $u_{q,p_1,p_2,w} \in X_{t,j}$ and $v_{q',p_1',p_2',w'} \in X_{t',j}$. There are two cases to consider.

- Case 1 ($p_2 \notin [(j-1)\log n + 1, j\log n]$): We connect $u_{q,p_1,p_2,w}$ and $v_{q',p_1',p_2',w'}$ if and only if $w = w'$. In this case, the tape head of the work tape is not on the j-th block, so no change to w can happen. The remaining components of the configuration could have changed.

[2] Observe that this is indeed a partition: if a configuration appeared in more than one part, then M_Π does not halt on input I.

- Case 2 ($p_2 \in [(j-1)\log n+1, j\log n]$): Let $p_2^* = (p_2 \bmod \log n)+1$, so that the p_2-th position of the work tape content corresponds to the p_2^*-th position in the j-th block. We connect $u_{q,p_1,p_2,w}$ and $v_{q',p_1',p_2',w'}$ if and only if there is a transition in M_Π that upon reading $I[p_1]$ on the input tape and $w[p_2^*]$ on the work tape, writes $w'[p_2^*]$ to the work tape, moves the input tape head to position p_1' and the tape head position to p_2', and if $w[h] = w'[h]$ for all $h \in [\log n] \setminus \{p_2^*\}$.

We clean up the graph G in the following way:

1. From $X_{\mathbf{r}}$, where $\mathbf{r}$ is the root of $\mathbb{T}$, we remove all vertices except the ones that correspond to the starting configuration of M_Π on input w, i.e., are of the form $(q_0, 1, 1, \square^{\log n})$, where q_0 is the starting state of $\mathbb{T}$.

2. For each non-root node $t \in V(\mathbb{T})$: (a) If t has one child, then we remove all vertices from X_t that have co-non-deterministic states. (b) If t has two children, then we remove all vertices from X_t that have non-deterministic states. (c) If t has no children, then we remove all vertices from X_t that do not have the accepting state.

This finishes the reduction. Observe for the size of G that each X_t, for $t \in \mathbb{T}$, has at most $f'(k)n^{c'}$ vertices, for some function f' and small constant c'. Since the size of $\mathbb{T}$ is at most $f(k)n^c$, the size of G is at most $f''(k)n^{c''}$ for some function f'' and constant c''. Moreover, all operations to construct G can be performed using only $\mathcal{O}(f(k)\log n)$ space.

We use the following notation. A $\mathbb{T}$-*path in* G is a path $v_1, \ldots, v_{f(k)}$ such that there exists some $t \in V(\mathbb{T})$ where for all $j \in [f(k)]$, $v_j \in X_{t,j}$. We also call this path a $\mathbb{T}$-*path in* G_t, if t is not clear from context. A *local* $\mathbb{T}$-*grid in* G is a $2 \times f(k)$-grid with consisting of $u_1, \ldots, u_{f(k)}$, a $\mathbb{T}$-path in G_s, and $v_1, \ldots, v_{f(k)}$, a $\mathbb{T}$-path in G_t, for some $st \in E(\mathbb{T})$, where for all $j \in [f(k)]$, $u_j v_j \in E(G)$. We call the path on vertices $u_1, \ldots, u_{f(k)}$ its *top* $\mathbb{T}$-*path* and the path on vertices $v_1, \ldots, v_{f(k)}$ its *bottom* $\mathbb{T}$-*path*.

Observation 7.4.3.
a. *Each* $\mathbb{T}$*-path encodes a configuration of* M_Π *on input* I.
b. *Each local* $\mathbb{T}$*-grid encodes a pair* C_1, C_2 *of configurations of* M_Π *on input* I *such that* C_1 *yields* C_2*; moreover, its top* $\mathbb{T}$*-path encodes* C_1 *and its bottom* $\mathbb{T}$*-path encodes* C_2.

Claim 7.4.4. *There is an injection* ψ *from* $V(D)$ *to the* $\mathbb{T}$*-paths in* G *such that*
a. *For all* $t \in V(\mathbb{T})$, ψ *maps* V_t *to the* $\mathbb{T}$*-paths in* G_t.
b. *For each* $uv \in E(D)$, $G[V(\psi(u)) \cup V(\psi(v))]$ *is a local* $\mathbb{T}$*-grid.*
c. *Let* P *be a* $\mathbb{T}$*-path that is not in the image of* ψ*. Then, there is no tree-shaped thin multicolored grid that has* P *has a subgraph.*

Proof. We construct ψ as follows. Recall that each vertex $v \in V(D)$ corresponds to a configuration (q, p_1, p_2, s) of M_Π on input I. Moreover, there is some $t \in V(\mathbb{T})$ such that $v \in V_t$. For $j \in [f(k)]$, let w_j be the j-block of $\log n$ consecutive characters of s, such that $s = w_1 w_2 \ldots w_{f(k)}$. Then, we map v to the $\mathbb{T}$-path consisting of the vertices v_{q,p_1,p_2,w_1}, v_{q,p_1,p_2,w_2}, …, $v_{q,p_1,p_2,w_{f(k)}}$, where for all $j \in [f(k)]$, $v_{q,p_1,p_2,w_j} \in X_{t,j}$ (using the notation from the construction above). It is clear that this construction gives an

injection from $V(D)$ to the $\mathbb{T}$-paths in G that satisfies item a. For item b, consider an edge $uv \in V(D)$; then, for some $tt' \in E(\mathbb{T})$, where t is the parent and t' the child, $u \in V_t$ and $v \in V_{t'}$. By construction of D, the configuration corresponding to u yields the configuration corresponding to v. By construction of G, $G[V(\psi(u)) \cup V(\psi(v)]$ is a local $\mathbb{T}$-grid.

For item c, suppose for a contradiction that there is some $\mathbb{T}$-path P contained in a tree-shaped thin multicolored grid W such that P is not in the image of ψ. By construction, there is only one $\mathbb{T}$-path P_0 in $G_{\mathbf{r}}$, where $\mathbf{r}$ is the root of $\mathbb{T}$. Therefore, W must contain P_0. Moreover, P_0 encodes the starting configuration of M_Π on input (I,k). This configuration in turn is represented by the vertex 0 in D, and therefore P_0 is in the image of ψ. Using induction and Observation 7.4.3, we eventually reach the contradiction that P is in fact in the image of ψ. ■

From Claim 7.4.4 and the construction of G we can infer that there is a bijective correspondence between the branchings in $\overline{D}$ (and therefore the accepting computation trees of M_Π on input (I,k) by the proof of Theorem 7.1.1) and the tree-shaped thin narrow grids in G. □

As a corollary of Lemma 7.4.2 we also find hardness for the following, more well-known variant of #Thin Multicolored Grids.

#Tree-Chained Multicolored Clique
Input: Graph G, binary tree T, partition $\{V_t\}_{t\in V(T)}$ of $V(G)$, and for each $t \in V(T)$, a partition of V_t into $V_{t,1}, \ldots, V_{t,k}$.
Parameter: k
Question: How many sets $W \subseteq V(G)$ are there, such that for all $(t,i) \in V(T) \times [k]$: $|W \cap V_{t,i}| = 1$; and for all $st \in E(T)$: $W \cap (V_s \cup V_t)$ is a clique in G?

A relevant special case of the previous problem is #Chained Multicolored Clique, where we restrict T to be a path. In this case, we may for convenience assume that instances come with a partition $V_1, \ldots, V_r$ of $V(G)$ (where the order of the V_i's indicates the path structure) and ask to count sets W that induce cliques on each $V_i \cup V_{i+1}$.

The #Chained Multicolored Independent Set and #Tree-Chained Multicolored Independent Set problems are the analogues of the two above problems where we are interested in counting independent sets instead of vertex sets that induce cliques in adjacent parts. Note that in both problems, we may assume that there are no edges between parts that are not adjacent in the underlying tree- or path-structure.

Lemma 7.4.5.

a. *#Chained Multicolored Clique and #Chained Multicolored Independent Set are #XLP-hard.*

b. *#Tree-Chained Multicolored Clique and #Tree-Chained Multicolored Independent Set are #XALP-hard.*

Proof. We reduce from #Tree-Shaped Thin Multicolored Grids to #Tree-Chained Multicolored Clique and note that the reduction from #Thin Multicolored Grids

to #Chained Multicolored Clique is analogous. We also note that it is not hard to see that #Tree-Chained Multicolored Clique and #Tree-Chained Multicolored Independent Set are equivalent (idem for #Chained Multicolored Clique and #Chained Multicolored Independent Set), by swapping edges and non-edges between adjacent vertex sets V_t.

Given an instance of #Tree-Shaped Thin Multicolored Grids, we perform the following modifications to retrieve an instance G' of #Tree-Chained Multicolored Clique. For $V_{t,i}$ and $V_{t,j}$ such that $|i-j|>1$ we add all edges uv for $u \in V_{t,i}$ and $v \in V_{t,j}$. For $V_{t,i}$ and $V_{t',j}$ such that $i \neq j$ and $tt' \in E(T)$ we add all edges uv for $u \in V_{t,i}$ and $v \in V_{t',j}$. Note that this means that in the resulting instance of #Tree-Chained Multicolored Clique the adjacency requirements between these sets are automatically satisfied. The remaining adjacencies are precisely those involved in the #Tree-Shaped Thin Multicolored Grids instance. We find that a set $W \subseteq V(G')$ is a tree-chained multicolored clique in G' if and only if it is a tree-shaped thin multicolored grid in G.

It is clear that this reduction can be performed using logarithmic space, as all operations are local and can be described by only the indices of the sets involved. □

7.5 Satisfiability and Hitting Set Problems

In this section we give a series of simple reductions to prove hardness for #Partitioned Tree-Chained Weighted CNF-SAT, #Positive Partitioned Tree-Chained Weighted CNF-SAT, #Tree-Chained Multicolored Hitting Set and their linear counterparts. As before we state the formal definitions for the tree-shaped versions of the problems and note that the linear version of the problem is defined by replacing the tree in the definition by a path. We start with #Partitioned Tree-Chained Weighted CNF-SAT.

#Partitioned Tree-Chained Weighted CNF-SAT
Input: A binary tree T, a set of Boolean variables X partitioned into sets $\{X_t\}_{t\in V(T)}$, for each $t \in V(T)$, a partition of X_t into $X_{t,1},\dots,X_{t,k}$, and for each $e = st \in E(T)$, a CNF-formula F_e over variables $X_s \cup X_t$.
Parameter: k
Question: How many satisfying assignments of $F = \bigwedge_{e\in E(T)} F_e$ are there that set exactly one variable in each $X_{t,i}$ to true?

The #Partitioned Chained Weighted CNF-SAT problem is the restriction of #Partitioned Tree-Chained Weighted CNF-SAT to the case when T is a path. Here, instead of the explicit path structure, we may simply consider having a partition $X_1,\dots,X_r$ of X and formulas $F_1,\dots,F_{r-1}$ where for each $i \in [i-1]$, F_i has variables $X_i \cup X_{i+1}$.

Lemma 7.5.1.

a. *#Partitioned Chained Weighted CNF-SAT is #XLP-hard, even when all formulas are in 2-CNF and only contain negated literals.*

b. *#Partitioned Tree-Chained Weighted CNF-SAT is #XALP-hard, even when all formulas are in 2-CNF and contain only negated literals.*

Proof. We first prove item a. We reduce from #Chained Multicolored Independent Set, which is #XLP-hard by Lemma 7.4.5.a. Let $(G, V_{1,1}, \ldots, V_{1,k}, \ldots, V_{r,1}, \ldots, V_{r,k})$ be an instance of #Chained Multicolored Independent Set. We create an instance $(X_{1,1}, \ldots, X_{1,k}, \ldots, X_{r,1}, \ldots, X_{r,k}, F_1, \ldots, F_{r-1})$ of #Partitioned Chained Weighted CNF-SAT, where for all $i \in [r-1]$, F_i is in 2-CNF containing only negated literals. For each $(i,j) \in [r] \times [k]$, we let $X_{i,j} = \{x_v \mid v \in V_{i,j}\}$. Let $uv \in E(G)$. As in the #Chained Multicolored Independent Set instance, we may assume that there are no edges between V_i and V_j whenever $|i-j| > 1$, there is some $i \in [r]$ such that $\{u,v\} \subseteq V_i \cup V_{i+1}$. We add the clause $(\overline{x_u} \vee \overline{x_v})$ to F_i. Observe that this reduction can be implemented to run in logarithmic space.

There is a natural bijective correspondence between chained multicolored independent sets in G and the satisfying truth assignments to $F = \bigwedge_{i \in [r-1]} F_i$ that set one variable per $X_{i,j}$ to true: the variable set to true is the one that corresponds to the vertex picked in the independent set.

Using the same proof, but starting the reduction from #Tree-Chained Multicolored Independent Set instead, proves item b. □

Using the a simple reduction we can extend Lemma 7.5.1 to the following restricted setting.

#Positive Partitioned Tree-Chained Weighted CNF-SAT
Input: A binary tree T, a set of Boolean variables X partitioned into sets $\{X_t\}_{t \in V(T)}$, for each $t \in V(T)$, a partition of X_t into $X_{t,1}, \ldots, X_{t,k}$, and for each $e = st \in E(T)$, a CNF-formula F_e over variables $X_s \cup X_t$, using no negated variables.
Parameter: k
Question: How many satisfying assignments of $F = \bigwedge_{e \in E(T)} F_e$ are there that set exactly one variable in each $X_{t,i}$ to true?

Note that this problem is essentially a rephrasing of the following problem.

#Tree-Chained Multicolored Hitting Set
Input: A binary tree T, a universe U partitioned into sets $\{U_t\}_{t \in V(T)}$, for each $t \in V(T)$, a partition of U_t into $U_{t,1}, \ldots, U_{t,k}$, and for each $e = st \in E(T)$, a collection $\mathcal{A}_e$ of subsets of $\bigcup_{i=1}^{k} (U_{s,i} \cup U_{t,i})$.
Parameter: k
Question: How many sets $W \subseteq U$ are there, such that for all $(t,i) \in V(T) \times [k]$: $|W \cap U_{t,i}| = 1$; and for all $e \in E(T)$, for all $A \in \mathcal{A}_e$: $W \cap A \neq \emptyset$?

Lemma 7.5.2.

a. *#Positive Partitioned Chained Weighted CNF-SAT and #Chained Multicolored Hitting Set are #XLP-hard.*

b. *#Positive Partitioned Tree-Chained Weighted CNF-SAT and #Tree-Chained Multicolored Hitting Set are #XALP-hard.*

Proof. We first prove item a. First note that #Positive Partitioned Chained Weighted CNF-SAT and #Chained Multicolored Hitting Set are equivalent, by mapping either a clause to the set of variables appearing in that clause, or a set to the clause containing all element from that set. We will therefore focus on #Positive Partitioned Chained Weighted CNF-SAT.

Let $(F_e)_{e\in E(T)}$ be the sequence of CNF-formulas for a given instance of #Partitioned Chained Weighted CNF-SAT. Note that, since we must set exactly one variable from each set $X_{t,i} = \{x_1,\dots,x_\ell\}$ to true, we have that the formulas $\neg x_j$ and $\phi_j := x_1 \vee x_2 \vee \cdots x_{j-1} \vee x_{j+1} \vee \cdots x_\ell$ are equivalent. We can therefore create an equivalent sequence of CNF-formulas $(F'_e)_{e\in E(T)}$ by replacing each instance of x_j with ϕ_j. Note that if we do this for every negated variable, we produce an instance of #Positive Partitioned Chained Weighted CNF-SAT.

Using the same proof, but starting the reduction from #Partitioned Tree-Chained Weighted CNF-SAT instead, proves item b. □

7.6 AntiFactor

In this section we show #XLP- and #XALP-completeness for the #AntiFactor$_1$/pw and #AntiFactor$_1$/tw respectively. We give detailed proofs for the parameterization by treewidth and note that the proofs for pathwidth are essentially analogues. The problem (parameterized by treewidth) is defined as follows.

#AntiFactor$_1$
Input: A graph G, a width tw tree decomposition $(\mathbb{T},(B_x)_{x\in V(\mathbb{T})})$ of G, an integer d.
Parameter: tw
Question: How many edge subsets $A \subseteq E(G)$ are there, for which no vertex in $V(G)$ is incident to exactly d edges in A.

We can write #AntiFactor$_1$ as a Holant problem, by setting all edge weights to 1 and setting

$$f_v(x) := \begin{cases} 0 & \text{if } x(e) = 1 \text{ for exactly } d \ e \in \delta(v), \\ 1 & \text{otherwise.} \end{cases}$$

There is a closely related problem called #Factor$_1$, where instead of forbidding a degree d, we require every vertex to be incident to exactly d edges in A. Both #Factor$_1$ and #AntiFactor$_1$ are special cases of the more general #General Factor. In this chapter we only consider the #AntiFactor$_1$ problem because #Factor$_1$ can be shown to be FPT-time solvable for tree- and pathwidth, as a corollary of [127, Theorem 1.3].

Corollary 7.6.1. *There exists an algorithm that can solve #Factor$_1$/tw in time* $(\mathrm{tw}+1)^{\mathrm{tw}}$ $n^{O(1)}$.

Proof. Let d be the allowed degree and G the graph from the given #Factor$_1$/tw instance. First note that if the minimum degree $\delta(G)$ of G is $\delta(G) < d$, then we trivially have 0 solutions. Also note that the treewidth tw of G is at least $\mathrm{tw} \geq \delta(G)$ and thus

we may assume $\mathrm{tw} \geq d$. By [127, Theorem 1.3] we can then solve the instance in $(d+1)^{\mathrm{tw}} n^{O(1)} \leq (\mathrm{tw}+1)^{\mathrm{tw}} n^{O(1)}$ time. □

We now turn our attention to the #AntiFactor$_1$ problem. We first show membership of #AntiFactor$_1$/tw (#AntiFactor$_1$/pw) in #XALP (#XLP).

Lemma 7.6.2.

a. *The problem #AntiFactor$_1$/tw is contained in the class #XALP.*

b. *The problem #AntiFactor$_1$/pw is contained in the class #XLP.*

Proof. We give a reduction from #AntiFactor$_1$/tw to #SAD Branchings (and from #AntiFactor$_1$/pw to #Path-Like SAD Branchings). By Lemma 7.3.3 the claim then follows.

We will essentially turn a simple dynamic program for #AntiFactor$_1$ into an adorned DAG and then show that has a succinct representation. A branching in this DAG will correspond to a collection of cells that together form a valid solution, i.e. a subset of the edges that does not have any vertices with the forbidden degree d. We give the proof for treewidth and note that we get a proof for pathwidth, by ignoring the join bags.

Let $(G, (\mathbb{T}, (B_x)_{x \in V(\mathbb{T})}), d)$ be an instance of #AntiFactor$_1$/tw and let $x \in V(\mathbb{T})$ be a node in the tree decomposition. We may assume that we are given a nice tree decomposition with edge-introduce bags. Furthermore, for convenience we assume that the root node and leaves of the decomposition have empty bags. We will now describe the adorned DAG $\overline{D} = (D, w, B, T)$.

We write $I(B_x)$ for the set of edges incident to a vertex in B_x, that have been introduced by x or some descendant of x. For each $A \subseteq I(B_x)$, we create a vertex $v_{x,A}$.

If x is a vertex-introduce bag with child y, we create an edge from $v_{x,A}$ to $v_{y,A}$. If x is an edge-introduce bag for edge e with child y, we create an edge from $v_{x,A}$ to $v_{y,A \setminus \{e\}}$. In other words we create an edge to each vertex corresponding to an edgeset that is consistent with A.

If x is a vertex-forget bag for vertex v with child y, we create an edge from $v_{x,A}$ to $v_{y,A}$, if v does not have degree d in (B_x, A). Otherwise we add no edges leaving $v_{x,A}$.

If x is a join node with children y_1 and y_2 we add edges $v_{x,A}v_{y_1,A}$ and $v_{x,A}v_{y_2,A}$. We add the vertices $v_{x,A}$ to the set of branch vertices B.

If x is a leaf, we add $v_{x,\emptyset}$ to the set T of terminal vertices. All edges e get weight $w(e) = 1$.

We will now show that there is a one-to-one mapping between branchings of $\overline{D}$ and edge subsets $S \subseteq E(G)$ such that no vertex has degree d in $(V(G), S)$. For a node $x \in V(\mathbb{T})$ let $V_x = \{v_{x,A} : A \subseteq I(B_x)\}$ and if x has two children, let $U_x^i = \{u_{x,A}^i : A \subseteq I(B_x)\}$.

Let R be a branching in $\overline{D}$. First note that R must contain the root $v_{r,}$ and every leaf $v_{x,\emptyset}$ and thus every node x has at least one vertex in V_x visited[3] by the branching. Since every edge is directed from V_x towards V_y, if y is a child of x, and only branch vertices have multiple children, a node x has at most one, and thus exactly one vertex in V_x visited. It follows that we can define $A_x \subseteq I(B_x)$ as the unique set such that v_{x,A_x} is visited by the branching.

[3]Here we refer to a vertex as being visited, if it has at least one edge adjacent to in in the branching.

We will map R to the set $S := \cup_{x \in V(\mathbb{T})} A_x$. We will now argue that no vertex has degree d in $(V(G), S)$. First note that by construction for any node x with a single child y, we have that $A_x \cap I(B_y) = A_y \cap I(B_x)$, i.e. the edge subsets A_x are consistent with one another. We find that a vertex $v \in V(G)$ has degree d in $(V(G), S)$, if and only if it has degree d in $(V(G), A_x)$ at the vertex-forget node x for v. Since v_{x,A_x} has a child in R, we find that v cannot have degree d in $(V(G), A_x)$. Finally note that by construction we have that for a join node x, $A_x = A_{y_1} = A_{y_2}$ and thus the choice of edges is consistent at join nodes. We conclude that S is a valid solution.

Clearly this mapping is injective, as each subset $S \subseteq E(G)$ defines a unique set of vertices $V_S = \{v_{x,S \cap I(B_x)} : x \in V(\mathbb{T})\}$ in the DAG, which a branching R must visit in order to be mapped to S.

Now let $S \subseteq E(G)$ be an edge subset such that no vertex has degree d in $(V(G), S)$. Let R be the branching given by taking all edges between vertices in V_S. Clearly the root $v_{r,\emptyset}$ and each terminal vertex is visited by the branching. Since we picked exactly one vertex from each V_x, we find that each branch vertex has exactly two children and each vertex from a edge-or vertex-introduce bag has exactly one child. Finally for vertex-forget nodes we find that since no vertex has degree d in $(V(G), S)$, the selected vertex has a child. We conclude that R is a valid branching. It is easy to see that R is the preimage of S under the previously described mapping.

It remains to show that there is some Turing machine T that encodes $\overline{D}$. Note that we can easily determine whether a vertex $v_{x,A}$ is a branch vertex, terminal vertex or neither, by examining the node x it corresponds to. Vertex-introduce, vertex-forget and edge-introduce nodes produce vertices with outdegree 1. Join nodes produce vertices with degree 2. We can produce out neighbors by choosing a child node. As nodes can be indexed using $O(\log n)$ space this is clearly a proper encoding. □

For the hardness, we follow the #W[1]-hardness proof by Marx et al. [128] (which in turn uses reductions from [127]). This proof consists of a series of small reductions. Rather than restating the full proofs for all these reductions, we will only discuss the changes we make and the additional insights necessary to prove #XALP-hardness.

Theorem 7.1.6.

a. *#AntiFactor$_1$/pw is #XLP-complete.*

b. *#AntiFactor$_1$/tw is #XALP-complete.*

Proof. By Lemma 7.6.2 we find membership for both cases. For hardness, we first apply either Lemma 7.6.3 or Lemma 7.6.4 to reduce either from #Tree-Chained Multicolored Hitting Set or #Chained Multicolored Hitting Set respectively to relation-weighted #AntiFactor$_1^{\mathcal{R}}$ with relation weights 1, -1 and $2-n$, parameterized by treewidth or pathwidth respectively. We then apply Lemma 7.6.8 to reduce from #AntiFactor$_1^{\mathcal{R}}$ to #AntiFactor$_1$, while increasing the tree- and pathwidth by at most an additive constant. □

We start by reducing from #Tree-Chained Multicolored Hitting Set to relation-weighted #AntiFactor$_1^{\mathcal{R}}$ a generalization of #AntiFactor$_1$, where some vertices are

replaced by *complex nodes*. In place of the degree constraint, these nodes have an arbitrary function on them called a *relation* that assigns a weight to the node based on the set of edges incident to the node. The weight of the solution is then the product of the weights of all complex nodes. For more details we refer the reader to [128]. In our proofs, the complex nodes will all be hidden in gadgets that we lift from the original proofs by Marx et al. [127, 128]. Throughout we write Δ^* to denote the maximum over all bags of the path/tree decomposition, of the total degree of the complex nodes in that bag.

Lemma 7.6.3. *There is an FPT-time many-one parameterized logspace reduction from #Tree-Chained Multicolored Hitting Set to relation-weighted #Anti- Factor$_1^{\mathcal{R}}$ with relation weights* $1, -1$ *and* $2-n$*. Let* n *be the size of the given universes,* m *the number of sets between adjacent nodes in the tree structure and* t *the number of nodes in the tree structure. The size of the produced instance is bounded by* $O(k \cdot n \cdot m \cdot t)$*, the total degree of the complex nodes in any bag is* $\Delta^* = O(1)$*, and the treewidth is at most* $\mathrm{tw} \leq 2k + O(1)$*.*

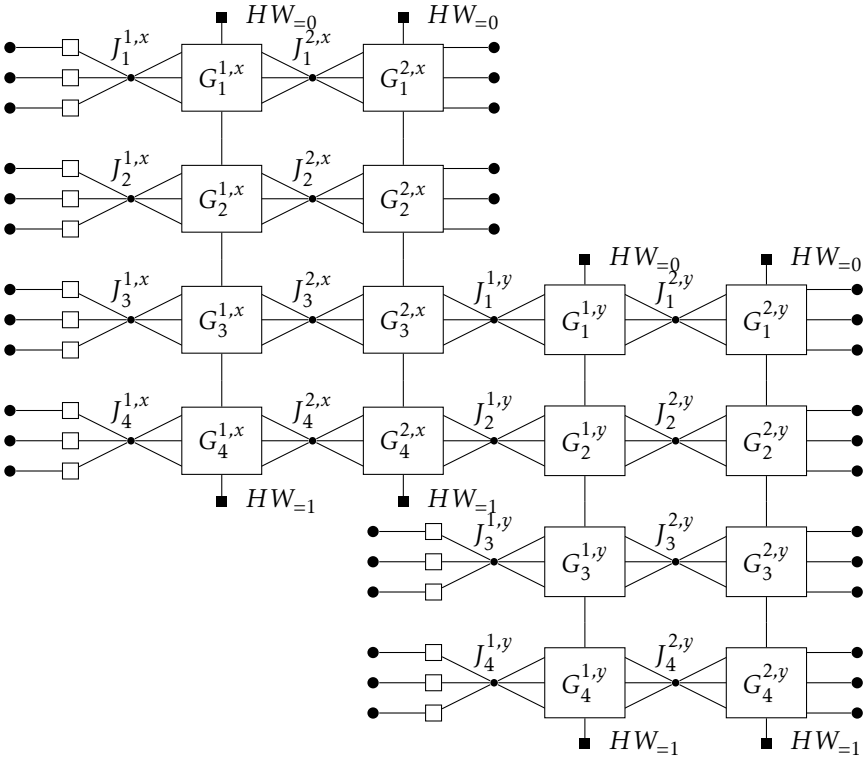


Figure 7.3: An example of two attached gridblocks, for $m = 2$ and $k = 4$. Here U_3^x coincides with U_1^y and U_4^x coincides with U_2^y.

Proof. We will closely follow the proof of [128, Lemma 7.14], which gives a reduction from #Multicolored Hitting Set to #AntiFactor$_1^{\mathcal{R}}$. The general idea is to perform

this construction for each pair of neighboring instances and then chaining them together at the shared elements. Figure 7.3 depicts an example of the construction. We advise the reader to consult this figure before reading the construction in more detail.

Given an instance of #Tree-Chained Multicolored Hitting Set, we refer to the parts of the tree structure as nodes, with r as root. Let $U_1^x, \ldots, U_k^x$ be the universes in node x from which we chose one element each. Also let $A_1^x, \ldots, A_m^x$ be the sets we want to hit that have elements in x and its parent node.

Let x be a node in the tree structure. Suppose x has a unique child y. With the universes ordered as $U_1^x, \ldots, U_k^x, U_1^y, \ldots, U_k^y$, we perform the construction from [128, Lemma 7.14] with the sets $A_1^y, \ldots, A_m^y$ to receive a gridblock B_y. We will adopt the convention of adding 'y' as a superscript to any part of B_y to distinguish it from similar parts from other gridblocks. We then attach B_y to B_x as follows. For each universe U_i^x that that is shared with y, let j be such that U_i^x appears in the list of universes of y as U_j^y. We now contract the dangling vertices at $F_i^{m,x}$ into $J_j^{1,y}$ and remove the monotonicity gadgets at $J_j^{1,y}$.

Now suppose x has two children y_1, y_2. We first duplicate the choice of elements at node x, by creating a block B'_x as follows. For $U_1^x, \ldots, U_k^x$ we create copies $V_1^x, \ldots, V_k^x$ and use the convention that for an element $u \in U_i^x$ we refer to its copy as $u' \in V_i^x$. We now create the block B'_x by performing the construction from [128, Lemma 7.14] for the sets $U_1^x, \ldots, U_k^x, V_1^x, \ldots, V_k^x$, where we want to hit the sets $\{u\} \cup (V_i^x \setminus \{u'\})$ for all $i \in [k]$ and $u \in U_i^x$. We then create the blocks B_{y_1} and B_{y_2} in the same way as before. We refer to the block F_i^j corresponding to U_i^x as $F_{i,U}^{j,x}$ and to the block F_i^j corresponding t V_i^x as $F_{i,U}^{j,x}$. We connect B_{y_1} to B'_x by contracting the dangling vertices at $F_{i,U}^{m,x}$ into J_j^{1,y_1}, where j is such that U_i^x appears in the list of universes of y_1 as U_j^y. We connect B_{y_2} to B'_x by contracting the dangling vertices at $F_{i,V}^{m,x}$ into J_j^{1,y_2}, where j is such that U_i^x appears in the list of universes of y_2 as U_j^y. Again we remove the extra monotonicity gadgets at J_i^{1,y_1} and J_i^{1,y_2}.

Correctness Note that by the proof of [128, Lemma 7.14] each local solution for a block B_x corresponds one-to-one to a solution of its corresponding hitting set instance. Thus we only need to show that a global solution consists of local solutions that are consistent with one another and any set of pairwise consistent local solutions forms a valid global solution.

As seen in the original proof, the F_i^j gadgets have a monotone in-/output and and the information about the selected element is transferred consistently, from the left side to the right side of the gadget. This means that the first ℓ in-/out-edges are selected for some ℓ. Furthermore for any t, the t-th in-edge is selected if and only if the t-th out edge is selected.

Let x be a node with a unique child y. A block $F_i^{m,x}$ has the first ℓ dangling edges selected if and only if the corresponding $J_j^{1,y}$ in B_y has the first ℓ incoming edges selected. By the functionality of $F_j^{1,y}$ in B_y and the proof of [128, Lemma 7.14] we

find that each block in B_y has the first ℓ incoming and outgoing edges selected if and only if $F_i^{m,x}$ in B_x has the first ℓ dangling edges selected. We conclude that the local solutions of B_x and B_y must be consistent. Furthermore we conclude that that if the local solutions of B_x and B_y are consistent, then they can be glued together.

Let x be a node with two children y_1, y_2. Node that, by definition, in the corresponding hitting set instance we choose exactly one $u \in U_i^x$ and one $u' \in V_i^x$ for each i. Furthermore if we choose u', then we hit all sets $\{v\} \cup (V_i^x \setminus \{v'\})$ for $v' \neq u'$. Thus we must choose u to hit the set $\{u\} \cup (V_i^x \setminus \{u'\})$. We find that the choice of element is successfully copied. By the same reasoning as before we find that this choice gets consistently transferred to B_{y_1} and B_{y_2}. Furthermore if B_{y_1} and B_{y_2} have consistent solutions, they translate into choices of u (and u') that give a valid solution to the hitting set instance at x, since they make the same choices. This means that consistent solutions of B_{y_1}, B_{y_2} and B_x can be glued together.

Bounded treewidth The proof of [128, Lemma 7.14] provides a path decomposition for a block B_x, which starts with the bag $\{J_{1,x}^1, \ldots, J_{k,x}^1\}$. If we contract the dangling vertices at the $G_i^{m,x}$ and then follow the construction for the path decomposition, it ends with the bag $\{J_{1,y}^1, \ldots, J_{k,y}^1\}$ corresponding to the child y of x. If x is a join node, the start and end bags are $\{J_{1,x}^{1,U}, \ldots, J_{k,x}^{1,U}, J_{1,x}^{1,V}, \ldots, J_{k,x}^{1,V}\}$ and $\{J_{1,y_1}^1, \ldots, J_{k,y_1}^1, J_{1,y_2}^1, \ldots, J_{k,y_2}^1\}$, corresponding to the two copies of the universe, and the two children y_1 and y_2 of x.

We then connect these path decompositions to one another at the shared start/end bags to find the desired tree decomposition.

Logspace We first note that the gadgets F_i^j have bounded pathwidth and that the individual parts of the gadget are only dependent on the hitting set instance and not on the rest of the graph. We find that we can construct the gadgets one bag at a time, which requires only $O(f(\mathrm{tw}))$ space for the construction of the bags and $O(\log(n))$ space for an index to indicate which bag we are constructing. To produce a specific bag in the tree decomposition, we simply find the path from the root bag to that bag and construct all bag on that path. □

Note that the same construction, but without the join blocks gives a reduction from #Chained Multicolored Hitting Set to #AntiFactor$_1^{\mathcal{R}}$ with relation weights $1, -1$ and $2 - n$, but with pathwidth bounded by $k + O(1)$.

Lemma 7.6.4. *There is an FPT-time many-one parameterized logspace reduction from #Chained Multicolored Hitting Set to relation-weighted #AntiFactor$_1^{\mathcal{R}}$ with relation weights $1, -1$ and $2 - n$. Let n be the size of the given universes, m the number of sets between adjacent nodes in the tree structure and t the number of nodes in the tree structure. The size of the produced instance is bounded by $O(k \cdot n \cdot m \cdot t)$, the total degree of the complex nodes in any bag is $\Delta^* = O(1)$, and the pathwidth is at most* $\mathrm{pw} \leq k + O(1)$.

Proof. We perform the same construction as in the proof of Lemma 7.6.3, leaving out the join blocks. Correctness and logarithmic space usage follow from the same argument. To prove bounded pathwidth we also use the same argument as used for bounded treewidth, but again ignoring the join nodes, thus finding a bound of $k + O(1)$, instead of $2k + O(1)$. □

We now turn our attention to the reduction from #AntiFactor$_1^{\mathcal{R}}$ to #Antifactor$_1$. We again follow the reduction by Marx et al. [128], which consists of a chain of smaller reductions. Most of these small reductions can be performed using logarithmic space. The following lemma allows us to adapt these reductions to our needs, while only examining parts of their proofs.

Lemma 7.6.5. *Let (r, g, h) be a parameterized reduction from counting problem Π to counting problem Γ. Let $I \in \mathcal{I}_\Pi$ and $n = |I|$. Suppose that (r, f, g) produces $r(I)$ by performing $n^{O(1)}$ many modifications to I. Furthermore suppose that each modification can be performed in $n^{O(1)}$ time, only affects an amount of space of $O(g(k_\Pi(I)) + \log(n))$ and doesn't affect a previously modified part of the instance. Then (r, g, h) is also a parameterized logspace reduction.*

Proof. We need to show that the reduction can be performed using only $O(g(k_\Pi(x)) + \log(n))$ space. We first note that, since an XNLP machine requires read only access to the input tape, we must be able to produce an arbitrary bit in the input tape. We do so by performing the given reduction one modification at a time, until we modify the desired bit. If we reach the end of the reduction before this happens, we simply output the input bit at this location.

By assumption we have enough memory to perform the individual modifications and to keep track of an index that indicates our place in the reduction. Furthermore, by assumption the reduction never modifies the same section twice and thus we know that once a section has been modified, we can safely output from that section. □

While Lemma 7.6.5 lets us easily adapt most of the reductions in the chain, there are a few that require a bit more care. We will handle these first and deal with the remaining reductions in Lemma 7.6.8.

The reductions in question require interpolation of polynomials. As shown in Lemma 2.7.2 this can be done using logarithmic space. We will handle these two steps now and handle the rest of the chain afterwards.

Lemma 7.6.6. *Given some edge weighted instance of #AntiFactor$_1^{\mathcal{R}}$, that uses the weights -1, $2-n$ and $\frac{1}{2}$, we can find a polynomial number of instances of #AntiFactor$_1^{\mathcal{R}}$ without edge weights, such that solving those instances is equivalent to solving the edge weighted instance. We can do so using $O(\log n)$ space. The only complex nodes we need to introduce for this are $HW_{=1}$ nodes. The size of the instances is at most an additive factor of $O(n^2 \log^2 n)$ bigger and the treewidth, pathwidth and Δ^* are at most a constant additive facter bigger than that of the original instance.*

Proof. We closely follow the proof of [128, Lemma 7.11] (or similarly [127, Lemma 7.6]). We first replace the edge-weight we want to remove by a variable x, which turns the solution of the instance into a polynomial. Using the construction from [128, Lemma 7.11] we can evaluate this polynomial at sufficiently many points. Note that the construction essentially replaces each weighted edge by a subdivision of a ladder shape, which has pathwidth at most 4 and size $O(\log^2 n)$ and can thus be constructed using $O(\log n)$ space. By Lemma 2.7.2 we can then compute the polynomial at the point corresponding to the weight we wish to remove. □

For this next lemma we use the notation $w[x,y]$, which indicates a unary signature (i.e. on a vertex with degree 1) which has weight y if the only dangling edge is selected and x otherwise. Note that in the notation of [128], $w[1,1]$ and $w[0,1]$ correspond to $HW^{(1)}_{\in\{0,1\}}$ and $HW^{(1)}_{=1}$ respectively.

Lemma 7.6.7. *There is a polynomial time Turing reduction from* Holant $(HW_{\in\overline{Y}}, w[1,1], w[-1,1], w[0,1])$ *to* #AntiFactor$_1$*, using* $O(\log n)$ *space. The reduction creates instances of tree or pathwidth at most that of the original instance and whose size is at most an additive factor of* $O(\Delta^* n^2)$ *bigger.*

Proof. We closely follow the proof of [128, Lemma 7.21]. We first replace the weight we want to remove by a variable x, which turns the solution of the instance into a polynomial. Using the construction from [128, Lemma 7.21] we can evaluate this polynomial at arbitrarily many points. Note that the construction replaces the $w[x,y]$ nodes with a gadget consisting of a long path with d pendant vertices attached to each vertex on the path, where d is the forbidden degree from the instance. These gadgets have pathwidth 1 and, since we may assume that $d \leq n$, they have size $O(n^2)$. They can thus be constructed using $O(\log n)$ space. By Lemma 2.7.2 we can then compute the polynomial at the point corresponding to the weight we wish to remove. □

Lemma 7.6.8. *There is some computable function f and a polynomial time Turing reduction from relation-weighted* #AntiFactor$^{\mathcal{R}}_1$ *with relation weights* $1, -1$ *and* $2-n$ *and total degree of the complex nodes in any bag is at most* Δ^**, to* #AntiFactor$_1$*, using* $O(f(\mathrm{tw}) + \log n)$ *space. That the size of produced instances is at most* $n^{O(1)} + f(\Delta^*)$ *and the treewidth and pathwidth are at most* $\mathrm{tw} + f(\Delta^*)$ *and* $\mathrm{pw} + f(\Delta^*)$ *respectively.*

Proof. For clarity we will adopt the notation and phrasing used by the lemmas we refer to. Since those lemmas are more general than we need, we may set $X = Y = \{d\}$ for d the integer given in the #AntiFactor$^{\mathcal{R}}_1$ instance.

We first apply the reduction of [128, Lemma 7.10] to receive an instance of edge-weighted #AntiFactor$^{\mathcal{R}}_1$ with edge weights $1, -1$ and $2-n$. Note that the reduction consists entirely of modifying relations and adding degree 1 vertices. These modifications can all clearly be made in polynomial time and in logarithmic space. By Lemma 7.6.5 we find that this reduction is a pl-reduction. This step increases the size by a constant multiplicative factor and the treewidth, pathwidth and Δ^* by a constant additive factor.

Using Lemma 7.6.6 we find a pl-reduction to unweighted #AntiFactor$^{\mathcal{R}}_1$. This step increases the size by an additive factor of $O(n^2 \log^2 n)$ and the treewidth, pathwidth and Δ^* by a constant additive factor.

Next, we will follow the chain of reductions in [128, Lemma 7.2]. We will state the intermediate problems as Holant problems.

The first step is to use [127, Lemmas 7.5] to reduce to Holant $(HW_{\in\overline{Y}}, HW_{=1})$ with edge weights. [127, Lemma 7.5] only introduces a polynomial number of constant size[4] gadgets and modifies a polynomial number of relations. It introduces

[4] The size depends on Δ^*.

edge weights 1, $\frac{1}{2}$ and -1. By Lemma 7.6.5 we find that this reduction is a pl-reduction. This step increases the size by an additive factor of $O((\Delta^*)^2 2^{\Delta^*})$, the treewidth and pathwidth by an additive factor of $O(\Delta^* 2^{\Delta^*})$ and Δ^* by an additive factor of $O(\Delta^* 2^{2\Delta^*})$. Again we remove the edge-weights using Lemma 7.6.6, increasing the size by an additive factor of $O(n^2 \log^2 n)$ and the treewidth, pathwidth and Δ^* by a constant additive factor.

The next step is to use [128, Lemmas 7.19] (case 2 of [128, Lemma 7.17]) to reduce to Holant $(HW_{\in\overline{Y}}, w[1,1], w[-1,1], w[0,1])$. [128, Lemma 7.19] only creates a star-shaped gadget with complex nodes as leaves. Note that we can construct this gadget one leaf at a time, by only saving the central gadget and the current leaf to memory, thus only using log-space. By Lemma 7.6.5 we find that this is a pl-reduction. This step increases the size by an additive factor of $O(\Delta^* n)$, the treewidth and pathwidth by an additive constant and Δ^* by an additive factor of $O(\Delta^* n)$.

Finally we apply Lemma 7.6.7 to find an instance of #AntiFactor$_1$. This step increases the size by an additive factor of $O(\Delta^* n^2)$ and does not increase the treewidth or pathwidth. □

7.7 Conclusion

In this chapter we have shown various problems to be #XLP or #XALP complete. Many of these problems were already known to be #W[t]-hard for some t, but without full completeness it was unclear whether these hardness results fully captured the difficulty of the problems. Our results settle this question by providing classes that fully capture the difficulty of these problems.

The introduction of these new classes also raises new questions. For instance, it is unclear whether #XLP and #XALP are truly distinct classes. We know that #XLP $\subseteq$ #XALP and intuitively one would expect #XALP-complete problems to be substantially more difficult that #XLP-complete problems, but currently we do not have any strong evidence to support this intuition. It would be interesting to have a better understanding for this relation, especially when compared to the non-parameterized setting. There the class #P is defined in terms of the class NP, but we never speak of #coNP, because it turns out that this class is equivalent to #P. Indeed, if one can count the number of correct solutions, one can typically also count the number of incorrect solutions (usually by subtracting the former from the total number of potential solutions). The class coNP, in terms of Turing machines, allows for a single co-non-deterministic step. The class XALP allows for a number of alternations between non-deterministic and co-non-deterministic steps. If #XLP and #XALP are indeed distinct, then these alternations add substantially more power to the machine than just a single co-non-deterministic step, even in the counting regime.

One could also ask how #XLP and #XALP relate to their decision counterparts XNLP and XALP. While inclusions between these two sets of classes are not possible for syntactical reasons (one is a class of counting problems and the other a class of decision problems), they can still be compared in terms of computational complexity. For instance #XLP can be said to be more difficult than XNLP, as solving the counting version of a problem also solves the decision version. Similarly #XALP can be said to be more difficult than XALP. One would then expect #XALP to also be

more difficult than XNLP, but it is unclear how #XLP and XALP relate. Is one more difficult than the other, or are they incomparable?

Both of these questions can be viewed through the lens of Toda's Theorem [146], which intuitively states that the class PP is as hard as the entire polynomial hierarchy (PH). This implies that #P is also as hard as PH and in particular as hard as NP and coNP. We find ourselves in a similar situation, with the counting versions of some non-deterministic classes and thus one might wonder if there is some analogue of Toda's theorem for this setting. This question would also be relevant to the study of XLPP [19], since Toda's theorem applies to both #P and PP.

8 Epilogue

The Sultan asked Solomon for a
Signet motto, that should hold
good for Adversity or Prosperity.
Solomon gave him,
"This too shall pass."

Persian fable

In this thesis we have studied various parameterized problems, most of which have been counting problems. We have gained new insight into these problems, in the form of hardness results and fine-grained bounds. In many cases the upper and lower bounds match, in other words, we have presented a number of algorithms that are optimal, either in a fine-grained sense or in terms of hardness.

Determining the Parameterized Complexity of the Tutte Polynomial

In Part II we have given a number of reductions and algorithms for many different cases of computing the Tutte polynomial, parameterized by cutwidth, pathwidth and treewidth. This has resulted in the following extensive and nearly complete complexity classification for this problem in Chapter 5: the Tutte polynomial can be computed in

- in polynomial time at the known easy points,
- in $\mathrm{tw}^{O(\mathrm{tw})}\, n^{O(1)}$ time but not in $\mathrm{ctw}^{o(\mathrm{ctw})}\, n^{O(1)}$ time at integer points for which $(x-1)(y-1) < 0$ or $x = 1$,
- in $q^{\mathrm{tw}} n^{O(1)}$ time but not in $2^{o(\mathrm{ctw})}$ at integer points for which $(x-1)(y-1) = q > 1$ or $y = 1$ (and for many points not even in $r^{\mathrm{ctw}} n^{O(1)}$ time for some constants $r < q$).

As far as we are aware and at the time of writing, no parameterized analysis of this kind has been done for the Tutte polynomial. The (nearly) matching upper and lower bounds for tree- and cutwidth indicate that the cost of using a potentially larger parameter like cutwidth is not balanced out by any substantial benefits. This is noteworthy since for related problems like computing the chromatic number of a graph there exists a $2^{\mathrm{ctw}} n^{O(1)}$ time algorithm, but any $\mathrm{pw}^{o(\mathrm{pw})}\, n^{O(1)}$ time algorithm would contradict the ETH [124]. Natural open questions that arise whether these results can be extended to non-integer points, what such classifications look like

for other parameters and (perhaps less obviously) what the complexity is for the modular counting version of the problem. For the latter, some research has been done [5, 96], but a full classification has not been found. We see all of this as evidence that the Tutte polynomial forms an interesting and fruitful framework for studying problems within (parameterized) complexity.

In Chapters 3 and 4 we have used the rank-bases approach, which we have further developed to work in a counting setting, to determine the complexity of the special cases of counting colorings, connected edgesets and forest. The results form Chapter 4 provide a particularly interesting case of computing the Tutte polynomial, as the curve associated with counting forests forms an asymmetry in an otherwise symmetric complexity classification. In particular, this curve can be computed in $4^{\mathrm{pw}}n^{O(1)}$ or $64^{\mathrm{tw}}n^{O(1)}$, while its mirror image cannot be computed in time $\mathrm{ctw}^{o(\mathrm{ctw})}n^{O(1)}$. The techniques we use could potentially be applicable to other related problems like counting Feedback Vertex Sets or spanning trees with a fixed number of components. For the latter a $\mathrm{tw}^{O(\mathrm{tw})}n^{O(1)}$ time algorithm is known [136], but we suspect our techniques could improve this to $2^{O(\mathrm{tw})}n^{O(1)}$.

Parameterized Logarithmic Space Complexity

In Chapter 3 we have given bounds for counting (list) colorings and connected edgesets that even hold for modular counting. Our results show that the modulus can influence the complexity of the problem in interesting ways, allowing for slightly faster algorithms when the modulus p divides $q-1$, where q is the number of colors. Our results trivially extend to the non-modular setting, where they show that counting colorings is significantly harder than the associated decision problem, when parameterized by cutwidth. Namely, for the latter there exists a $2^{\mathrm{ctw}}n^{O(1)}$ time algorithm [109] for any q, while we can exclude $(q-\varepsilon)^{\mathrm{ctw}}n^{O(1)}$ time algorithms for the counting version.

In Part III we have studied the parameterized classes XNLP, XALP, #XLP and #XALP. In Chapter 6 we have studied the Integer Multicommodity Flow problem and given algorithms and hardness results for various variations on the problem, including some XNLP- and XALP-completeness results, see Figure 8.1. Any XP algorithms for the XNLP- and XALP-complete problems are unlikely to only use $O(f(k)n^{O(1)})$ space by the Slice-wise Polynomial Space Conjecture (see Conjecture 2.5.5). We note that there are still some open cases left like Undirected Integer 2-Commodity Flow parameterized by pathwidth, which we suspect to be NP-complete. Furthermore, we only have an approximation algorithm for Integer 2-Commodity Flow parameterized by vertex cover number, but we suspect there is an FPT-time exact algorithm.

In Chapter 7 we have connected the theme of Part III to the general theme of the thesis, by introducing the classes #XLP and #XALP, i.e. the counting versions of XNLP and XALP. For these classes we have given canonical hardness results, for the newly introduced problems of #SAD Branchings and #Path-Like SAD Branchings. Additionally, we have given a chain of other hardness results, some of which give #XLP- or #XALP-completeness for problems which were previously only known to be W[t]-hard for some t. The introduction of these new classes raises the question

Parameter	unary capacities	binary capacities
pathwidth	XNLP-complete	para-NP-complete
treewidth	XALP-complete	para-NP-complete
weighted tree partition width	FPT (1)	FPT (1)
vertex cover	(2); in XP	(2); open

Table 8.1: Overview of our results for Integer 2-Commodity Flow. para-NP-complete means NP-complete for fixed value of parameter. (1) Capacities of arcs inside bags can be arbitrary, capacities of arcs between bags are bounded by weighted tree partition width. (2) Approximation, see Theorem 6.1.11; conjectured in FPT. For the undirected case, the same results hold, except that for the para-NP-completeness for the parameters pathwidth and treewidth, we need a third commodity.

how they relate to both each other and to existing classes. We know that #XLP $\subseteq$ #XALP, but are they actually distinct or is this inclusion really an equality? Furthermore we do not have a clear understanding of how #XLP and #XALP relate to other parameterized classes. Clearly #XLP and #XALP are both more difficult than XNLP and #XALP is also more difficult than XALP, but how does #XLP relate to XALP? And how do both relate to the classes XLPP and XP? We believe that some parameterized version of Toda's theorem [146] could be an interesting goal for further research in this direction, as it would likely shed light on a number of these questions.

Final Remarks

In conclusion, we have studied counting problems in a parameterized setting and linked this to the relatively new study of the classes XNLP and XALP. We have found that the rank-based approach lends itself well to counting problems and it can even be argued that the method makes more intuitive sense in the setting of counting problems, than that of decision problems.

We suspect that the study of the Tutte polynomial, in a parameterized complexity setting, has a lot of potential for further results and structural insights. In particular we believe that the framework of our approach should be applicable to other parameters and it could perhaps even be adapted into a broader meta theorem for a large class of (width) parameters. Additionally we suspect that the study of modular computation of the Tutte polynomial could significantly deepen our understanding of the Tutte polynomial in general. The existing proofs for tractability of the easy points are somewhat unsatisfying, as they are somewhat ad-hoc and lack an intuitive explanation for their tractability.

Furthermore we are hopeful that the classes of #XLP and #XALP will form a natural home for many parameterized counting problems without completeness results, in the same way that XNLP and XALP do for many decision problems. As previously mentioned there are still many open questions about their place in the broader complexity landscape and resolving (some of) these questions would represent a significant step forward in parameterized counting complexity.

Bibliography

[1] Faisal N. Abu-Khzam, Shouwei Li, Christine Markarian, Friedhelm Meyer auf der Heide, and Pavel Podlipyan. Efficient parallel algorithms for parameterized problems. *Theoretical Computer Science*, 786:2–12, 2019.

[2] Akanksha Agrawal and M. S. Ramanujan. On the parameterized complexity of Clique Elimination Distance. In *Proceedings of the 15th International Symposium on Parameterized and Exact Computation, IPEC 2020*, volume 180 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

[3] Martin Aigner and Günter M. Ziegler. Bertrand's postulate. In *Proofs from THE BOOK*, pages 7–12. Springer, 2001.

[4] Artur Andrzejak. An algorithm for the Tutte polynomials of graphs of bounded treewidth. *Discrete Mathematics*, 190(1-3):39–54, 1998.

[5] James Douglas Annan. *The Complexity of Counting Problems*. PhD thesis, University of Oxford, 1994.

[6] K. Appel and W. Haken. Every planar map is four colorable. Part I: Discharging. *Illinois Journal of Mathematics*, 21(3):429 – 490, 1977.

[7] K. Appel, W. Haken, and J. Koch. Every planar map is four colorable. Part II: Reducibility. *Illinois Journal of Mathematics*, 21(3):491 – 567, 1977.

[8] Ilia Averbouch, Benny Godlin, and J.A. Makowsky. An extension of the bivariate chromatic polynomial. *European Journal of Combinatorics*, 31(1):1–17, 2010.

[9] Cynthia Barnhart, Niranjan Krishnan, and Pamela H. Vance. Multicommodity Flow problems. In *Encyclopedia of Optimization*, pages 2354–2362. Springer, 2009.

[10] Rodney J. Baxter. *Exactly Solved Models in Statistical Mechanics*. Academic Press, 1982.

[11] Richard Ernest Bellman. *An Introduction to the Theory of Dynamic Programming*. RAND Corporation, 1953.

[12] Umberto Bertelè and Francesco Brioschi. *Nonserial Dynamic Programming*. Academic Press, 1972.

[13] Andreas Björklund and Thore Husfeldt. Inclusion-exclusion based algorithms for Graph Colouring. In *Proceedings of the Electronic Colloquium on Computational Complexity, ECCC 2006*, volume 13 of *Lecture Notes in Computer Science*. Sringer, 2006.

[14] Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Computing the Tutte polynomial in vertex-exponential time. In *Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008*, pages 677–686. Institute of Electrical and Electronics Engineers, 2008.

[15] Andreas Björklund and Petteri Kaski. The fine-grained complexity of computing the Tutte polynomial of a linear matroid. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021*, pages 2333–2345. Society for Industrial and Applied Mathematics, 2021.

[16] Hans Bodlaender, Rodney Downey, Michael Fellows, Michael Hallett, and Todd Wareham. Parameterized complexity analysis in computational biology. *Computer Applications in the Biosciences*, 11:49–57, 1995.

[17] 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 *Lecture Notes in Computer Science*, pages 84–97. Springer, 2022.

[18] Hans L. Bodlaender, Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Deterministic single exponential time algorithms for connectivity problems parameterized by treewidth. In *Proceedings of the 40th International Colloquium on Automata, Languages and Programming, ICALP 2013*, volume 243 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 86–111. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015.

[19] Hans L. Bodlaender, Nils Donselaar, and Johan Kwisthout. Parameterized completeness results for Bayesian inference. In *Proceedings of the 11th International Conference on Probabilistic Graphical Models, PGM 2022*, volume 186 of *Proceedings of Machine Learning Research*, pages 145–156. Proceedings of Machine Learning Research, 2022.

[20] Hans L. Bodlaender, Carla Groenland, and Hugo Jacob. On the parameterized complexity of computing tree-partitions. In *Proceedings of the 17th International Symposium on Parameterized and Exact Computation, IPEC 2022*, volume 249 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:20. 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 17th International Symposium on Parameterized and Exact Computation, IPEC 2022*, volume 249 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[22] Hans L. Bodlaender, Carla Groenland, Hugo Jacob, Marcin Pilipczuk, and Michał 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 *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:17. 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 62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021*, pages 193–204. Institute of Electrical and Electronics Engineers, 2022.

[24] 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 *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[25] Hans L. Bodlaender and Marieke van der Wegen. Parameterized complexity of scheduling chains of jobs with delays. In *Proceedings of the 15th International Symposium on Parameterized and Exact Computation, IPEC 2020*, volume 180 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[26] Narek Bojikian, Vera Chekan, Falko Hegerfeld, and Stefan Kratsch. Tight bounds for connectivity problems parameterized by cutwidth. In *Proceedings of the 40th International Symposium on Theoretical Aspects of Computer Science, STACS 2023*, volume 254 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

[27] Édouard Bonnet, Eun Jung Kim, Stéphan Thomassé, and Rémi Watrigant. Twin-width I: Tractable FO model checking. *Journal of the ACM*, 69(1), 2021.

[28] Cornelius Brand, Holger Dell, and Marc Roth. Fine-grained dichotomies for the Tutte plane and Boolean #CSP. *Algorithmica*, 81(2):541–556, 2019.

[29] Thomas H Brylawski. The Tutte polynomial. In *Matroid Theory and Its Applications: Lectures Given at the Centro Internazionale Matematico Estivo*. Springer, 1980.

[30] Andrei Bulatov and Martin Grohe. The complexity of partition functions. *Theoretical Computer Science*, 348(2):148–186, 2005.

[31] Andrei A. Bulatov. The complexity of the Counting Constraint Satisfaction problem. *Journal of the ACM*, 60(5), 2013.

[32] Andrei A. Bulatov and Víctor Dalmau. Towards a dichotomy theorem for the Counting Constraint Satisfaction problem. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2003*, volume 205, pages 651–678. Institute of Electrical and Electronics Engineers, 2007.

[33] Jin-Yi Cai and Xi Chen. A decidable dichotomy theorem on directed graph homomorphisms with non-negative weights. In *Proceedings of the 51st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2010*, pages 437–446. Institute of Electrical and Electronics Engineers, 2010.

[34] Jin-Yi Cai, Xi Chen, and Pinyan Lu. Non-Negatively Weighted #CSP: An effective complexity dichotomy. In *Proceedings of the IEEE 26th Annual Conference on Computational Complexity, CCC 2011*, pages 45–54. Institute of Electrical and Electronics Engineers, 2011.

[35] Jin-Yi Cai, Xi Chen, and Pinyan Lu. Graph homomorphisms with complex values: A dichotomy theorem. *SIAM Journal on Computing*, 42(3):924–1029, 2013.

[36] Jin-Yi Cai and Pinyan Lu. Holographic algorithms: From art to science. *Journal of Computer and System Sciences*, 77(1):41–61, 2011.

[37] Jin-Yi Cai, Pinyan Lu, and Mingji Xia. Holographic algorithms by Fibonacci gates and holographic reductions for hardness. In *Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008*, pages 644–653. Institute of Electrical and Electronics Engineers, 2008.

[38] Jin-Yi Cai, Pinyan Lu, and Mingji Xia. Holant problems and counting CSP. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009*, page 715–724. Association for Computing Machinery, 2009.

[39] Jin-Yi Cai, Pinyan Lu, and Mingji Xia. The complexity of Complex Weighted Boolean #CSP. *Journal of Computer and System Sciences*, 80(1):217–236, 2014.

[40] Jin-Yi Cai, Pinyan Lu, and Mingji Xia. *Encyclopedia of Algorithms*, chapter Holographic Algorithms, pages 1–8. Springer, 2014.

[41] Chris Calabro, Russell Impagliazzo, Valentine Kabanets, and Ramamohan Paturi. The complexity of Unique k-SAT: An isolation lemma for k-CNFs. *Journal of Computer and System Sciences*, 74(3):386–393, 2008.

[42] Pafnuty Chebyshev. Mémoire sur les nombres premiers. *Journal de Mathématiques Pures et Appliquées, Série 1*, 1852.

[43] Yijia Chen, Marc Thurley, and Mark Weyer. Understanding the complexity of Induced Subgraph Isomorphisms. In *Proceedings of the 35th International Colloquium on Automata, Languages, and Programming, ICALP 2008*, Lecture Notes in Computer Science, pages 587–596. Springer, 2008.

[44] Rajesh Chitnis and Graham Cormode. Towards a theory of parameterized streaming algorithms. In *14th International Symposium on Parameterized and Exact Computation, IPEC 2019*, Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[45] Fan R. K. Chung. On the cutwidth and the topological bandwidth of a tree. *SIAM Journal on Algebraic Discrete Methods*, 6(2):268–277, 1985.

[46] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Fourth Edition*. MIT Press, 2022.

[47] Richard W. Cottle, B. Curtis Eaves, and Mukund N. Thapa. Dantzig, George B. (1914–2005). In *The New Palgrave Dictionary of Economics*, pages 1–14. Palgrave Macmillan UK, 2016.

[48] Nadia Creignou and Miki Hermann. Complexity of generalized satisfiability counting problems. *Information and Computation*, 125(1):1–12, 1996.

[49] Nadia Creignou, Sanjeev Khanna, and Madhu Sudan. *Complexity Classifications of Boolean Constraint Satisfaction Problems*. Society for Industrial and Applied Mathematics, 2001.

[50] Radu Curticapean. Parity separation: A scientifically proven method for permanent weight loss. In *Proceedings of the 43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016*, volume 55 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 47:1–47:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016.

[51] Radu Curticapean. Count on CFI graphs for #P-hardness. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2024*, pages 1854–1871. Society for Industrial and Applied Mathematics, 2024.

[52] Radu Curticapean, Holger Dell, and Dániel Marx. Homomorphisms are a good basis for counting small subgraphs. pages 210–223. Association for Computing Machinery, 2017.

[53] Radu Curticapean, Nathan Lindzey, and Jesper Nederlof. A tight lower bound for counting Hamiltonian cycles via matrix rank. In *Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 201*, pages 1080–1099. Society for Industrial and Applied Mathematics, 2018.

[54] Radu Curticapean and Dániel Marx. Complexity of counting subgraphs: Only the boundedness of the vertex-cover number counts. In *55th Annual Symposium on Foundations of Computer Science, FOCS 2014*, pages 130–139. Institute of Electrical and Electronics Engineers, 2014.

[55] Radu Curticapean and Dániel Marx. Tight conditional lower bounds for counting perfect matchings on graphs of bounded treewidth, cliquewidth, and genus. In *Proceedings of the 2016 Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 1650–1669. Society for Industrial and Applied Mathematics, 2015.

[56] Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.

[57] Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Fast Hamiltonicity checking via bases of perfect matchings. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing*, *STOC 2013*, page 301–310. Association for Computing Machinery, 2013.

[58] Marek Cygan, Jesper Nederlof, Marcin Pilipczuk, Michał Pilipczuk, Johan M. M. Van Rooij, and Jakub Onufry Wojtaszczyk. Solving connectivity problems parameterized by treewidth in single exponential time. *ACM Transactions on Algorithms*, 18(2), 2022.

[59] Víctor Dalmau and Peter Jonsson. The complexity of counting homomorphisms seen from the other side. *Theoretical Computer Science*, 329(1):315–323, 2004.

[60] G.C. Danielson and C. Lanczos. Some improvements in practical Fourier analysis and their application to x-ray scattering from liquids. *Journal of the Franklin Institute*, 233(4):365–380, 1942.

[61] Holger Dell, Thore Husfeldt, Dániel Marx, Nina Taslaman, and Martin Wahlén. Exponential time complexity of the permanent and the Tutte polynomial. *ACM Transactions on Algorithms*, 10(4):1–32, 2014.

[62] Rodney Downey and Michael Fellows. *Fundamentals of Parameterized Complexity*. Springer, 2013.

[63] Rodney G. Downey and Michael R. Fellows. Fixed parameter tractability and completeness. In *Proceedings of the 21st Manitoba Conference on Numerical Mathematics and Computing*, volume 87, pages 161–187, 1992.

[64] Feodor F. Dragan and Ekkehard Köhler. An approximation algorithm for the Tree t-Spanner problem on unweighted graphs via generalized chordal graphs. *Algorithmica*, 69(4):884–905, 2014.

[65] Stuart Dreyfus. Richard Bellman on the birth of dynamic programming. *Operations Research*, 50(1):48–51, 2002.

[66] Martin Dyer, Leslie Ann Goldberg, and Mark Jerrum. The complexity of Weighted Boolean #CSP. *SIAM Journal on Computing*, 38(5):1970–1986, 2009.

[67] Martin Dyer, Leslie Ann Goldberg, and Mike Paterson. On counting homomorphisms to directed acyclic graphs. *Journal of the ACM*, 54(6):27–es, 2007.

[68] Martin Dyer and Catherine Greenhill. The complexity of counting graph homomorphisms. *Random Structures & Algorithms*, 17(3-4):260–289, 2000.

[69] Martin E. Dyer and David M. Richerby. On the complexity of #CSP. In *Proceedings of the 42nd ACM Symposium on Theory of Computing*, *STOC 2010*, page 725–734. Association for Computing Machinery, 2010.

[70] Eduard Eiben, Robert Ganian, and Iyad Kanj. The parameterized complexity of Coordinated Motion Planning. In *Proceedings of the 39th International Symposium on Computational Geometry, SoCG 2023*, volume 258 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

[71] 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.

[72] J.A. Ellis-Monaghan and I. Moffatt. *Handbook of the Tutte Polynomial and Related Topics*. Monographs and Research Notes in Mathematics. C&H/CRC Press, 2022.

[73] Joanna A. Ellis-Monaghan and Criel Merino. *Structural Analysis of Complex Networks*, chapter Graph Polynomials and Their Applications I: The Tutte Polynomial, pages 219–255. Birkhäuser Boston, 2011.

[74] P. Erdös and P. 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.

[75] Leonhard Euler. *Commentarii Academiae scientiarum imperialis Petropolitanae*. Petropolis, Typis Academiae, 1736.

[76] Shimon Even, Alon Itai, and Adi Shamir. On the complexity of Timetable and Multicommodity Flow problems. *SIAM Journal on Computing*, 5(4):691–703, 1976.

[77] Andreas Emil Feldmann, Karthik C. S., Euiwoong Lee, and Pasin Manurangsi. A survey on approximation in parameterized complexity: Hardness and algorithms. *Algorithms*, 13(6), 2020.

[78] Michael R. Fellows, Danny Hermelin, Frances A. Rosamond, and Stéphane Vialette. On the parameterized complexity of multiple-interval graph problems. *Theoretical Computer Science*, 410(1):53–61, 2009.

[79] Krzysztof Fleszar, Matthias Mnich, and Joachim Spoerhase. New algorithms for Maximum Disjoint Paths based on tree-likeness. *Mathematical Programming*, 171(1-2):433–461, 2018.

[80] Jörg Flum and Martin Grohe. The parameterized complexity of counting problems. *SIAM Journal on Computing*, 33(4):892–922, 2004.

[81] Jacob Focke, Dániel Marx, and Paweł Rzażewski. Counting list homomorphisms from graphs of bounded treewidth: Tight complexity bounds. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2022*, pages 431–458. Society for Industrial and Applied Mathematics, 2022.

[82] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[83] Steven Fortune, John E. Hopcroft, and James Wyllie. The Directed Subgraph Homeomorphism problem. *Theoretical Computer Science*, 10:111–121, 1980.

[84] A. Frank and Éva Tardos. An application of simultaneous Diophantine approximation in combinatorial optimization. *Combinatorica*, 7(1):49–65, 1987.

[85] Nicholas Awde Fred James Hill. *A History of the Islamic World*. Hippocrene Books, 2003.

[86] Tobias Friedrich, Davis Issac, Nikhil Kumar, Nadym Mallek, and Ziena Zeif. A primal-dual algorithm for multicommodity flows and multicuts in treewidth-2 graphs. In *Proceedings of the 25th International Conference on Approximation Algorithms for Combinatorial Optimization Problems and 26th International Conference on Randomization and Computation*, *APPROX/RANDOM 2022*, volume 245 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 55:1–55:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[87] Tobias Friedrich, Davis Issac, Nikhil Kumar, Nadym Mallek, and Ziena Zeif. Approximate max-flow min-multicut theorem for graphs of bounded treewidth. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, *STOC 2023*, page 1325–1334. Association for Computing Machinery, 2023.

[88] Solomon Gandz. The sources of Al-Khowārizmī's algebra. *Osiris*, 1:263 – 277, 1936.

[89] Robert Ganian and Sebastian Ordyniak. The power of cut-based parameters for computing edge-disjoint paths. *Algorithmica*, 83(2):726–752, 2021.

[90] Robert Ganian, Sebastian Ordyniak, and M. S. Ramanujan. On structural parameterizations of the Edge Disjoint Paths problem. *Algorithmica*, 83(6):1605–1637, 2021.

[91] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[92] Delia Garijo, Andrew Goodall, and Jaroslav Nešetřil. Graph homomorphisms, the Tutte polynomial and "q-state Potts uniqueness". In *Proceedings of the European Conference on Combinatorics, Graph Theory and Applications*, *EuroComb 2009*, volume 34 of *Electronic Notes in Discrete Mathematics*, pages 231–236, 2009.

[93] Omer Giménez, Petr Hliněný, and Marc Noy. Computing the Tutte polynomial on graphs of bounded clique-width. In *Proceedings of the 31th International Workshop on Graph-Theoretic Concepts in Computer Science*, *WG 2005*, pages 59–68. Springer, 2005.

[94] Leslie Ann Goldberg, Martin Grohe, Mark Jerrum, and Marc Thurley. A complexity dichotomy for partition functions with mixed signs. *SIAM Journal on Computing*, 39(7):3336–3402, 2010.

[95] Leslie Ann Goldberg and Mark Jerrum. Inapproximability of the Tutte polynomial. *Information and Computation*, 206(7):908–929, 2008.

[96] A.J. Goodall. The Tutte polynomial modulo a prime. *Advances in Applied Mathematics*, 32(1):293–298, 2004.

[97] Catherine Greenhill. The complexity of counting colourings and independent sets in sparse graphs and hypergraphs. *Computational Complexity*, 9(1):52–72, 2000.

[98] Carla Groenland, Isja Mannens, Jesper Nederlof, Marta Piecyk, and Paweł Rzażewski. Towards tight bounds for the Graph Homomorphism problem parameterized by cutwidth via asymptotic matrix parameters. In *Proceedings of the 51th International Colloquium on Automata, Languages, and Programming, ICALP 2024*, Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.

[99] Carla Groenland, Isja Mannens, Jesper Nederlof, and Krisztina Szilágyi. Tight bounds for counting colorings and connected edge sets parameterized by cutwidth. In *Proceedings of the 39th International Symposium on Theoretical Aspects of Computer Science, STACS 2022*, Leibniz International Proceedings in Informatics (LIPIcs), pages 36:1–36:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

[100] Martin Grohe. The parameterized complexity of database queries. In *Proceedings of the 20th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2001*, page 82–92. Association for Computing Machinery, 2001.

[101] Jiong Guo, Falk Hüffner, and Rolf Niedermeier. A structural view on parameterizing problems: Distance from triviality. In *Proceedings of the 1st International Symposium on Parameterized and Exact Computation, IWPEC 2004*, Lecture Notes in Computer Science, pages 162–173. Springer, 2004.

[102] Yuri Gurevich, Larry Stockmeyer, and Uzi Vishkin. Solving NP-hard problems on graphs that are almost trees and an application to facility location problems. *Journal of the ACM*, 31(3):459–473, 1984.

[103] Muhammad ibn Musa al Khwarizmi. *Algorithmo de Numero Indorum (Latin translation)*. 8th century CE.

[104] Muhammad ibn Musa al Khwarizmi. *Al-Jabr*. ca 820.

[105] Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367–375, 2001.

[106] F. Jaeger, D. L. Vertigan, and D. J. A. Welsh. On the computational complexity of the Jones and Tutte polynomials. *Mathematical Proceedings of the Cambridge Philosophical Society*, 108(1):35–53, 1990.

[107] Lars Jaffke and Bart M. P. Jansen. Fine-grained parameterized complexity analysis of graph coloring problems. *Discrete Applied Mathematics*, 327:33–46, 2023.

[108] Bart M. P. Jansen and Jari J. H. de Kroon. FPT algorithms to compute the elimination distance to bipartite graphs and more. In *Proceedings of the 47th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2021*, pages 80–93. Springer, 2021.

[109] Bart MP Jansen and Jesper Nederlof. Computing the chromatic number using graph decompositions via matrix rank. *Theoretical Computer Science*, 795:520–539, 2019.

[110] Klaus Jansen, Stefan Kratsch, Dániel Marx, and Ildikó Schlotter. Bin packing with fixed number of bins revisited. *Journal of Computer and System Sciences*, 79(1):39–49, 2013.

[111] Mark Jerrum. Two-dimensional monomer-dimer systems are computationally intractable. *Journal of Statistical Physics*, 48(1-2):121–134, 1987.

[112] Mark Jerrum and Alistair Sinclair. Polynomial-time approximation algorithms for the Ising model. *SIAM Journal on Computing*, 22(5):1087–1116, 1993.

[113] George Karakostas. Faster approximation schemes for Fractional Multicommodity Flow problems. *ACM Transactions on Algorithms*, 4(1):13:1–13:17, 2008.

[114] Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations, 1972*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[115] P. W. Kasteleyn. The statistics of dimers on a lattice. *Physica D: Nonlinear Phenomena*, 27:281–298, 1961.

[116] P.W. Kasteleyn and F. Harary. *Graph Theory and Theoretical Physics*, chapter Graph theory and crystal physics, pages 43–110. Academic Press, 1967.

[117] Amirhossein Kazeminia and Andrei A. Bulatov. Counting homomorphisms modulo a prime number. In *Proceedings of the 44th International Symposium on Mathematical Foundations of Computer Science, MFCS 2019*, volume 138 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 59:1–59:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019.

[118] Ton Kloks. *Treewidth: Computations and Approximations*. Springer, 1994.

[119] Mikko Koivisto. An $O^*(2^n)$ algorithm for Graph Coloring and other partitioning problems via inclusion–exclusion. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2006*, pages 583–590. Institute of Electrical and Electronics Engineers, 2006.

[120] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer, 2000.

[121] John E. Krizan, Peter F. Barth, and M.L. Glasser. Phase transitions for the Ising model on the closed Cayley tree. *Physica A: Statistical Mechanics and its Applications*, 119(1):230–242, 1983.

[122] Richard E. Ladner. On the structure of polynomial time reducibility. *Journal of the ACM*, 22(1):155–171, 1975.

[123] Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. Known algorithms on graphs of bounded treewidth are probably optimal. *ACM Transactions on Algorithms*, 14(2):13:1–13:30, 2018.

[124] Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. Slightly superexponential parameterized problems. *SIAM Journal on Computing*, 47(3):675–702, 2018.

[125] J.A. Makowsky. Coloured Tutte polynomials and Kauffman brackets for graphs of bounded tree width. *Discrete Applied Mathematics*, 145(2):276–290, 2005.

[126] Isja Mannens and Jesper Nederlof. A fine-grained classification of the complexity of evaluating the Tutte polynomial on integer points parameterized by treewidth and cutwidth. In *Proceedings of the 31st Annual European Symposium on Algorithms, ESA 2023*, volume 274 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 82:1–82:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

[127] Dániel Marx, Govind S. Sankar, and Philipp Schepper. Degrees and gaps: Tight complexity results of General Factor problems parameterized by treewidth and cutwidth. In *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming, ICALP 2021*, volume 198 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 95:1–95:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

[128] Dániel Marx, Govind S. Sankar, and Philipp Schepper. Anti-factor is FPT parameterized by treewidth and list size (but counting is hard). In *Proceedings of the 17th International Symposium on Parameterized and Exact Computation, IPEC 2022*, volume 249 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

[129] Dániel Marx and Ildikó Schlotter. Obtaining a planar graph by vertex deletion. In *Proceedings of the 33th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2007*, pages 292–303. Springer, 2007.

[130] Neeldhara Misra and Piyush Rathi. The parameterized complexity of Dominating Set and friends revisited for structured graphs. In *Proceedings of the 14th International Computer Science Symposium in Russia, CSR 2019*, pages 299–310. Springer, 2019.

[131] Wendy Myrvold. Counting k-component forests of a graph. *Networks*, 22(7):647–652, 1992.

[132] Jesper Nederlof. Bipartite TSP in $O(1.9999^n)$ time, assuming quadratic time matrix multiplication. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020*, pages 40–53. Association for Computing Machinery, 2020.

[133] Takao Nishizeki, Jens Vygen, and Xiao Zhou. The Edge-Disjoint Paths Problem is NP-complete for series-parallel graphs. *Discrete Applied Mathematics*, 115(1-3):177–186, 2001.

[134] Steven D. Noble. Evaluating the Tutte polynomial for graphs of bounded tree-width. *Combinatorics, Probability and Computing*, 7(3):307–321, 1998.

[135] James G Oxley, Dominic JA Welsh, et al. The Tutte polynomial and percolation. *Graph Theory and Related Topics*, pages 329–339, 1979.

[136] Xin-Zhuang Chen Peng-Fei Wan. Computing the number of k-component spanning forests of a graph with bounded treewidth. *Journal of the Operations Research Society of China*, 7(2):385, 2019.

[137] Michal 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.

[138] R. B. Potts. Some generalized order-disorder transformations. *Mathematical Proceedings of the Cambridge Philosophical Society*, 48(1):106–109, 1952.

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

[140] Marc Roth, Johannes Schmitt, and Philip Wellnitz. Detecting and counting small subgraphs, and evaluating a parameterized Tutte polynomial: Lower bounds via toroidal grids and Cayley graph expanders. In *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming, ICALP 2021*, volume 198 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 108:1–108:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

[141] Detlef Seese. Tree-partite graphs and the complexity of algorithms. In *Proceedings of the 5th International Conference on Fundamentals of Computation Theory, FCT 1985*, volume 199 of *Lecture Notes in Computer Science*, pages 412–421. Springer, 1985.

[142] Sebastian Siebertz and Alexandre Vigny. Parameterized Distributed Complexity Theory: A logical approach. *arXiv:1903.00505*, 2021.

[143] Dirk J. Struik. *A Concise History of Mathematics*. Dover Publications, 1987.

[144] Lajos Takács. On Cayley's formula for counting forests. *Journal of Combinatorial Theory, Series A*, 53(2):321–323, 1990.

[145] Amira Tarek, Hagar Elsayed, Manar Rashad, Manar Hassan, and Passant el kafrawy. Dynamic programming applications: A suvrvey. In *Proceedings of the 2nd Novel Intelligent and Leading Emerging Sciences Conference, NILES 2020*, pages 380–385. Institute of Electrical and Electronics Engineers, 2020.

[146] Seinosuke Toda. PP is as hard as the polynomial-time hierarchy. *SIAM Journal on Computing*, 20(5):865–877, 1991.

[147] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1936.

[148] W. T. Tutte. A contribution to the theory of chromatic polynomials. *Canadian Journal of Mathematics*, 6:80–91, 1954.

[149] William T. Tutte. *Algebraic Theory of Graphs*. PhD thesis, University of Cambridge, 1948.

[150] W.T. Tutte. On dichromatic polynomials. *Journal of Combinatorial Theory*, 2(3):301–320, 1967.

[151] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[152] Leslie G. Valiant. Holographic algorithms. *SIAM Journal on Computing*, 37(5):1565–1594, 2008.

[153] L.G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979.

[154] D. L. Vertigan and D. J. A. Welsh. The computational complexity of the Tutte plane: The bipartite case. *Combinatorics, Probability and Computing*, 1(2):181–187, 1992.

[155] I-Lin Wang. Multicommodity network flows: A survey, part I: Applications and formulations. *International Journal of Operations Research*, 15(4):145–153, 2018.

[156] I-Lin Wang. Multicommodity network flows: A survey, part II: Solution methods. *International Journal of Operations Research*, 15(4):155–173, 2018.

[157] Robin James Wilson. *Four Colors Suffice: How the Map Problem was Solved*. Princeton University Press, 2014.

[158] F. Y. Wu. The Potts model. *Reviews of Modern Physics*, 54:235–268, 1982.

[159] F. Y. Wu. Potts model of magnetism (invited). *Journal of Applied Physics*, 55(6):2421–2425, 1984.

[160] Xiao Zhou, Syurei Tamura, and Takao Nishizeki. Finding edge-disjoint paths in partial k-trees. *Algorithmica*, 26(1):3–30, 2000.

Samenvatting

De meeste resultaten uit dit proefschrift hebben betrekking tot zogeheten *Grafen*. Een graaf is simpelweg een netwerk van punten en lijnen, maar er wordt binnen zowel de wiskunde als de informatica veel onderzoek gedaan naar deze objecten. Grafen kunnen gebruikt worden om allerlei soorten problemen te modelleren. Zo kan een treinnetwerk weergegeven worden als een graaf, maar ook een sociaal netwerk, of de bestanden structuur van een computer.

Ook problemen die op het eerste gezicht niets met netwerken te maken hebben kunnen door de lens van grafen gezien worden. Zo kun je het inplannen van lessen op een school als graaf tekenen, door voor elke les een punt te tekenen en twee lessen met een lijn te verbinden als deze niet tegelijk gepland kunnen worden. Het bepalen van een geldige planning is nu een zogenaamd *kleuringsprobleem* op de graaf. Om een geldige *graafkleuring* te vinden moeten we elk punt in de graaf een kleur geven, op zo'n manier dat twee punten met een lijn ertussen nooit dezelfde kleur hebben. In het voorbeeld over de planning kunnen we kleuren als tijdstippen zien en betekent dit dat twee lessen niet op hetzelfde moment gepland mogen worden als er een lijn tussen de lessen is. Laat dit nou precies zijn hoe we de graaf gedefinieerd hadden!

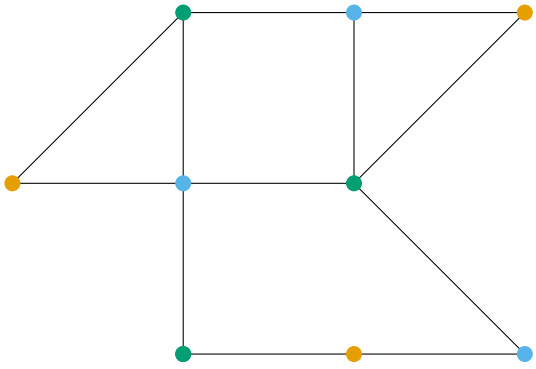


Figure 8.1: Een voorbeeld van een geldige graafkleuring.

Voor kleine grafen lukt het vaak nog wel om zo'n kleuring met de hand te vinden (als deze bestaat), maar voor een middelgrote school wordt dit al vrij snel erg veel werk. Het liefst laten we dit dan ook door een computer doen. Om dit soort problemen goed op te kunnen lossen, heeft een computer instructies nodig. Deze instructies worden *Algoritmen* genoemd. Bij het ontwerpen van algoritmen is het doel meestal om een zo snel mogelijk algoritme de vinden (al is soms het doel om

de hoeveelheid werkgeheugen die nodig is te beperken). We zeggen dat een probleem een hoge *complexiteit* heeft als er (waarschijnlijk) geen snelle algoritmen voor gevonden kunnen worden. De complexiteit van verschillende algoritmische problemen is het belangrijkste onderwerp van studie in dit proefschrift. In het bijzonder kijken we naar *geparameteriseerde* complexiteit, wat kort gezegd inhoud dat we extra structurele eigenschappen van het probleem, genaamd *parameters*, meenemen in onze analyse. Het type problemen waar we naar zullen kijken zijn zogenaamde *tel problemen*. Niet geheel verrassend vraagt een tel probleem om het aantal oplossingen van een standaard probleem. Zo zouden we in het voorbeeld van de school kunnen vragen naar het **aantal** mogelijke planningen, in plaats van één concrete planning.

Het Tutte Polynoom In de eerste helft van dit proefschrift onderzoeken we de complexiteit van het *Tutte polynoom*. Het Tutte polynoom is een voorbeeld van een graafpolynoom. Dit is een polynoom (dus iets als $x^2 - 2y + 3xy$) waarvan de vorm afhangt van de structuur van een gegeven graaf. Een heel simpel voorbeeld is '$n \cdot x$', waarbij 'n' het aantal punten in de graaf is.

Iedere graaf G heeft dus een Tutte polynoom, welke meestal aangeduid wordt als $T(G; x, y)$. Het Tutte polynoom van een graaf bevat veel informatie over de graaf, zo kan het aantal q-kleuringen van een graaf G bepaald worden door $T(G; 1-q, 0)$ uit te rekenen en kunnen we het aantal opspannende bossen[1] vinden door $T(G; 2, 1)$ uit te rekenen.

In dit proefschrift bepalen we eerst de complexiteit van het tellen van kleuringen en bossen, waarna we deze resultaten uitbreiden tot het berekenen van het Tutte polynoom op andere punten.

Complexiteitsklassen In de tweede helft van dit proefschrift onderzoeken we een aantal zogenaamde *complexiteitsklassen*. Een complexiteitsklasse is een grote collectie problemen die ongeveer even moeilijk op te lossen zijn, oftewel die een vergelijkbare complexiteit hebben[2]. Wij onderzoeken de klassen XNLP, XALP, #XLP en #XALP die specifiek voor geparameteriseerde problemen gedefinieerd zijn. XNLP en XALP zijn twee klassen waar recentelijk interesse in is ontstaan, omdat deze een natuurlijk thuis vormen voor een aantal problemen waarvan de precieze complexiteit voorheen nog onbekend was, zoals graafkleuring met bepaalde parameters[3]. Wij breiden dit onderzoek uit door de klassen #XLP en #XALP de introduceren. Deze klassen zijn de tel versies van XNLP en XALP. Het doel binnen de complexiteitstheorie is meestal om te laten zien dat een probleem *moeilijk* of *compleet* is voor een complexiteitsklasse. Een probleem is moeilijk voor een klasse als het minstens net zo moeilijk is als ieder willekeurig probleem in de klasse en het is compleet als het daarnaast ook zelf in de klasse zit. Problemen die XNLP-compleet zijn hebben meestal een #XLP-complete tel versie (idem voor XALP en #XALP), maar andersom is niet altijd het geval. Soms is de tel versie veel moeilijker dan het standaard probleem!

[1] Een bos is een substructuur in de graaf die geen cycli bevat.

[2] Iets preciezer: de problemen in een klasse hebben een zekere gedeelde bovengrens op hun complexiteit.

[3] In het bijzonder 'pathwidth', 'linear clique-width' en 'linear mim-width'

Curriculum Vitea

I count the spiders on the wall...
I count the cobwebs in the hall...
I count the candles on the shelf...
When I'm alone, I count myself!

Count von Count

Isja Mannens was born on 15 april 1996, in Almere, The Netherlands. In 2014 he received his VWO-diploma from Baken Park Lyceum in Almere. From 2014 until 2020 he studied Mathematics at the University of Amsterdam. He obtained his bachelor's degree in 2018 and his master's degree in 2020. After finishing his studies he started working as a PhD candidate at Utrecht University, where he researched parameterized graph problems, with a particular focus on counting problems. He was supervised by dr. Jesper Nederlof and prof. dr. Hans L. Bodlaender. During this time he won the best paper award at the 18th edition of the International Symposium on Parameterized and Exact Computation (IPEC), for his joint paper 'The Parameterised Complexity of Integer Multicommodity Flow'. He also spent six weeks at the IT University of Copenhagen for a research visit, where he was hosted by dr. Radu Curticapean. The work he did during this visit culminated in the results found in Chapter 7.

Acknowledgements

You know me, I think there ought to be a big old tree right there. And let's give him a friend. Everybody needs a friend.

Bob Ross

Any large project, PhD theses obviously included, require a tremendous amount of support from those around you. My experience has been no exception to this and as such, I would like to thank a number of people, in no particular order.

Mijn ouders **Marga en Marcel**. Jullie opvoeding heeft van mij een ware nachtmerrie gemaakt voor de andere ouders, maar het heeft mij ook ongetwijfeld gebracht waar ik nu ben. Op de vraag 'waarom?' kwam van jullie altijd een goed antwoord, zelfs als dat 'geen idee' was. Met het antwoord: 'omdat ik dat zeg', wat ik van andere ouders kreeg nam ik uiteraard dus geen genoegen. Deze nieuwsgierigheid (en enigszins antiautoritaire houding) houd ik tot de dag van vandaag bij me. Jullie hebben mij altijd de middelen geboden om me te ontwikkelen, maar ook de vrijheid en verantwoordelijkheid om mijn eigen middelen te creëren.

Mijn broer **Camiel**, de spraakzamere van de twee broers. Hoe jij vijf pagina's hebt weten te vullen met je dankwoord vind ik onbegrijpelijk, aangezien ik al moeite heb met twee. Als kind liep ik altijd in jouw voetsporen, zelfde muziek, gedeelde interesses, enzovoorts. Met jou samenwonen tijdens het begin van mijn studie heeft de overgang naar het uit-huis-wondende leven een stuk makkelijker gemaakt. Toen jij 6 jaar naar Stockholm ging stond ik er ineens alleen voor. Dat was moeilijk, maar heeft me ook de gelegenheid gegeven om uit te vinden wie ik ben als ik zelf alles moet regelen (en als ik ineens voor mezelf moet koken!). Ik ben wel heel blij dat je weer (een soort van) terug bent en kijk uit naar de rest van ons broer-schap.

Abigaël, Emile, Martijn, Martine, Sophia en Vincent. Als er iemand is die mij geaard houdt, dan zijn jullie dat. Ongeacht wat voor gekke dingen er in mijn (of jullie) leven gebeuren, ik kan altijd bij jullie terecht om het in perspectief te plaatsen. Waar iedereen ook is en hoe lang we elkaar ook niet zien, ik weet dat elke ontmoeting naadloos aansluit op de vorige.

Falco, Serge en Tim. Niet alleen hebben jullie mij door mijn studie heen gesleept, jullie geven mij ook broodnodige een uitlaat voor mijn meer 'nerdy' interesses. Ik ben blij dat jullie de moeite hebben gedaan om contact te houden, iets waar ik zelf niet altijd even goed in ben geweest.

De vele vrienden die ik de afgelopen twee jaar heb gemaakt in de klimaatbeweging, in het bijzonder bij **Milieudefensie Jong**. Jullie hebben mij onwijs veel onvergetelijke ervaringen bezorgd en mij een gemeenschap geboden waar ik een compleet andere kant van mezelf heb leren kennen. Voor de vele dingen die ik van jullie heb geleerd en het zelfvertrouwen wat ik met jullie heb opgedaan ben ik onwijs dankbaar.

De **Fencing Club Almere** en al haar leden. Bij FCA heb ik een doorzettingsvermogen opgebouwd, wat ongetwijfeld van pas is gekomen tijdens mijn PhD. Belangrijker waren echter de bekende gezichten die elke maandagavond voor mij klaar stonden. Er zijn buiten mijn familie weinig mensen die ik zo lang ken als mijn vrienden van FCA en dat neem ik niet voor lief.

All my other family and friends, who have supported me and encouraged me throughout my life.

Jesper. Je vorige PhD Céline schreef in haar dankwoord dat jouw volgende PhD blij mag zijn met jou als begeleider en ik kan dit (samen met Krisztina) zeker bevestigen. Je hebt me altijd de ruimte en vrijheid gegeven die ik nodig had, maar wist ook wanneer er een klein beetje druk nodig was om alles op de rails te houden. Ik heb altijd het gevoel gehad dat jij je inzette voor mijn ontwikkeling en ik heb nooit het gevoel gehad dat ik er alleen voor stond. Ik ben ervan overtuigd dat ik enorm veel geluk heb gehad met jou als begeleider. Bedankt voor alles.

My office mates **Krisztina and Sukanya** for costing me many (un)productive hours through the most delightful distractions. We clearly were the best office in the division, as evidenced by the many intruders and laughs. I wish you well in life and hope to stay in touch.

My many other **colleagues** in the Algorithms division for giving me a warm an welcoming place to learn and grow. I am sure that the many lunches, coffee breaks, boardgame evenings, escaperooms and bouldering sessions will stay with me for a long time.

The members of my **reading committee**, for spending the time to read and correct this thesis.

My many coauthors **Jesper, Céline, Krisztina, Carla, Marc, Paul, Linda, Jelle, Erik Jan, Sukanya, Hans, Radu, Lars and Paloma**, many of whom are also covered in other categories of gratitude. I quite literally could not have produced this thesis without you.

The many **students and staff** of Utrecht University who spoke out against injustice and stood up for what they knew is right, even when they were undermined and attacked by those that had a burden of care to them. *"Let it bring hope, let it be a tale."*