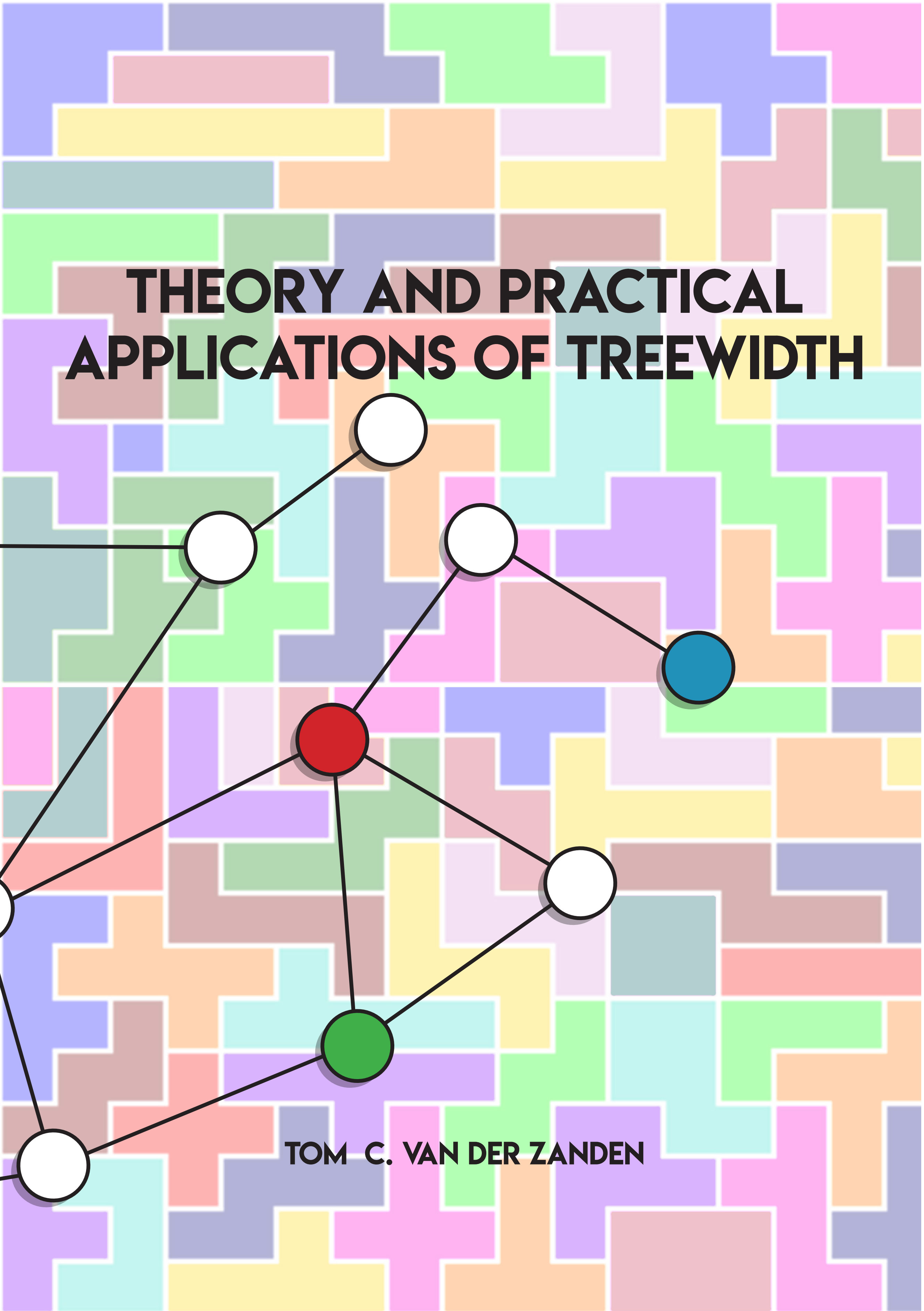
THEORY AND PRACTICAL APPLICATIONS OF TREEWIDTH



TOM C. VAN DER ZANDEN

Theory and Practical Applications of Treewidth

Tom C. van der Zanden

Tom C. van der Zanden
Theory and Practical Applications of Treewidth

ISBN: 978-90-393-7147-3

`tom@tomvanderzanden.nl`
`www.tomvanderzanden.nl`

Cover design and printing: Drukkerij Haveka, Alblasserdam

Theory and Practical Applications of Treewidth

Theorie en Praktische Toepassingen van Treewidth

(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de rector magnificus, prof.dr. H.R.B.M. Kummeling, ingevolge het besluit van het college voor promoties in het openbaar te verdedigen op woensdag 26 juni 2019 des middags te 2.30 uur

door

Tom Cornelis van der Zanden

geboren op 29 december 1992
te Nieuwegein

Promotor: Prof. dr. H. L. Bodlaender

Acknowledgements

The four years of my PhD have been an amazing adventure. Four years to develop new ideas and to explore the mathematical beauty of algorithms, to meet many new friends, and to visit conferences in wonderful and inspiring locations.

First and foremost, I want to thank my advisor, Hans Bodlaender. I was very lucky to have Hans as an advisor: most of the supervision happened during the daily morning and afternoon coffee sessions, where small talk with colleagues was interspersed with occasional discussions about our research. Formal meetings were rare, but Hans was always available to discuss if I had something to ask. Hans provided helpful research directions, but also left me plenty of freedom to explore my own interests.

My research would not have been possible without the input of many collaborators. Key among these is Sándor Kisfaludi-Bak, with whom I initiated collaboration during the Lorentz Center workshop Fixed-Parameter Computational Geometry in 2016. Sándor was my city guide during our joint two-week research visit to Dániel Marx at the MTA SZTAKI in Budapest. This visit, together with many days in Eindhoven and Utrecht spent in a small room with a whiteboard, culminated in no less than three papers. I cannot commend Sándor enough[1]: he is a very smart, talented and hard-working mathematician and moreover, a very humble and kind person.

Another very important research visit was that to the Fukuoka Institute of Technology in Japan in 2017. While this was also a very productive visit (resulting in papers on memory-efficient algorithms, Subgraph Isomorphism, k-Path Vertex Cover and minimal separators), it also resulted in personal growth: being able to spend three weeks immersed in Japanese culture was transformative, the experience that I got can simply not be had as a tourist. I want to say *arigatō gozaimashita* to all of my Japanese friends: Tesshu Hanaka, Eiji Miyano, Yota Otachi, Toshiki Saitoh, Hisao Tamaki, Ryuhei Uehara, Yushi Uno, Tsuyoshi Yagita and to the students at the Fukuoka Institute of Technology who invited me to their homes.

To people who are not familiar with mathematics and computer science research, it might appear boring and dry. However, to me, it sometimes feels more like a thinly veiled excuse for having fun and an extension of my hobby in puzzles. In fact, my work together with Hans was initiated when I, during his Algorithms and Networks course, took a homework assignment on the puzzle game Bloxorz a bit too far. I have met many people who think alike at various International Puzzle Parties, Gathering for Gardner 10 and 13, the 2018 Fun with Algorithms Conference in Sardinia and the 2019 Bellairs Workshop on Computational Geometry in Barbados (certainly, the only thing better than having fun with mathematics is having fun with mathematics on a beautiful tropical island). I want to acknowledge all of the people that I met pursuing

[1]And Sándor tells me I shouldn't.

my interest in recreational mathematics, puzzles and their design, and in particular Andrew Cormier, Martin and Erik Demaine, Oskar van Deventer, Bob Hearn, Marc van Kreveld, Jayson Lynch, Jonathan Schroyer, Joris van Tubergen, Siert Wijnia and many others. I also want to thank the Gathering for Gardner Foundation and the late Tom Rodgers for their support.

Of course, not everything is fun and games (or at least, not the type of game that is fun to play). I want to thank Herbert Hamers for our collaboration on terrorist networks, which was initiated at Lunteren 2017. I am very happy Herbert was willing to take on this interdisciplinary challenge.

I would like to thank the reading committee for critically evaluating this manuscript and offering valuable comments: Hajo Broersma, Alexander Grigoriev, Marc van Kreveld, Jan Arne Telle and Gerhard Woeginger.

During the NETWORKS training weeks, visiting Eindhoven and at other conferences I had a lot of fun with my fellow PhD students and other colleagues: Tom Bannink, Julien Baste, Paul Bouman, Jan-Willem Buurlage, Mark de Berg, Huib Donkers, Lars Jaffke, Bart Jansen, Aleksandar Markovic, Jesper Nederlof, Tim Ophelders, Astrid Pieterse, Carolin Rehs, Jan Arne Telle, Abe Wits, Gerhard Woeginger and many, many others. Being able to interact with so many people, and especially getting to meet fellow PhDs in similar and different stages of their programme was very valuable to me. I want to say a special thank you to Astrid and Sándor, who have been good friends from the very beginning of our PhD programmes.

Of course, there are also many wonderful and inspiring people at my home base of Utrecht. My interest in algorithms was first sparked by taking part in programming contests, together with my teammates Bas te Kolste, Harry Smit and coaches Jeroen Bransen and Bas den Heijer. I want to thank Gerard Tel for teaching a wonderful introduction to algorithms class and inspiring me in the area of teaching. I have been a student assistant for many of Gerard's classes, and I appreciate the opportunities he gave me to substitute during classroom lectures and to experiment with developing practical assignments. I want to thank Erik Jan van Leeuwen for our collaboration while teaching the algorithms course, Johan van Rooij for advising me on job searching inside and out of academia, Tomas Klos for occasionally reminding me to take the train instead of flying, Han Hoogeveen for our work on data mining and flower power, Jacco Bikker for discussions on GPU programming, Linda van der Gaag for making sure Hans stayed on top of his negotiating skills, Marjan van den Akker, Rob Bisseling, Janneke Bolt, Anneke Bonsma, Wouter ten Bosch, Cassio de Campos, Gunther Cornelissen, Federico D'Ambrosio, Marcell van Geest, Roland Geraerts, Jurriaan Hage, Ivor van der Hoog, Jan van Leeuwen, Till Miltzow, Thijs van Ommen, Silja Renooij, Edith Stap, Dirk Thierens, Sjoerd Timmer, Jérôme Urhausen, Marinus Veldhorst, Peter de Waal and Jordi Vermeulen. I would also like to thank the students who I shared the joy of many moments of learning and discovery with. I would like to thank my long-term office mates, Roel van den Broek, Nils Donselaar and Marieke van der Wegen, for the good times we shared inside and out of the office and at conferences in Vienna, Lunteren, Helsinki, Nový Smokovec and Heeze.

Finally, I would like to thank my parents Hans and Loes, and several good friends and people who meant a lot for me: Elsie, Gerrit, Irene, Marrie, Cor, Daan, Jacco – thank you all very much.

Contents

1 Introduction

A combinatorial problem is a problem in which, from a finite set of possibilities, we are asked to pick a solution that is optimal (e.g., lowest cost) or has other desirable properties. Such problems occur frequently in daily life: finding the fastest route to drive from home to work, determining where to purchase what items to take optimal advantage of store promotions and creating a seating map for a celebratory dinner following a PhD defence are all examples of combinatorial problems.

An algorithm is a systematic approach to solve a given problem. Algorithms are often implemented as computer programs (e.g., as the shortest path computation in a navigation system or as a sorting algorithm to organize files by creation date) but can also be procedures carried out by hand: for example, when children are taught to multiply two numbers, what they are being taught is a multiplication *algorithm*.

When we design algorithms, we aim to make them efficient: we want to do as little work as possible to arrive at the desired result. A very naïve method to compute the product of two numbers $A \times B$ is to simply add B to itself $A - 1$ times. The product 5×6 might be computed by first computing $6 + 6 = 12$, $12 + 6 = 18$, …, $24 + 6 = 30$. This is a rather tedious process, and quickly becomes infeasible if we want to compute a larger product, such as 153×12. Long multiplication, as taught in primary school, is much more efficient. Instead of adding 153 to itself 12 times, we can instead compute $153 \times 10 + 153 \times 2 = 1530 + 306 = 1836$. To multiply two n-digit numbers, long multiplication requires n^2 elementary operations, while the naïve method can require $10^n - 2$ additions.

The complexity of an algorithm determines how much time it uses to solve a particular problem instance, and also what the maximum size of a problem instance is that can be solved within reasonable time: the repeated addition method already becomes impractical for numbers with more than, say, two or three digits, while long multiplication can be performed by hand for numbers tens of digits long.

This thesis deals primarily with problems on *graphs*: a graph is a set of points (vertices), connected by lines (edges). Graphs can be used to model for instance road networks (where edges correspond to roads and vertices to intersections), social networks (where vertices correspond to people and edges to friendships), or any other structure with relations between objects.

If a graph has a special underlying structure, this structure can sometimes be exploited to obtain faster algorithms. As a simple example, many problems that are difficult to solve on graphs in general, often have much lower complexity (or are even trivial) if the graph considered is a tree (i.e., a graph in which any two vertices are connected by a unique path). The *treewidth* of a graph measures, in some sense, how "close" a graph is to being a tree: a graph with low treewidth can be decomposed into small groups of vertices, the groups connected together in a tree, such that the connections in the graph are respected by the grouping and the tree. Problems which can be solved efficiently on trees can often also be solved efficiently on graphs of low treewidth.

As a very high-level overview of this thesis, we study the application of treewidth (and related techniques) to several types of problems, and give both algorithms and lower bounds, showing that these algorithms are (likely) optimal. We also study practical applications and implementations of treewidth-related algorithms.

In the first part, we study the application of bounded treewidth and geometric structure (e.g., planarity) to graph embedding problems (e.g., recognizing whether some small pattern occurs somewhere as a subgraph in a larger host graph). One might reasonably expect that planarity could be exploited to obtain faster algorithms for these problems. However, it turns out that this is not the case: while we do obtain slightly faster algorithms, these algorithms are not as efficient as one might expect. We also give evidence that the algorithms obtained are probably optimal.

In the second part, we study geometric intersection graphs. A geometric intersection graph can be obtained, for instance, by identifying the vertices of graphs with points in space, and connecting two points with an edge if they are within a certain distance of each other. While these graphs do not have small treewidth, we show how a slightly modified variant of treewidth can still be used to obtain faster algorithms for problems on these graphs. These algorithms also turn out to be (probably) optimal.

In the third and final part, we study practical uses of treewidth. We first study the computation of treewidth itself: we give a parallel algorithm, implemented on a GPU, for computing treewidth using partial elimination orderings. We then study how treewidth can be used to compute the Shapley Value of connectivity games. This value, a tool from game theory, helps us to estimate the importance of vertices in a graph, i.e., it is a so-called centrality measure. It can be (and has been) used to identify key players in criminal and terrorist networks. Our method, using treewidth, gives a practical approach to compute this value for graphs on which it could otherwise not be computed.

1.1. Hard Problems

As observed in the previous section, the long multiplication algorithm is much more efficient than the naive one. The time required for long multiplication scales quadratically with the number of digits, whereas the naive algorithm scales exponentially with the number of digits (n). Doubling the number of digits in long multiplication increases the running time fourfold, whereas adding just one more digit in naive multiplication increases it tenfold. Long multiplication is an example of a *polynomial-time* algorithm (running in time $O(n^c)$ for some $c > 0$; in this case, $c = 2$), whereas the naive algorithm is an example of an *exponential-time* algorithm (running in time $O(c^n)$ for some $c > 1$).

While on the one hand we strive to find efficient algorithms for problems, on the other hand, we also try to find lower bounds, showing when this can (not) be done. Doing so gives us a greater understanding of the structure of a problem, and also enables us to tell when we should stop looking for a faster algorithm (since none is likely to exist).

Perhaps the most widely celebrated technique for showing lower bounds is the theory of $\mathcal{NP}$-completeness, introduced by Cook [33] and Karp [73] and independently discovered by Levin [81]. $\mathcal{NP}$-completeness is a tool for showing that the existence of a polynomial algorithm for a given problem is unlikely. By using *reductions*, that is, showing that one problem A can be rewritten as another problem B, we can show that if problem B has a polynomial-time algorithm, then A does too. $\mathcal{NP}$-complete problems are known to be mutually reducible to one another, and, if any one $\mathcal{NP}$-complete problem has a polynomial-time algorithm, then all $\mathcal{NP}$-complete problems do. It is widely thought unlikely that $\mathcal{NP}$-complete problems have polynomial-time algorithms, so showing $\mathcal{NP}$-completeness for a problem is good evidence that it does not admit a polynomial-time algorithm.

A central problem in the theory of $\mathcal{NP}$-completeness is Satisfiability. In satisfiability, we are given boolean *variables* $x_1, \ldots, x_n$, each of which may be set to either *true* or *false* and a set of *clauses*. A clause is the disjunction of one or more *literals*, where a literal is either a variable x_i or its negation $\neg x_i$. For example, the clause $(x_1 \vee \neg x_2 \vee \neg x_4)$ can be satisfied by making either x_1 true, x_2 false, or x_4 false.

The Satisfiability problem is to determine whether all given clauses can be satisfied simultaneously by an assignment. For example, the clauses $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ can be satisfied by making x_1 true and x_2 false (so these two clauses together would form a yes-instance), but the combination $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$ does not have any satisfying assignments: the first two clauses imply at least one variable should be true and at least one variable should be false, but the assignment $x_1 = \text{false}, x_2 = \text{true}$ does not satisfy the third clause, and $x_1 = \text{true}, x_2 = \text{false}$ does not satisfy the fourth (and so this is a no-instance).

The fact that Satisfiability is $\mathcal{NP}$-complete can be used to show other problems $\mathcal{NP}$-complete[1]. As a simple example, consider the Independent Set problem: given a graph G, we want to find (at least) k vertices such that no two vertices are adjacent (i.e., are connected by an edge). Given a satisfiability formula with n variables and m clauses, we can create a graph as follows: for every clause with l literals, we create a group of l vertices, one corresponding to each literal in the clause. We connect these vertices by edges, so that from each group, we can select at most one vertex. We then connect vertices corresponding to conflicting literals by edges, i.e., if v is a vertex corresponding to literal v_i in one clause, and u is a vertex corresponding to $\neg v_i$ in another clause, we add the edge (u, v), ensuring that these vertices cannot be selected simultaneously.

[1]For the purpose of this introduction, we omit the proof that Independent Set is in $\mathcal{NP}$.

If we can find an independent set of size m in this graph, we know that it contains exactly one vertex from each group corresponding to a clause (since there are m such groups, and the edges in each group ensure we can select at most one vertex from each group), and the literals corresponding to these vertices from a satisfying assignment. Conversely, a satisfying assignment corresponds to an independent set of size m in this graph (by picking, from each clause, a literal that satisfies it and selecting the vertex corresponding to that literal to be in the independent set).

This shows that the existence of a polynomial-time algorithm for Independent Set implies a polynomial-time algorithm for Satisfiability, because we can solve a Satisfiability instance by applying the above transformation and then solving Independent Set. Conversely, if Satisfiability cannot be solved in polynomial time, then neither can Independent Set.

1.2. The Exponential Time Hypothesis

If one is willing to assume that $\mathcal{P} \neq \mathcal{NP}$ (i.e., Satisfiability does not have a polynomial-time algorithm), then $\mathcal{NP}$-completeness can be viewed as a tool to show that certain problems do not have polynomial-time algorithms. However, merely showing that a problem does not have a polynomial-time algorithm does not preclude the possibility there might be a super-polynomial, but still reasonably fast algorithm: for example, an algorithm with running time $O(n^{\log \log n})$ is not polynomial, but would still be fast enough for many practical applications. By making a stronger assumption, we can exclude such running times:

Hypothesis 1.1 (Exponential Time Hypothesis [67]). *There exists a constant $c > 1$ such that* Satisfiability *for n-variable 3-CNF formulas has no $O(c^n)$-time algorithm.*

It is possible to use the Sparsification Lemma [68] to show that the hypothesis implies that there is no $O(c'^n)$-time algorithm for *n-clause* 3-CNF formulas either (for a different constant $c' > 0$).

Now, the previous reduction showed that Satisfiability for a (3-CNF) formula with n clauses can be reduced to Independent Set on graphs with $3n$ vertices. Thus, an algorithm for Independent Set on m-vertex graphs running in time $O((c'/3)^m)$ would contradict the Exponential Time Hypothesis (ETH). Using the ETH, it is possible to derive similar lower bounds for many problems [68].

1.3. Graph Separators and Treewidth

Consider the example graph G in Figure 1.1. If we wanted to compute the maximum size of an independent set in G, we could consider all $2^9 = 512$ possible subsets of vertices in G, determine which subsets make valid independent set (i.e., contain no two vertices sharing an edge), and find the largest among these sets. In general, this gives an $O(2^n m)$-time algorithm (for n-vertex, m-edge graphs). While the base of the exponent can be improved by using more advanced branching techniques (see e.g. [116] for a $O^*(1.1996^n)$-time algorithm), the Exponential Time Hypothesis implies that we should not expect an algorithm without exponential dependence on n.

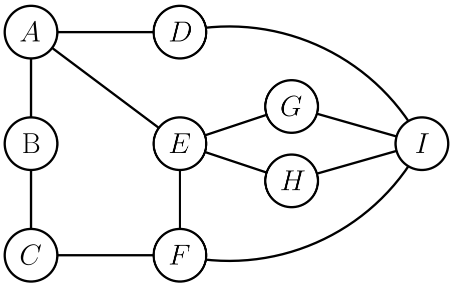


Figure 1.1. Example of a graph with 9 vertices and 12 edges.

However, if the given graph has some underlying structure, it might be possible to exploit this structure to obtain a faster algorithm. For example, consider the vertices D, E, F: they form a *separator*, separating the vertices A, B, C from G, H, I – there is no path from the left part of the graph to the right part that does not pass through D, E, F. To find a maximum independent set, we can consider the two halves of the graph separately, and combine maximum independent sets in both halves to find one for the entire graph.

To this end, we group independent sets by their *characteristic*, which is their intersection with the separator D, E, F. For example, if we consider the left half of the graph only (i.e., the vertices A, B, C, D, E, F), the largest independent set with characteristic D, F is B, D, F. If we tabulate all possible characteristics and the sizes of the largest independent sets in both halves of the graph having that characteristic, we can easily find which characteristic gives the largest possible independent set overall.

Characteristic	Left	Right	Combined
$\emptyset$	A, B	G, H	A, B, G, H
D	B, D	D, G, H	B, D, G, H
E	B, E	E, I	B, E, I
F	A, F	F, G, H	A, F, G, H
D, E	B, D, E	D, E	B, D, E
D, F	B, D, F	D, F, G, H	B, D, F, G, H
E, F	-	-	-
D, E, F	-	-	-

Table 1.1. Characteristics of independent sets with respect to the separator D, E, F. For each characteristic, the table lists the largest independent set in the left half of the graph (A, B, C, D, E, F), the right half of the graph (D, E, F, G, H, I) and in the entire graph.

Table 1.1 lists all possible characteristics and the largest independent sets having those characteristics. Note that characteristics that contain both E and F do not result in any valid independent sets since those vertices are adjacent. For each characteristic, we obtain a *partial solution* for both the left and right halves of the graph. By combining these partial solutions, we can see that the maximum independent set for the entire graph is B, D, F, G, H. The number of possibilities we had to consider was greatly reduced, since we could consider each half of the graph in isolation.

This process of considering a separator and dividing the graph can be repeated. For example, when we are considering the left half of the graph (A, B, C, D, E, F), we can again split it on the separator B, E and combine solutions for the subgraphs A, B, D, E and B, C, E, F to obtain the needed partial solutions for the left half, and we can process the right half similarly.

A formalization of this process of repeatedly dividing the graph using separators can be found in *tree decompositions*. A tree decomposition of a graph G is a tree, where each vertex of the tree is associated with a subset of the vertices of G; each such subset is also called a *bag*. An example of a tree decomposition (for the graph in Figure 1.1) is shown in Figure 1.2.

Each bag in a tree decomposition is a separator, which separates the vertices appearing in bags of the subtrees connected to that bag from each other. For instance, the root vertex in the tree decomposition shown in Figure 1.1 contains the vertices D, E, F, which separates the vertices appearing in the left subtree (A, B, D, E, F) from those appearing in the right subtree (D, E, F, G, H, I). Moving down one level in the tree, the bag B, D, E, F separates the parts A, B, D, E, B, C, E, F and D, E, F, G, H, I from each other.

In the preliminaries, we will give the formal definition of what constitutes a tree decomposition. For now, it suffices to think of the tree decomposition as a way to recursively split up a graph using separators.

The running time of an algorithm operating on a tree decomposition depends strongly on the number of vertices appearing in a bag. In Table 1.1, we had to consider $2^3 = 8$ different characteristics, since our separator consisted of 3 vertices. For a given graph, there may be many different tree decompositions. This motivates defining the *width* of a tree decomposition to be equal to the largest number of vertices appearing in any bag *minus one*[2], and the *treewidth* of a graph to be the smallest width of any tree decomposition of the graph. Thus, as the largest bag consists of 4 vertices, the decomposition in Figure 1.2 has width 3. However, there exists a decomposition with width 2 and this is (for this graph) the smallest width possible, so the treewidth of the graph in Figure 1.1 is 2.

As the name suggests, treewidth is a measure of how "tree-like" a graph is. If a graph, on a macroscopic scale, resembles a tree, then treewidth in some sense measures

[2]For historical and technical reasons.

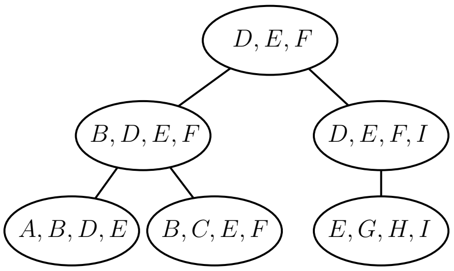


Figure 1.2. Example of a tree decomposition of the graph in Figure 1.1.

how much it deviates from being a tree locally (and in fact trees are precisely the graphs of treewidth 1).

Exploiting the fact that a graph has bounded treewidth is a celebrated and widely used technique for dealing with $\mathcal{NP}$-complete problems: thanks to the simple structure of trees, many $\mathcal{NP}$-complete problems can be solved in polynomial time on trees. Very often, such problems can also be solved in polynomial time on graphs of bounded treewidth: while the best known algorithms for such problems in general require exponential time, we can often construct an algorithm that is only exponential in the treewidth (by doing some exponential computation within each vertex of the tree, which contains a bounded number of vertices of the original graph), and then using the properties of the problem that allow it to be solved in polynomial time on trees to combine the results computed within each node of the decomposition to a solution for the original problem. For example, on an n-vertex graph of treewidth k, Independent Set can be solved in time $O(2^k k^{O(1)} n)$.

1.4. The Square Root Phenomenon in Planar Graphs

A graph is *planar* if it can be drawn in the plane without edges crossing each other. The well-known Lipton-Tarjan Separator Theorem [84, 85] states that any n-vertex planar graph has a separator, consisting of $O(\sqrt{n})$ vertices, that divides the graph into connected components, each having at most $2/3n$ vertices. As a consequence of this, it follows that planar graphs have treewidth $O(\sqrt{n})$ [14].

This fact can be combined with algorithms operating on tree decompositions, to obtain *subexponential-time* algorithms for $\mathcal{NP}$-complete problems on planar graphs. For instance, while (under the ETH) Independent Set cannot be solved in time $2^{o(n)}$ on general (n-vertex) graphs, if the graph is planar, we can exploit the fact that its treewidth is at most $O(\sqrt{n})$ and solve the problem in $2^{O(\sqrt{n})}$ time [85].

It turns out that this running time is optimal, unless the ETH fails. The standard reduction from Satisfiability to Independent Set shows that Independent Set cannot be solved in $2^{o(n)}$ time on general graphs. It is possible to take the graph created in this reduction, draw it in the plane in some arbitrary way, and then replace every crossing with a *crossover gadget* to make the graph planar. Since (in an n-vertex graph) there can be $O(n^2)$ crossings, this blows up the size of the graph from n vertices to $O(n^2)$ vertices. This, in turn, implies that a $2^{o(\sqrt{n})}$-time algorithm on planar graphs would give a $2^{o(n)}$ time algorithm on general graphs (by applying this transformation), and thus, no $2^{o(\sqrt{n})}$-time algorithm for Independent Set on planar graphs should exist, unless the ETH fails.

Similar arguments show that for many problems, $2^{\Theta(\sqrt{n})}$ is the best achievable running time on planar graphs. Examples include Dominating Set, k-Coloring (for fixed k), Hamiltonian Path and Feedback Vertex Set [86]. Square roots also appear in many other settings when dealing with planar graphs, for instance in parameterized complexity: using Bidimensionality Theory [41], it is possible to find (if it exists) an independent set of size k in an n-vertex planar graph in $2^{O(\sqrt{k})} n^{O(1)}$ time.

The behaviour that square roots often appear in the (optimal) running time for algorithms on planar graphs is so pervasive, that it has been dubbed the *Square Root Phenomenon* [88].

1.5. Graph Embedding Problems

The first part of this thesis deals with *graph embedding problems*. A graph embedding problem is any problem where we are asked to embed a graph into another graph or some other structure. As prime example of such a problem, consider Subgraph Isomorphism: given two graphs G (guest) and H (host), determine whether G is a subgraph of H. That is, can we delete vertices and edges from H to obtain (a graph that is isomorphic to) G?

If we let n denote the number of vertices of H and let k denote the number of vertices of G, then there is a trivial $O(n^k)$-time algorithm, that for each vertex of G, tries all vertices of H to map it to. Surprisingly, this simple algorithm is optimal: assuming the Exponential Time Hypothesis, there is no algorithm solving Subgraph Isomorphism in $n^{o(k)}$ time[3] [49].

Of course, a natural question then is whether bounded treewidth could help in solving Subgraph Isomorphism. In particular, an open question of Marx [87] is whether a square root phenomenon occurs for Subgraph Isomorphism on planar graphs.

A straightforward reduction from 3-Partition shows that even if G and H are both disjoint unions of paths, Subgraph Isomorphism is $\mathcal{NP}$-complete. As such graphs have treewidth 1, we cannot expect the problem to be fixed-parameter tractable parameterized by treewidth or even polynomial-time solvable for graphs of treewidth bounded by a constant. However, if we additionally impose the restriction that the graph has so-called *log-bounded fragmentation*, Subgraph Isomorphism is polynomial-time solvable for graphs of bounded (constant) treewidth [60].

Still, there is some hope: it is known that if H is planar, Subgraph Isomorphism can be solved in $2^{O(k)}n$ time [43] – faster than the lower bound for general graphs.

In the first chapter of part one, we give an algorithm for Subgraph Isomorphism on planar graphs running in time $2^{O(n/\log n)}$. This slightly improves the worst-case running time of [43] but, in contrast, is not fixed-parameter tractable with respect to k. While the algorithm does take advantage of the fact that planar graphs have bounded treewidth, the running time is dominated by the necessity of keeping track of subsets of connected components (which is exactly what [60] circumvents by assuming bounded fragmentation). We show that through *canonization*, that is, recognizing isomorphic partial solutions in the dynamic programming, we can achieve subexponential running time of $2^{O(n/\log n)}$.

In the second chapter, we show that, surprisingly, this result is optimal under the ETH. There is no algorithm for Subgraph Isomorphism on planar graphs running in time $2^{o(n/\log n)}$. Thus, a "square root phenomenon" does *not* hold for Subgraph Isomorphism on planar graphs – answering an open question of Marx [87]. This result is obtained by a reduction from Satisfiability, creating graphs with many non-isomorphic connected components.

Our results are in fact slightly stronger than stated here (the algorithm works for a more general class of graphs, and the lower bounds hold even for more restrictive classes of graphs).

[3]In [49], the authors actually show a stronger result: assuming the ETH, there is no $2^{o(n \log n)}$-time algorithm for Subgraph Isomorphism. This also rules out the existence of a $n^{o(k)}$-time algorithm.

This result is closely related with that of Fomin et al. [50] who show that, if the graph (in addition to being planar) is connected and has bounded maximum degree, a type of square root phenomenon does occur. The techniques of [50] also allow us to turn our algorithm into a parameterized one (running in time $2^{O(k/\log k)}n^{O(1)}$) if the graph is connected.

It turns out that $2^{\Theta(n/\log n)}$ is, in fact, the optimal running time for a wide range of problems involving graph embedding on planar (or H-minor free graphs): not only for simple simple variations (such as Induced Subgraph, where we are allowed to delete vertices but not edges) and other problems such as Graph Minor, but also for problems which involve embedding a graph into some other structure (such as a tree decomposition with few bags or an interval graph with few colours).

Thus, for graph embedding problems on planar graphs, it could be said that there is no square root phenomenon, but that instead there is a "$n/\log n$-phenomenon".

In the third and final chapter of part one, we give an interesting and fun application of these techniques: we study the relation of the "$n/\log n$-phenomenon" to solving polyomino packing puzzles.

1.6. Problems in Geometric Intersection Graphs

Part two of this thesis is dedicated to problems in geometric intersection graphs. A planar graph is an example of a geometric graph, arising from connecting points in the plane with non-crossing lines. A *geometric intersection* graph arises from a collection of objects in $\mathbb{R}^d$, where each object corresponds to a vertex, and two vertices are connected by an edge if and only if their corresponding objects intersect.

Without further assumptions on the objects, any graph can be represented in this way. Different graph classes can be obtained by imposing restrictions on the objects. A common example of a class of geometric intersection graphs are the *unit disk graphs*, which are the intersection graphs of collections of unit disks in the plane. Alternatively, one can think of such a graph as taking a collection of points in 2D, and connecting those points that are within distance at most 2 of each other. An example of an (unit) disk graph can be seen in Figure 1.3.

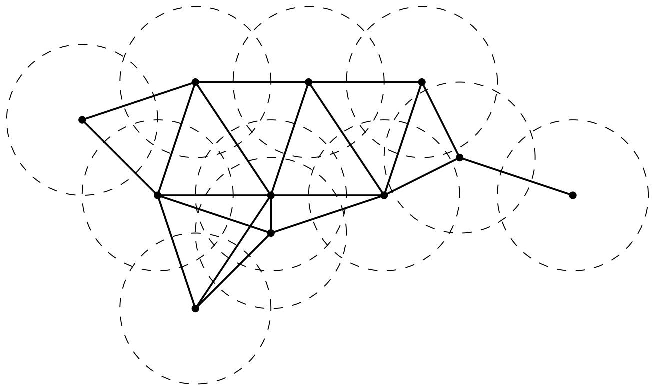


Figure 1.3. Example of a unit disk graph.

Problems in geometric intersection graphs arise naturally. Consider the Independent Set problem in geometric intersection graphs. It corresponds to finding, among a set of objects (with fixed locations in space), the largest set of objects that can be packed together without intersecting.

As we have observed with the square root phenomenon, taking advantage of the (geometric) structure of a graph can often result in significant speedups. An interesting question is whether it is possible to take advantage of similar phenomena in graphs with a different (geometric) structure, other than planarity.

Intersection graphs can be very dense and can have large cliques (consider a set of unit disks all mutually intersecting) and therefore, we would not expect treewidth to help in this case (as the maximum size of a clique is a lower bound on the treewidth). It turns out that several problems can nevertheless be solved in subexponential time on geometric intersection graphs. For instance, Fu [53] showed that Independent Set can be solved in $2^{O(\sqrt{n})}$ time in unit disk graphs (with n disks). More generally, if one considers the d-dimensional analogue (of finding a maximum independent set in an intersection graph of d-dimensional unit balls), it is possible to solve the problem in $2^{O(n^{1-1/d})}$ time.

A running time of the form $2^{O(n^{1-1/d})}$ or $n^{O(n^{1-1/d})}$ appears for many different geometric problems in various classes of d-dimensional intersection graphs [91, 102], and this running time is often (close) to optimal under the ETH [91]. Thus, d-dimensional geometric intersection graphs exhibit a type of "$n^{1-1/d}$-phenomenon".

We present an algorithmic framework for solving problems on d-dimensional geometric intersection graphs. The framework unifies various ad-hoc techniques, closes gaps in the running time bounds for several problems, and provides an easy-to-apply method for solving problems on geometric intersection graphs.

The main observation behind the framework is that while the graphs (being dense) do not admit small separators, we can instead use a kind of weighted separator based on a partition of the graph into cliques. We then use these separators to build tree decompositions, whose "width" is measured as a function of the number and size of cliques appearing in a bag. Many problems are easy to solve on cliques (for instance, Independent Set is trivial on a clique as we can select at most one vertex) and these problems are often also easy to solve for graphs of bounded clique-partitioned treewidth.

Using the framework, we obtain $2^{O(n^{1-1/d})}$-time algorithms for many problems on intersection graphs of so-called *similarly sized fat objects*, including Independent Set, Feedback Vertex Set, Dominating Set and Steiner Tree. This improves the best known running time bounds for these problems, and the running time obtained is in fact optimal (under the ETH).

To complement the algorithmic framework, there also exists a lower-bound framework – showing matching lower bounds (under the ETH) of the form $2^{\Omega(n^{1-1/d})}$ for many problems on (very restricted) classes of geometric intersection graphs. This framework is presented in [36] but is not part of this thesis.

1.7. Treewidth in Practice

While in both the first and second part of this thesis we obtain subexponential-time algorithms whose running time is optimal under the Exponential Time Hypothesis, there exists a very nice contrast between the two: for graph embedding problems – where one would expect to be able to take significant advantage of planarity – we show that the benefit of bounded treewidth and/or planarity is severely limited. On the other hand, for problems in geometric graphs, where one would *not* expect to be able to benefit (much) from bounded treewidth, we obtain an alternate variation of tree decompositions that allows us to nevertheless obtain faster algorithms.

From a theoretical perspective, having algorithms whose running time matches lower bounds obtained under the Exponential Time Hypothesis is very satisfying and provides a lot of insight into the structure of a problem. However, the algorithms obtained in the first part are not very conducive to practical use: the use of asymptotic notation in the exponent hides very large constant factors, making the algorithms unsuitable for many practical purposes (however, some of the ideas contained within, such as identifying isomorphic partial solutions in dynamic programming, may have practical applications). This is why the third part investigates using treewidth in practice.

Of course, to be able to take advantage of theory for solving practical problems, one first needs a practical method of obtaining a tree decomposition of small width. In recent years large advances have been made in this area, inspired by the PACE challenges [39, 37]. In particular, the work of Tamaki [104] has been groundbreaking. The first chapter of of Part 3 investigates exploiting the massive computational power of graphics cards (GPUs) to compute tree decompositions. Compared to traditional processors (CPUs), GPUs offer vast amounts of computation power at a relatively low cost, but present unique challenges in designing and implementing algorithms. Investigating the applications of GPU computation to parameterized and exact algorithms is a very promising direction for future research, and we present one of the first steps in this direction.

The second chapter of Part 3 deals with using bounded treewidth to solve a practical problem: in (social) network analysis, *centrality measures* play an important role in determining who the most important participants are. Recently, a very powerful class of centrality measures, *game-theoretic centrality measures* has received considerable attention in the literature. However, such measures are often hard to compute and are feasible to evaluate for only very small graphs. By exploiting treewidth, we are able to compute several such centrality measures (based on the *Shapley Value*) for graphs of bounded treewidth. We evaluate this algorithm using several graphs representing terrorist networks, and show that it indeed yields a promising method to compute game-theoretic centralities.

1.8. Published Papers

The following is a list of peer-reviewed papers that were (co-)authored by the author. This thesis is based in part on the papers [1] (Chapter 3), [3] (Chapter 4), [4] (Chapter 5), [6] (Chapter 6), [13] (Chapter 7); Chapter 8 is based on unpublished joint work with Hans L. Bodlaender and Herbert J.M. Hamers.

[1] Hans L. Bodlaender, Jesper Nederlof, and Tom C. van der Zanden. Subexponential time algorithms for embedding H-minor free graphs. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, volume 55 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:14, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[2] Hans L. Bodlaender, Marieke van der Wegen, and Tom C. van der Zanden. Stable divisorial gonality is in NP. In *International Conference on Current Trends in Theory and Practice of Informatics (SOFSEM)*, pages 111–124. Springer, 2019.

[3] Hans L. Bodlaender and Tom C. van der Zanden. Improved lower bounds for graph embedding problems. In Dimitris Fotakis, Aris Pagourtzis, and Vangelis Th. Paschos, editors, *10th International Conference on Algorithms and Complexity (CIAC 2017)*, volume 10236 of *LNCS*, pages 92–103. Springer, 2017.

[4] Hans L. Bodlaender and Tom C. van der Zanden. On the exact complexity of polyomino packing. In Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe, editors, *9th International Conference on Fun with Algorithms (FUN 2018)*, volume 100 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:10, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[5] Hans L. Bodlaender and Tom C. van der Zanden. On exploring always-connected temporal graphs of small pathwidth. *Information Processing Letters*, 142:68 – 71, 2019.

[6] Mark de Berg, Hans L. Bodlaender, Sándor Kisfaludi-Bak, Dániel Marx, and Tom C. van der Zanden. A framework for ETH-tight algorithms and lower bounds in geometric intersection graphs. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2018, pages 574–586, New York, NY, USA, 2018. ACM.

[7] Tesshu Hanaka, Hans L. Bodlaender, Tom C. van der Zanden, and Hirotaka Ono. On the maximum weight minimal separator. In T.V. Gopal, Gerhard Jäger, and Silvia Steila, editors, *Theory and Applications of Models of Computation*, pages 304–318, Cham, 2017. Springer International Publishing.

[8] Han Hoogeveen, Jakub Tomczyk, and Tom C. van der Zanden. Flower power: Finding optimal plant cutting strategies through a combination of optimization and data mining. *Computers & Industrial Engineering*, 127:39 – 44, 2019.

[9] Sándor Kisfaludi-Bak and Tom C. van der Zanden. On the exact complexity of Hamiltonian cycle and q-colouring in disk graphs. In Dimitris Fotakis, Aris Pagourtzis, and Vangelis Th. Paschos, editors, *Algorithms and Complexity*, pages 369–380, Cham, 2017. Springer International Publishing.

[10] Eiji Miyano, Toshiki Saitoh, Ryuhei Uehara, Tsuyoshi Yagita, and Tom C. van der Zanden. Complexity of the maximum k-path vertex cover problem. In M. Sohel Rahman, Wing-Kin Sung, and Ryuhei Uehara, editors, *WALCOM: Algorithms and Computation*, pages 240–251, Cham, 2018. Springer International Publishing.

[11] Tom C. van der Zanden. Parameterized complexity of graph constraint logic. In Thore Husfeldt and Iyad Kanj, editors, *10th International Symposium on Parameterized and Exact Computation (IPEC 2015)*, volume 43 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 282–293, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[12] Tom C. van der Zanden and Hans L. Bodlaender. PSPACE-completeness of Bloxorz and of games with 2-buttons. In Vangelis Th. Paschos and Peter Widmayer, editors, *Algorithms and Complexity*, pages 403–415, Cham, 2015. Springer International Publishing.

[13] Tom C. van der Zanden and Hans L. Bodlaender. Computing treewidth on the GPU. In Daniel Lokshtanov and Naomi Nishimura, editors, *12th International Symposium on Parameterized and Exact Computation (IPEC 2017)*, volume 89 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:13, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

The paper [3] received the Best Paper Award at the 10^{th} International Conference on Algorithms and Complexity.

2 Preliminaries

2.1. Basic Notations and Definitions

The following notations and definitions are used throughout this thesis.

Graphs Given a graph G, we let $V(G)$ denote its vertex set and $E(G)$ its edge set; alternatively, we may simply write V for the vertex set and E for the edge set. Given $X \subseteq V(G)$, let $G[X]$ denote the subgraph of G induced by X (i.e., a graph on vertex set X, with an edge present whenever there is an edge between the corresponding vertices in G) and use shorthand notation $E(X) = E(G[X])$. Let $Nb(v)$ denote the open neighbourhood of v, that is, the vertices adjacent to v, excluding v itself. For a set of vertices S, let $Nb(S) = \bigcup_{v \in S} Nb(v) \setminus S$. Let $CC(G)$ denote the set of the connected components of G. Given $X \subseteq V(G)$, we write as shorthand $CC(X) = CC(G[X])$.

Separators A (vertex) *separator* is a vertex set $S \subseteq V$ whose removal disconnects the graph into two or more connected components (thus, for a disconnected graph, the empty subset is a separator). We say that a subset $S \subseteq V$ is a *c-balanced separator* if no connected component of $G[V \setminus S]$ has more than cn vertices. A separator S is a *minimal separator* if no proper subset of S separates G. Note that in Chapter 6, we use a slightly different notion of (balanced) separator, where a set S that "separates" the graph into only one connected component is also considered a separator (i.e., any set S containing at least $(1 - c)n$ vertices is considered a c-balanced separator).

Functions Given a function $f : A \to B$, we let $f^{-1}(b) = \{a \in A \mid f(a) = b\}$. Depending on the context, we may also let $f^{-1}(b)$ denote (if it exists) the unique $a \in A$ so that $f(a) = b$. We say $g : A \to B$ is a *restriction* of $f : A' \to B'$ if $A \subseteq A'$ and

$B \subseteq B'$ and for all $a \in A$, $g(a) = f(a)$. We say g is an *extension* of f if f is a restriction of g.

Isomorphism We say a graph P is *isomorphic* to a graph G if there is a bijection $f : V(P) \to V(G)$ so that $(u, v) \in E(P) \iff (f(u), f(v)) \in E(G)$. We say a graph P is a *subgraph* of G if we can obtain a graph isomorphic to P by deleting edges and or vertices from G, and we say P is an *induced subgraph* if we can obtain it by deleting only vertices (and not edges).

Contractions, minors We say a graph G' is obtained from G by *contracting* edge (u, v), if G' is obtained from G by replacing vertices u, v with a new vertex w which is made adjacent to all vertices in $Nb(u) \cup Nb(v)$. A graph G' is a minor of G if a graph isomorphic to G' can be obtained from G by contractions and deleting vertices and/or edges. G' is an *induced minor* if we can obtain it by contractions and deleting vertices (but not edges). G' is a *topological minor* if we can subdivide the edges of G' to obtain a subgraph G'' of G (that is, we may repeatedly take an edge (u, v) and replace it by a new vertex w and edges (u, w) and (w, v)). Finally, G' is an *immersion minor* of G if we can obtain a graph G'' from G by a sequence of lifting operations (that is, taking a pair of edges $(u, v), (v, w)$ and replacing them by the single edge (u, w)) that contains G' as a subgraph [46].

For each of (induced) subgraph, induced (minor), topological minor, and immersion minor we define the corresponding decision problem, that is, to decide whether a pattern graph P is an (induced) subgraph/(induced minor)/topological minor/immersion minor of a host graph G. For precise definitions of these problems, we refer to the list of problems in Appendix A.

2.2. Tree Decompositions

A tree decomposition of a graph G is a rooted tree T with

- for every vertex $i \in V(T)$ a bag $X_i \subseteq V(G)$, such that $\bigcup_{i \in V(T)} X_i = V(G)$,
- for all $(u, v) \in E(G)$ an $i \in V(T)$ so that $\{u, v\} \subseteq X_i$, and
- for all $v \in V(G)$, $T[\{i \in V(T) \mid v \in X_i\}]$ is connected.

The *width* of a tree decomposition is $\max_{i \in V(T)} |X_i| - 1$ and the *treewidth* of a graph G is the minimum width over all tree decompositions of G. For a node $t \in T$, we let $G[t]$ denote the subgraph of G induced by the vertices contained in the bags of the subtree of T rooted at t.

A *path decomposition* is a tree decomposition where T is a path, and the *pathwidth* of a graph G is the minimum width of a path decomposition of G.

To simplify our algorithms, we often assume that a tree decomposition is given in *nice* form, where each node is of one of four types:

- **Leaf**: A leaf node is a leaf $i \in T$, and $|X_i| = 1$.

- **Introduce**: An introduce node is a node $i \in T$ that has exactly one child $j \in T$, and X_i differs from X_j only by the inclusion of one additional vertex.
- **Forget**: A forget node is a node $i \in T$ that has exactly one child $j \in T$, and X_i differs from X_j only by the removal of one vertex.
- **Join**: A join node is a node $i \in T$ with exactly two children $j, k \in T$, so that $X_i = X_j = X_k$.

A tree decomposition can be converted to a nice tree decomposition of the same width and of linear size in linear time [13].

Computing a minimum width tree decomposition is itself an $\mathcal{NP}$-hard problem. The state of the art includes an exact exponential-time algorithm running in time $O^*(2.9512^n)$[1] [20] and a fixed-parameter tractable algorithm running in time $2^{O(tw^3)}n$ [20]. It is thus not known whether a fixed-parameter tractable algorithm with single-exponential running time exists, but where single-exponential running times are desired a 5-approximation algorithm running in time $O(1)^k n$ due to Bodlaender et al. [18] can often be used instead (we say that a value of a minimization problem is c-approximated if the obtained value is at most c times the optimum).

2.3. Dynamic Programming on Tree Decompositions

As mentioned in the introduction, many problems can be solved efficiently of graphs of bounded treewidth, using dynamic programming. In this section, we describe the framework for dynamic programming that will later be used in Chapters 3, 6 and 8.

We assume that we are given a graph G (on which we have to solve some problem of interest) along with a nice tree decomposition $(T, \{X_t \mid t \in T\})$ of width tw. For each node t, we consider the subgraph $G[t]$, which is the subgraph of G induced by vertices appearing in the bags below t (including X_t itself, and note that since a nice tree decomposition is rooted, "below" is well-defined: these are the nodes that can be reached from t without going closer to the root).

To proceed, we need to define a notion of *partial solution*: we consider how a solution to the problem would look when restricted to a subgraph $G[t]$. For instance, in the Independent Set problem, a solution is a subset $S \subseteq V$ that is independent. Thus, it might be reasonable to define a partial solution (for Independent Set) as a subset $S \subseteq V(G[t])$ (however, the way a partial solution is defined depends on the intricacies of the problem under consideration, and as is often the case in defining subproblems for dynamic programming, it may be necessary to consider more general versions of the problem to solve the original target problem).

Once a notion of partial solution has been chosen, it is necessary to define the *characteristic* of a partial solution: as in traditional dynamic programming, one subproblem often has many possible solutions, but it is only necessary to store *one* of these (optimal) solutions due to the optimality principle: any optimal solution can be modified to include any chosen (optimal) solution to a subproblem, while remaining optimal. In dynamic programming on tree decompositions, characteristics tell us which

[1]The O^*-notation suppresses polynomial factors.

partial solutions can be grouped together as they can replace one another in any optimal solution.

For Independent Set, a suitable choice of characteristic is the intersection of the partial solution with the current bag, i.e.: given a partial solution $S \subseteq V(G[t])$, its characteristic is $S \cap X_t$. We can then create a dynamic programming *table*: for each possible characteristic, we store an (optimal) solution having that characteristic (or, more often, we store only the value of the solution, since, as is also the case in traditional dynamic programming, this often provides enough information to do the computation, and the solution itself can later be reconstructed).

Thus, in our Independent Set example, for every node t, we would create a table listing for every subset $S \subseteq X_t$ the largest size of an independent set S' of $G[t]$ such that $S' \cap X_t = S$. We let $V_t(S)$ denote this quantity (i.e., $V_t(S) = \max\{|S'| \mid S' \subseteq V(G[t]), S' \cap X_t = S, S \text{ is independent}\}$).

We process the nodes of T in post-order, i.e., the leaves first and such that whenever we process a node t, its children have already been processed (and thus their tables already computed). By assuming niceness of a tree decomposition, we can specify the entire algorithm by giving four procedures:

- **Leaf**: given a leaf node, compute its table from scratch. This procedure is usually quite simple, as for leaf node t, $G[t]$ is an isolated vertex.
- **Introduce**: given an introduce node t and the tables for its child j, compute the new table that results from introducing a given vertex.
- **Forget**: given a forget node t and the tables for its child j, compute the new table that results from forgetting a given vertex.
- **Join**: given a join node t and its two children j, k, combine partial solutions for $G[j]$ and $G[k]$ to obtain the table for $G[t]$.

Again considering our Independent Set example, the leaf case is quite simple: given a leaf node t and its bag $X_t = \{v\}$, there are two possible partial solutions for $G[t]$ (the empty set, and the singleton set $\{v\}$), thus $V_t(\emptyset) = 0$ and $V_t(\{v\}) = 1$.

In the introduce case, the graph $G[t]$ differs from $G[j]$ (for which we already know the tables) by the inclusion of one additional vertex v and its incident edges (note that these edges can only be incident to vertices in X_j); clearly, we can obtain any partial solution for $G[t]$ from a partial solution for $G[j]$ by either adding vertex v or not. Thus, the table for an introduce node t can be computed as:

$$V_t(S) = \begin{cases} V_j(S) & \text{if } v \notin S, \\ V_j(S \setminus \{v\}) + 1 & \text{if } v \in S \text{ and } S \text{ is independent}, \\ \infty & \text{otherwise.} \end{cases}$$

In the forget case, the graph $G[t]$ is the same as $G[j]$ but the characteristics might change, as there is now one less vertex included in X_t — essentially, this is a “projection” operation: several characteristics might become grouped together as one. In our Independent Set example, $V_t(S) = \max\{V_j(S), V_j(S \cup \{v\})\}$.

Finally, in the join case, the graph $G[t]$ is obtained by taking two graphs $G[j]$ and $G[k]$, which are disjoint except for the vertices in X_t, and "gluing" them together on the vertices in X_t. In our Independent Set example, $V_t(S) = V_j(S) + V_k(S) - |S|$.

Once all the tables have been computed, the solution for the original problem can be recovered from the tables in the root node r: for Independent Set, the size of a maximum independent set of G is equal to $\max_{S \subseteq X_r} V_r(S)$.

The process of dynamic programming on a nice tree decomposition is formalized in Procedure 2.1.

Algorithm 2.1. General framework for dynamic programming on a nice tree decomposition, $(T, \{X_t \mid t \in T\})$.

```
1: $r$ be the root of $T$
2: for each node $t \in T$ in post-order do
3:     if $t$ is a leaf node then
4:         Compute the table for $t$ by calling the Leaf procedure
5:     if $t$ is an introduce node then
6:         Compute the table for $t$ by calling the Introduce procedure, passing the
           previously computed table for the child node $j$
7:     if $t$ is a forget node then
8:         Compute the table for $t$ by calling the Forget procedure, passing the previously
           computed table for the child node $j$
9:     if $t$ is an join node then
10:        Compute the table for $t$ by calling the Join procedure, passing the previously
           computed tables for the child nodes $j$ and $k$
11: recover the solution $S$ from the table for the root node $r$
12: return $S$
```

I

Graph Embedding Problems

3 Algorithms

3.1. Introduction

In this chapter, we will present algorithms for *graph embedding problems*. We consider (Induced) Subgraph, (Induced) Minor and Topological Minor. We show that each of these problems can be solved in time $2^{O(n/\log n)}$ on H-minor free graphs (where n denotes the number of vertices). In the next chapter, we will give a lower bound under the Exponential Time Hypothesis, showing that this running time is (likely) optimal.

Our algorithms are based on dynamic programming, enhanced with a *canonization* technique: we reduce the number of partial solutions by identifying that some of them are isomorphic.

These problems are amongst the first for which both upper and lower bounds of the form $2^{\Theta(n/\log n)}$ are known. We conjecture that there are many more such problems, and that our techniques may be useful to establish similar bounds for them.

In each of these problems, we are asked to embed a pattern graph P into a host graph G. We consider the case in which P excludes a specific minor H, $\epsilon > 0$ is a constant, $n = |V(G)|$ and give algorithms parameterized by the treewidth tw of G and the number of vertices k of P. Our algorithms are subexponential in k or the number of vertices of G if tw is sufficiently small. Specifically, we show that for any $\epsilon > 0$ and graph H, if P is H-minor free and G has treewidth tw, (Induced) Subgraph can be solved $2^{O(k^{\epsilon} tw + k/\log k)} n^{O(1)}$ time and (Induced) Minor can be solved in $2^{O(k^{\epsilon} tw + tw \log tw + k/\log k)} n^{O(1)}$ time.

As an important special case of our result, we show that Subgraph Isomorphism can be solved in time $2^{O(k^{\epsilon}\sqrt{n} + k/\log k)}$ on H-minor free graphs (which include planar, bounded-treewidth and bounded-genus graphs). Our result can be combined with a recent result of Fomin et al. [50] to show that Subgraph Isomorphism can be solved in $2^{O(k/\log k)} n^{O(1)}$ time if P is connected and G is apex-minor free. In the next chapter, we will present a lower bound showing that this is optimal (under the ETH).

Subgraph Isomorphism has received considerable attention in the literature. Results include polynomially solvable cases, such as recognizing a fixed (i.e., not part of the input) pattern in planar graphs [43, 47], biconnected outerplanar graphs [83], graphs of log-bounded fragmentation [60], graphs of bounded genus [29] and certain subclasses of graphs of bounded treewidth [92], exact algorithms [106], lower bounds [35, 58, 103] and results on parameterized complexity [89].

For a pattern graph P of treewidth t, Subgraph Isomorphism can be solved in $2^{O(k)}n^{O(t)}$ time using the colour-coding technique [4]. If the host graph is planar, Subgraph Isomorphism can be solved in $2^{O(k)}n$ time [43]. In general graphs, Subgraph Isomorphism can be solved in $2^{O(n \log n)}$ time and, assuming the ETH, this is tight [49].

Graph minor problems are also of interest, especially in the light of Robertson and Seymour's seminal work on graph minor theory (see e.g. [105]) and the recent development of bidimensionality theory [41]. Many graph properties can be tested by checking for the inclusion of some minor. Testing whether a graph G contains a fixed minor P can be done in $O(n^3)$ time [100]; this was recently improved to $O(n^2)$ time [74]. However, the dependence on $|V(P)|$ is superexponential. Testing whether a graph P is a minor of a planar graph G can be done in $2^{O(k)}n^{O(1)}$ time [2], which is only single-exponential. Our lower bound shows that this cannot be improved to $2^{o(k/\log k)}$ (assuming the ETH). Our algorithms are subexponential in k, but (in contrast to [2, 74]) superpolynomial in n. This is to our knowledge the first subexponential minor testing algorithm for a non-trivial class of graphs.

Our algorithms are based on dynamic programming on tree decompositions. In particular, we use dynamic programming on the host graph and store correspondences to vertices in the pattern graph. The key algorithmic insight is that this correspondence may or may not use certain connected components of the pattern graph (that remain after removing some separator vertices). Instead of storing for each component whether it is used or not, we identify isomorphic connected components and store only the number of times each is used.

In [60], the authors give an algorithm for Subgraph Isomorphism, which runs in polynomial time for a host graph of bounded treewidth and a pattern graph of log-bounded fragmentation (i.e. removing a separator decomposes the graph into at most logarithmically many connected components). This is achieved using a similar dynamic programming technique, which (in general) uses time exponential in the number of connected components that remain after removing a separator. By assuming the number of connected components (fragmentation) is logarithmic, the authors obtain a polynomial time algorithm. In contrast, we consider a graph class where fragmentation is unbounded, but the number of non-isomorphic connected components is small. This leads to subexponential algorithms.

This chapter builds on techniques due to Bodlaender, Nederlof and van Rooij [25, 28]. They give a $2^{O(n/\log n)}$-time algorithm for finding tree decompositions with few bags and a matching lower bound (based on the Exponential Time Hypothesis), and a $2^{O(n/\log n)}$-time algorithm for determining whether a given k-coloured graph is a subgraph of a properly coloured interval graph. These earlier papers, coupled with our results, suggest that this technique may have many more applications, and that there exists a larger class of problems sharing this upper and lower bound.

We first give the algorithm for Subgraph Isomorphism (which can be trivially adapted to handle Induced Subgraph), then the algorithm for (Induced) Minor (which extends the algorithm for Subgraph Isomorphism by keeping track of which vertices are contracted to form a new vertex, and ensuring that these vertices eventually induce a connected subgraph), and finally the algorithm for Topological Minor (which builds upon the one for Subgraph Isomorphism by considering ways to map vertices of the host graph to edges of the pattern graph).

3.2. An Algorithm for Subgraph Isomorphism

We begin by describing an algorithm for Subgraph Isomorphism, which is based on dynamic programming on a tree decomposition T of the host graph G. This algorithm is similar to that of Hajiaghayi et al. [60] for Subgraph Isomorphism on log-bounded fragmentation graphs, and we use similar notions of (extensible) partial solutions and characteristic of a partial solution (Section 3.2). The algorithm does not achieve the claimed running time bounds. Our main contribution is the canonization technique (Section 3.3) and its analysis (Section 3.4), which can be used to reduce the number of partial solutions and gives the subexponential running time.

Definition 3.1 ((Extensible) Partial Solution). For a given node $t \in T$ of the tree decomposition of G, a *partial solution (relative to t)* is a triple (G', P', ϕ) where G' is a subgraph of $G[t]$, P' is an induced subgraph of P and $\phi : V(G') \to V(P')$ is an isomorphism from G' to P'.

A partial solution (G', P', ϕ) relative to t is *extensible* if there exists an extension of ϕ, $\psi : V(G'') \to V(P)$ which is an isomorphism from a subgraph G'' of G to P where $V(G'') \cap V(G[t]) = V(G')$.

To facilitate dynamic programming, at node t of the tree decomposition we only consider partial solutions (G', P', ϕ) which *might* be extensible (i.e. we attempt to rule out non-extensible solutions). Note that in a partial solution we have already decided on how the vertices in $G[t]$ are used, and the extension only makes decisions about vertices not in $G[t]$. Instead of dealing with partial solutions directly, our algorithm works with characteristics of partial solutions:

Definition 3.2 (Characteristic of a Partial Solution). The *characteristic* (f, S) of a partial solution (G', P', ϕ) relative to a node $t \in T$ is a function $f : X_t \to V(P) \cup \{\square\}$, together with a subset $S \subseteq V(P) \setminus f(X_t)$, so that:

- for all $v \in V(G') \cap X_t$, $f(v) = \phi(v)$ and $f(v) = \square$ otherwise,
- f is injective, except that it may map multiple elements to $\square$,
- $S = V(P') \setminus \phi(X_t)$.

The following easy observation justifies restricting our attention to characteristics of partial solutions:

Lemma 3.3 (Equivalent to Lemma 10, [60]). *If two partial solutions have the same characteristic, either both are extensible or neither is extensible.*

Lemma 3.4. *If (f, S) is the characteristic of an extensible partial solution (G', P', ϕ) relative to a node $t \in T$, then S is a union of connected components of $P[V(P) \setminus f(X_t)]$.*

Due to Hajiaghayi et al. [60]. Suppose there exist adjacent vertices $v_1, v_2 \in V(P) \setminus \phi(X_t)$ and $v_1 \in V(P')$, $v_2 \notin V(P')$. Then it is never possible to find a preimage u for v_2 in an extension of (G', P', ϕ): we require that $u \notin V(G[t])$, but all vertices adjacent to $\phi^{-1}(v_1)$ are contained in $V(G[t])$. □

This latter fact will be key to achieving the subexponential running time. The requirement that S is a union of connected components also appears in the definition of 'good pair' in Bodlaender et al. [25]. We show how to compute the characteristics of partial solutions in a bottom-up fashion, so that we can tell whether G has a subgraph isomorphic to P by examining the characteristics of the root bag. We proceed by giving pseudocode for the leaf, introduce, forget and join cases and argue for their correctness.

Algorithm 3.1. Leaf case: computes the partial solution characteristics for a leaf bag $t \in T$, with $X_t = \{v\}$.

```
1: let $R = \emptyset$
2: for each $u \in V(P) \cup \{\square\}$ do
3:     let $f : X_t \to V(P) \cup \{\square\}$ be the function so that $f(v) = u$
4:     let $R = R \cup \{(f, \emptyset)\}$
5: filter $R$
6: return $R$
```

Algorithm 3.2. Introduce case: introduces a vertex v into a bag X_t.

```
1: let $R$ be the set of partial solution characteristics for $t$
2: let $R' = \emptyset$
3: for each $(f, S) \in R$ and each $u \in V(P) \setminus (f(X_t) \cup S) \cup \{\square\}$ do
4:     if $u = \square$ or for all $w \in Nb(u) \cap f(X_t), (v, f^{-1}(w)) \in E(G)$ then
5:         let $f' : X_t \cup \{v\} \to V(P) \cup \{\square\}$ be the extension of $f$ so that $f(v) = u$
6:         let $R' = R' \cup \{(f', S)\}$
7: filter $R'$
8: return $R'$
```

Algorithm 3.3. Forget case: forgets a vertex v from a bag X_t.

```
1: let $R$ be the set of partial solution characteristics for $t$
2: let $R' = \emptyset$
3: for each $(f, S) \in R$ do
4:     let $f'$ be the restriction of $f$ to $X_t \setminus \{v\}$
5:     if $f(v) = \square$ or $f(v)$ is not adjacent to any vertex of $V(P) \setminus (f(X_t) \cup S)$ then
6:         let $R' = R' \cup \{(f', S \cup \{f(v)\} \setminus \{\square\})\}$
7: filter $R'$
8: return $R'$
```

Algorithm 3.4. Join case: combines the partial solution characteristics for two bags $X_s = X_t$.

```
1: let $R$ be the set of partial solution characteristics for $s$
2: let $T$ be the set of partial solution characteristics for $t$
3: let $R' = \emptyset$
4: for each $(f, S) \in R$ and each $(g, Q) \in T$ do
5:     if $f = g$ and $S \cap Q = \emptyset$ then
6:         let $R' = R' \cup \{(f, S \cup Q)\}$
7: filter $R'$
8: return $R'$
```

The correctness of Algorithm 3.1 is self-evident, as we simply enumerate all the possibilities for f (which means guessing a vertex in P to map v to). We will give details of the *filter* algorithm in the next section, for now it suffices to treat the pseudocode as if this call were not present.

Algorithm 3.2 extends existing partial solutions by choosing a vertex to map v to. To ensure we obtain valid characteristics of partial solutions, we check that for any edge incident to v in $P[S \cup f(X_t)]$ there is a corresponding edge in G. Because S is a union of connected components of $G[V(P) \setminus f(X_t)]$, $f(v)$ cannot be adjacent to any vertex in S, and thus it suffices to check adjacency only against vertices in $f(X_t)$. Then S remains a union of connected components since the removal of a vertex can only further disconnect S. Note that u is chosen so that f remains injective.

Algorithm 3.3 discards any solutions that would result in S not remaining the union of connected components that we require after forgetting a vertex (note that this means we keep only partial solutions were we have already chosen preimages for all of the neighbours of the image of the vertex being forgotten).

Finally, consider Algorithm 3.4. Because (as a basic property of nice tree decompositions) $V(H[i]) \cap V(H[j]) = X_i$, we obtain an injective function if and only if $S \cap R = \emptyset$. We can therefore merge two partial solutions if they map the vertices of $X_t = X_s$ in the same way and $S \cap R = \emptyset$. Note that we do indeed create all possible partial solutions in this way: given a partial solution, we can split it into partial solutions for the left and right subtrees since (as there are no edges between the internal vertices of the left and right subtrees) a connected component of S must be covered entirely by either the left or right subtree.

These algorithms, on the tree decomposition as described in Section 2.3, give an algorithm that decides Subgraph Isomorphism. Note that in contrast to the example for Independent Set given in the framework, we do not compute a value for each characteristic, rather, we compute the set of valid characteristics. The solution is reconstructed as follows: P is a subgraph of G if and only if there exists a partial solution (f, S) for the root node such that $S = V(P) \setminus f(X_t)$.

However, this algorithm does not achieve the claimed running time bound: it is essentially that of Hajiaghayi et al. [60] and the number of partial solutions that need to be considered can be $2^{\Omega(k)}$ as the subset S appearing in a characteristic can consist of the union of an arbitrary subset of components. In the next section, we will address this.

3.3. Reducing the Number of Partial Solutions Using Isomorphism Tests

In this section, we show how adapt the algorithm from the previous section to achieve the claimed running time bound. This involves a careful analysis of the number of characteristics, and using isomorphism tests to reduce this number. Currently, if the connected components of S are small (e.g., $O(1)$ vertices each) then their number is large (e.g., $\Omega(n)$ components) and thus in the worst case we have $2^{\Omega(n)}$ partial solutions. However, if there are many small connected components many will necessarily be isomorphic to each other (since there are only few isomorphism classes of small connected components) and we can thus reduce the number of characteristics by identifying isomorphic connected components:

Definition 3.5 (Partial Solution Characteristic Isomorphism). Given a bag $t \in T$, two characteristics of partial solutions $(f, S), (g, R)$ for t are *isomorphic* if:

- $f = g$,
- there is a bijection $h : CC(S) \to CC(R)$,
- for all connected components $c \in CC(S)$, c and $h(c)$ are isomorphic when all vertices $v \in c$ vertices are labelled with $Nb(v) \cap f(X_t)$ (i.e., the set of vertices of $f(X_t)$ to which v is adjacent).

Clearly, the algorithm given in the previous section remains correct even if after each step we remove duplicate isomorphic characteristics. To this end, we modify the join case (Algorithm 3.4): the disjointness check $S \cap Q = \emptyset$ should be replaced with a check that if $P[V(P) \setminus f(X_t)]$ contains $N_P(y)$ connected components of isomorphism class y, and $P[S]$ (resp. $P[Q]$) contains $N_S(y)$ (resp. $N_Q(y)$) connected components of isomorphism class y, then $N_S(y) + N_Q(y) \leq N_P(y)$. Similarly, the statement $S \cup Q$ needs to be changed to, if the union is not disjoint, replace connected components that occur more than once with other connected components of the same isomorphism class (so as to make the union disjoint while preserving the total number of components of the same type).

Call a connected component *small* if it has at most $c \log k$ vertices, and large otherwise. We let $c > 0$ be a constant that depends only on $|V(H)|$ and ϵ. We do not state our choice of c explicitly, but in our analysis we will assume it is “small enough”.

For a small connected component s, we label each of its vertices by the subset of vertices of $f(X_t)$ to which it is adjacent. We then compute a canonical representation of this labeled component, for example by considering all permutations of its vertices, and choosing the permutation that results in the lexicographically smallest representation. Note that since we only canonize the small connected components using such a trivial canonization algorithm does not affect the running time of our algorithm, as $(c \log k)!$ is only slightly superpolynomial.

Algorithm 3.5 computes a canonical representation of a partial solution. It requires that we have some predefined ordering of the vertices of G. The canonization algorithm (Algorithm 3.5) allows us to define the *filter* algorithm (Algorithm 3.6).

Traditionally, a canonization is a function that maps non-isomorphic graphs to distinct strings, and isomorphic graphs to identical strings. We use this term slightly more loosely, as our canonization algorithm may map isomorphic graphs to distinct strings since it only canonizes the small connected components. Thus, Algorithm 3.6 may not remove all duplicate isomorphic partial solutions. However, we will show that it removes sufficiently many of them.

Algorithm 3.5. Connected Component Canonization: Computes a canonical representation for a union of connected components $S \subseteq V(P) \setminus f(X_t)$

```
1: let $S'$ be the union of the large connected components of $S$
2: let $Q = \emptyset$
3: for each small connected component $s$ of $S$ do
4:     compute the canonical representation $r$ of $s$ when each $v \in V(s)$ is labelled with
       $Nb(v) \cap f(X_t)$
5:     let $Q = Q \cup \{r\}$
6: Sort $S'$ and $Q$ lexicographically
7: return $(S', Q)$
```

Algorithm 3.6. Filtering Algorithm: Filters a set of partial solution characteristics R to remove duplicates

```
1: compute a canonical representation $C_S$ for every $(f, S) \in R$ using Algorithm 3.5
2: sort $R$ first by $f$, then by $C_S$ in lexicographical order
3: loop over $R$, removing all but one of each group of isomorphic partial solutions
4: return $R$
```

3.4. Bounding the Number of Non-Isomorphic Partial Solutions

In this section, we analyse the number of non-isomorphic partial solutions, and show that the algorithm given in the previous section indeed achieves the stated time bound. In the following, let $\epsilon > 0$ and let G be a graph of treewidth at most tw. Furthermore, suppose that P is H-minor free for some fixed graph H.

Recall that a partial solution for a node $t \in T$ of the tree decomposition consists of $f : X_t \to V(P) \cup \{\square\}$ and a subset $S \subseteq V(P) \setminus f(X_t)$, which is a union of connected components of the subgraph induced by $S \subseteq V(P) \setminus f(X_t)$. The number of choices for f is at most $(k+1)^{|X_t|} = 2^{O(tw \log k)}$. We now proceed to bound the number of cases for S.

We distinguish between connected components of $V(P) \setminus f(X_t)$ of which there are "few", and connected components of $V(P) \setminus f(X_t)$ of which there can be "many", but few non-isomorphic ones. For some constant c, we say a component is *small* if it has at most $c \log k$ vertices, and *large* otherwise. The large connected components are amongst the few, since there are at most $k/(c \log k)$ components with at least $c \log k$ vertices. For each of these components, we store explicitly whether or not it is contained in S.

They contribute a factor of $2^{O(k/\log k)}$ to the number of cases. For the small connected components, we will show a partition into the "few" (which we treat similarly to the large connected components), and into the "many, but few non-isomorphic" (for which we store, for each isomorphism class, the number of components from that isomorphism class contained in S).

Lemma 3.6. *For a given node t and function $f : X_t \to V(P)$, there are at most $O(k^{\epsilon/2}tw)$ isomorphism classes of small connected components.*

Proof. For a small connected component $Q \in CC(V(P) \setminus f(X_t))$, its isomorphism class is determined by the isomorphism class of the graph induced by Q, and the adjacency of vertices $v \in Q$ to vertices in $f(X_t)$. Since $|Q| \leq c \log k$ and P is H-minor free, there exists a constant $C_H > 1$ so that there are at most $2^{C_H \cdot c \log k}$ cases for the isomorphism class of the graph induced by Q (see [7]).

What remains is to bound the number of cases for adjacency of Q to X_t. In this specific case, $Nb(Q)$ denotes the set of vertices of X_t to which Q is incident, that is, $v \in Nb(Q)$ if and only if $v \in X_t$ and there exists a vertex $u \in Q$ so that $(u, v) \in E(P)$. Using the following lemma, we further divide the small connected components into two cases: the components with a large neighbourhood, and the components with a small neighbourhood.

Lemma 3.7. *Let H be a fixed graph. Then there exists a constant d (depending on H), so that if $G = (A, B, E)$ is H-minor free and bipartite, there are at most*

- *$O(|A|)$ vertices in B with degree greater than d,*
- *$O(|A|)$ subsets $A' \subseteq A$ such that $A' = Nb(u)$ for some $u \in B$.*

Lemma 3.7 follows as a special case of Lemma 3 due to Gajarský et al. [54]. We give a simpler, self-contained proof of this special case:

Proof. Let $d = |V(H)| - 1$. We prove the first part by repeatedly contracting a high-degree vertex u (degree $> d$) in B with one of its neighbours in A. Since G is H-minor free, it certainly does not contain the complete graph $K_{|H|}$ as a minor. Therefore, the neighbourhood of u cannot be complete and the contraction can be performed in such a way that it creates at least one new edge in the subgraph induced by A. At the end of this process, the number of edges in the subgraph induced by A forms an upper bound on the number of high-degree vertices originally in B. We now use the fact that H-minor free graphs are sparse: such a subgraph contains at most $O(|A|)$ edges.

For the second part of the lemma, consider only those vertices $u \in B$ with degree at most d – since there are at most $O(|A|)$ vertices with degree $> d$ they trivially contribute at most $O(|A|)$ distinct neighbourhoods. Now, repeatedly contract a low-degree vertex u for which $Nb(u)$ does not induce a complete subgraph of A – again creating at least one new edge in the subgraph induced by A. At the end of this process, for all remaining low-degree vertices in $u \in B$, $Nb(u)$ induces a clique in A. The result now follows from the fact that H-minor free graphs can have at most linearly many cliques (see e.g. [52]). □

Taking $A = X_t$, deleting the edges between vertices in X_t and contracting every connected component $Q \in CC(V(P) \setminus f(X_t))$ to a single vertex in B, the lemma states that there are at most $O(tw)$ components with $|Nb(Q)| > d$ and that the components with $|Nb(Q)| \leq d$ have at most $O(tw)$ distinct neighbourhoods in X_t.

For the connected components $Q \in CC(V(P) \setminus f(X_t))$ with $|Nb(Q)| \leq d$, we have $2^{C_H \cdot c \log k}$ cases for the isomorphism class of Q, $O(tw)$ cases for $Nb(x)$ and for every vertex of Q, at most 2^d cases for incidence to $Nb(Q)$. We thus have at most $2^{C_h \cdot c \log k} \cdot O(tw) \cdot (2^d)^{c \log k}$ isomorphism classes for $Q \in CC(S)$ with $Nb(Q) \leq d$. For sufficiently small $c > 0$, this is $O(k^{\epsilon/2} tw)$.

This finishes the proof of Lemma 3.6. □

Since of each component there can be most k occurrences in S, the total number of cases for storing the multiplicity of each class of components is $(k+1)^{O(k^{\epsilon/2} tw)} = 2^{O(k^{\epsilon/2} tw \log k)} = 2^{O(k^{\epsilon} tw)}$.

We now have all the results we need to finish the analysis. Storing the multiplicities of the small connected components gives $2^{O(k^{\epsilon} tw)}$ cases, while storing the subset of large connected components explicitly contributes $2^{O(k/\log k)}$ cases. A partial solution is further characterized by f, for which there are only $2^{O(tw \log k)}$ cases. For a given node t of the tree decomposition, there are thus at most $2^{O(k^{\epsilon} tw + k/\log k)}$ partial solutions.

Thus, augmenting the dynamic programming algorithm described in the previous section with the filtering algorithm, we obtain the following result:

Theorem 3.8. *For any graph H and $\epsilon > 0$,* Subgraph Isomorphism *can be solved in time $2^{O(k^{\epsilon} tw + k/\log k)} n^{O(1)}$ if the host graph has treewidth tw and the pattern graph is H-minor free.*

Note that a tree decomposition does not need to be given as part of the input, as we can compute a 5-approximate tree-decomposition of G in time exponential in tw [18] and this does not increase the asymptotic running time.

3.5. Graph Minors and Induced Problem Variants

In this section, we discuss how our algorithm for Subgraph Isomorphism can be adapted to Induced Subgraph and (Induced) Minor. We begin by describing the algorithm for Graph Minor, then give a brief description on how to adapt both algorithms for the induced cases.

Note that P is a minor of G if and only if we have a function $f : V(G) \to V(P) \cup \{\square\}$, such that

- for all $v \in V(P)$, $f^{-1}(v)$ is non-empty, and induces a connected subgraph of G,
- for all $(v, w) \in E(P)$, there are $x \in f^{-1}(v)$ and $y \in f^{-1}(w)$ with $(x, y) \in E(G)$.

Vertices that are deleted are mapped to $\square$, otherwise $f(v)$ gives the vertex that v is contracted to. Call such a function a *solution* for the Graph Minor problem.

If we restrict such solutions to a subgraph $G[t]$, we obtain the notion of partial solution:

Definition 3.9 (Partial Solution (Graph Minor)). Given a node $t \in T$ of the tree decomposition of G, a *partial solution* for the Graph Minor problem relative to t is a function $f : V(G[t]) \to V(P) \cup \{\square\}$, such that

1. For each $v \in V(P)$, at least one of the following three cases holds:
   (a) each connected component of $G[t][f^{-1}(v)]$ contains at least one vertex from X_t,
   (b) $G[t][f^{-1}(v)]$ has one connected component,
   (c) $f^{-1}(v)$ is empty.
2. For all $(v, w) \in E(P)$, at least one of the following cases holds:
   (a) Some vertex of $f^{-1}(v)$ is adjacent to some vertex of $f^{-1}(w)$ in $G[t]$,
   (b) $f^{-1}(v) \cap X_t \neq \emptyset$ and $f^{-1}(w) \cap X_t \neq \emptyset$,
   (c) $f^{-1}(v) \cap X_t \neq \emptyset$ and $f^{-1}(w) = \emptyset$,
   (d) $f^{-1}(w) \cap X_t \neq \emptyset$ and $f^{-1}(v) = \emptyset$,
   (e) $f^{-1}(w) = \emptyset$ and $f^{-1}(v) = \emptyset$.

Intuitively, in 1) we require that the preimage of v can still be made connected (a), is already connected (b) or has not been assigned yet (c), and in 2) we require that the edge (v, w) is already covered (a), can still be covered (b,c,d,e).

As before, our dynamic programming algorithm uses characteristics of partial solutions:

Definition 3.10 (Characteristic of a partial solution (Graph Minor)). Given a node $t \in T$ of the tree decomposition of G and a *partial solution for the* Graph Minor *problem relative to a node t* $f : V(G[t]) \to V(P) \cup \{\square\}$, the *characteristic* of f is a tuple $(f', S, \sim, F)$ such that:

1. f' is the restriction of f to X_t,
2. $S \subseteq V(P)$ with $S = f(V(G[t])) \setminus (f(X_t) \cup \{\square\})$,
3. $\sim$ is an equivalence relation on X_t, with $v \sim w$, if and only if $f(v) = f(w)$ and there exists a path from v to w in $G[t]$ such that for all vertices x on this path $f(x) = f(v)$.
4. $F \subseteq E(P[f(X_t)])$ such that for every $(v, w) \in E(P[f(X_t)])$, it holds that $(v, w) \in F$ if and only if each of the following holds:
   (a) $f^{-1}(v) \cap X_t \neq \emptyset$,
   (b) $f^{-1}(w) \cap X_t \neq \emptyset$,
   (c) There are $x \in f^{-1}(v)$ and $y \in f^{-1}(w)$ with $(x, y) \in E(G[t])$.

As before, it is easy to see the equivalence between the existence of a minor, solution, and partial solution with certain characteristic.

Compared to our approach for Subgraph Isomorphism, we no longer require f to be injective – $f^{-1}(v)$ corresponds to the vertices that are contracted to form v. We require that $f^{-1}(v)$ eventually becomes connected. This can either be achieved inside the bag, be achieved below the bag (in the tree decomposition), or above the bag. The relation $\sim$ tracks which components are already connected by vertices below the bag.

Similarly, edges inside $P[f(X_t)]$ might not have corresponding edges inside $G[X_t]$, but might instead correspond to edges below or above this bag. The set F stores which edges correspond to edges below the current bag.

The following lemma is the counterpart of Lemma 3.3 and shows that we can apply dynamic programming:

Lemma 3.11. *If partial solutions for the* Graph Minor *problem f and g have the same characteristic and are both relative to t, then f can be extended to a solution if and only if g can be extended to a solution.*

Proof. We will prove that if f can be extended to a solution, then so can g. By symmetry, this is sufficient to prove the lemma. Let both f and g have characteristic $(f', S, \sim, F)$ and let f_* be an extension of f to a solution. Let $g_* : V(G) \to V(P) \cup \{\square\}$ be an extension of g, such that $g_*(v) = g(v)$ for all $v \in V(G[t])$ and let $g_*(v) = f(v)$ otherwise. We claim that g_* is a solution:

- First, we check that for all $v \in V(P)$, $g_*^{-1}(v)$ is non-empty and induces a connected subgraph of G:
  - If $v \notin g(G[t])$ then the property trivially holds since $g_*^{-1}(v) = f(v)$
  - If $v \in g(G[t])$ then $g_*^{-1}(v)$ certainly is non-empty. To see that it is connected, consider $A = f_*^{-1}(v) \setminus G[t]$. For each of the equivalence classes B of $\sim$ contained in $f_*^{-1}(v) \cap G[t]$, some vertex of A must be adjacent to a vertex of B, since otherwise $f_*^{-1}(v)$ would not be connected. Therefore $g_*^{-1}(v)$ is connected.

- Secondly, we check that for all $(u, v) \in E(P)$, there exist $x \in g_*^{-1}(u)$ and $y \in g_*^{-1}(v)$ with $(x, y) \in E(G)$. Since f_* is a solution, there exist $x' \in f_*^{-1}(v)$ and $y' \in f_*^{-1}(w)$ with $(x', y') \in E(G)$. We distinguish several cases; since the roles of x, y' and y, y' are symmetrical, without loss of generality we omit cases with reverse roles of x and y.

  1. There exists $x' \in f_*^{-1}(u)$ and $y' \in f_*^{-1}(v)$ such that $x' \notin G[t]$ and $y' \notin G[t]$. Then the property holds, since $g'(x') = f(x')$ and $g'(x') = f(x')$.
  2. There exists $x' \in f_*^{-1}(u)$ and $y' \in f_*^{-1}(v)$ such that $x' \in X_t$ and $y' \notin G[t]$. Then the property holds, since $g'(x') = f(x')$ and $g'(x') = f(x')$.
  3. There exists $x' \in f_*^{-1}(u)$ and $y' \in f_*^{-1}(v)$ such that $x' \in X_t$ and $y' \in X_t$. Then the property holds, since $g'(x') = f(x')$ and $g'(x') = f(x')$.
  4. It holds that $f_*^{-1}(u) \subseteq V(G[t]) - X_t$ or $f_*^{-1}(v) \subseteq V(G[t]) - X_t$. Then, since f_* extends f which has the same characteristic g, $g^{-1}(u) \subseteq V(G[t]) - X_t$ or $g^{-1}(v) \subseteq V(G[t]) - X_t$. Then the property holds by Item 2. of Definition 3.9 since g is a partial solution, which is a restriction of g_*.
  5. It holds that $f_*^{-1}(u) \cap X_t \neq \emptyset$ and $f_*^{-1}(v) \cap X_t \neq \emptyset$. If $g^{-1}(u) \cap X_t = \emptyset$ or $g^{-1}(v) \cap X_t = \emptyset$ then we are in the first or second case (since both characteristics share S). Therefore $(u, v) \in F$ by the Item 4. of Definition 3.10.

□

The following lemma shows that we can apply our technique of reducing the number of partial solution characteristics by using isomorphisms:

Lemma 3.12. *A partial solution for* Graph Minor f *with characteristic* $(f', S, \sim, F)$ *can be extended to a solution only if* S *is a union of connected components of* $G[V(P) \setminus f(X_t)]$.

Proof. Suppose there exist adjacent vertices $v_1, v_2 \in V(P) \setminus f(X_t)$ and $v_1 \in S, v_2 \notin S \cup f(X_t)$. Then in any solution g which is an extension of f, $g^{-1}(v_1) \subseteq V(G) - X_t$ and $g^{-1}(v_2) \cap G[t] = \emptyset$, so there can never be $x \in f^{-1}(v_1), y \in f^{-1}(v_2)$ so that $(x, y) \in E(G)$. □

The analysis of the number of cases of f' and S remains unchanged. There are at most $(tw)^{tw} = 2^{O(tw \log tw)}$ cases for $\sim$, and since P is sparse, at most $2^{O(tw)}$ cases for F.

We proceed by giving pseudocode for the leaf, introduce, forget and join cases for minor testing. Note that Algorithm 3.10 still needs to be modified to handle canonized characteristics, as per Section 3.3.

Algorithm 3.7. Leaf case (Minor): computes the partial solution characteristics for a leaf bag $t \in T$, with $X_t = \{v\}$.

```
1: let $R = \emptyset$
2: **for** each $u \in V(P) \cup \{\square\}$ **do**
3:     let $f : X_t \to V(P) \cup \{\square\}$ be the function so that $f(v) = u$
4:     let $\sim$ be the trivial equivalence relation on $\{v\}$
5:     let $R = R \cup \{(f, \emptyset, \sim, F)\}$
6: **filter** $R$
7: **return** $R$
```

Algorithm 3.8. Introduce case (Minor): introduces a vertex v into a bag X_t.

```
1: let $R$ be the set of partial solution characteristics for $t$
2: let $R' = \emptyset$
3: **for** each $(f, S, \sim, F) \in R$ and each $u \in V(P) \setminus S \cup \square$ **do**
4:     let $f' : X_t \cup \{v\} \to V(P) \cup \{\square\}$ be the extension of $f$ so that $f(v) = u$
5:     let $\sim'$ be the equivalence relation obtained from $\sim$ by making $v$ equivalent with
       all vertices $w \in X_t$ to which $v$ is adjacent and $f(w) = u$ and then taking the
       transitive closure
6:     let $F' = F \cup \{(f(w), f(v)) \mid w \in Nb(v)\} \cap E(f(X_t) \cup \{u\})$
7:     let $R' = R' \cup \{(f', S, \sim', F')\}$
8: **filter** $R'$
9: **return** $R'$
```

For the induced cases, only a small modification is needed: it suffices to check in the introduce case that all neighbours (in X_t) of the vertex being introduced are mapped to vertices that are adjacent to the image of the introduced vertex and discard the partial solution otherwise. We thus obtain the following theorem:

Algorithm 3.9. Forget case (Minor): forgets a vertex v from a bag X_t.

```
1: let $R$ be the set of partial solution characteristics for $t$
2: let $R' = \emptyset$
3: for each $(f, S, \sim, F) \in R$ do
4:    let $f'$ be the restriction of $f$ to $X_t \setminus \{v\}$
5:    let $S' = S, F' = F$
6:    if the size of the equivalence class of $v$ is 1 then
7:       if $|f^{-1}(f(v) \cap X_t| > 1$ or there exists $u \in V(P)$ so that $(u, f(v)) \in E(P)$ and
         $(u, f(v)) \notin F$ then
8:          skip this instance
9:       else
10:         let $S' = S' \cup \{f(v)\}$
11:         let $F' = F' \setminus \{(u, f(v)) \mid u \in Nb(f(v))\}$
12:   let $\sim'$ be obtained from $\sim$ by restriction to $X_t \setminus \{v\}$
13:   let $R' = R' \cup \{(f', S', \sim', F')\}$
14: filter $R'$
15: return $R'$
```

Algorithm 3.10. Join case (Minor): combines the partial solution characteristics for two bags $X_s = X_t$.

```
1: let $R$ be the set of partial solution characteristics for $s$
2: let $T$ be the set of partial solution characteristics for $t$
3: let $R' = \emptyset$
4: for each $(f, S, \sim, F) \in R$ and each $(g, Q, \sim', F') \in T$ do
5:    if $f = g$ and $S \cap Q = \emptyset$ then
6:       let $\sim''$ be obtained by combining $\sim$ and $\sim'$, taking their transitive closure
7:       let $R' = R' \cup \{(f, S \cup Q, \sim'', F \cup F')\}$
8: filter $R'$
9: return $R'$
```

Theorem 3.13. *For any graph H and $\epsilon > 0$, if the host graph has treewidth tw and the pattern graph is H-minor free,* Subgraph Isomorphism *and* Induced Subgraph *can be solved in time $2^{O(k^\epsilon tw + k/\log k)} n^{O(1)}$ and* Graph Minor *and* Induced Minor *can be solved in time $2^{O(k^\epsilon tw + tw \log tw + k/\log k)} n^{O(1)}$.*

As a direct corollary, we have that for any fixed graph H (not part of the input), Subgraph Isomorphism, Graph Minor, Induced Subgraph and Induced Minor can be solved in $2^{O(n^{0.5+\epsilon} + k/\log k)}$ time if the host graph is H-minor free, as H-minor free graphs have treewidth $O(\sqrt{n})$ [3]. Important special cases include planar graphs, graphs of bounded genus, and graphs of bounded treewidth.

3.6. Topological Minor

In spite of what the name may suggest, our algorithm for Topological Minor is perhaps closer to our algorithm for Subgraph Isomorphism than it is to the one for

Graph Minor. The Topological Minor problem is a relaxation of Subgraph Isomorphism, where instead of requiring that every edge of P corresponds to an edge of G, we only require that an edge of P corresponds to a (vertex-disjoint) path in G. We extend the notion of partial solution (characteristic) to consider mappings of vertices of G to edges of P. In the following, we treat edges as ordered pairs (so that each edge has a unique left and right endpoint).

Given a function f, by $\tilde{f}(u, v)$ we denote $f^{-1}(\{u, v, (u, v)\})$ (which in our context denotes the set of vertices of G that are used to form the edge between u and v in P, and the vertices that are mapped to u or v themselves).

Definition 3.14 (Partial Solution (Topological Minor)). For a given node $t \in T$ of the tree decomposition of G, a *partial solution* for the Topological Minor problem relative to t is a function $\phi : V(G[t]) \to V(P) \cup \{\square\} \cup E(P)$, together with, for each $(u, v) \in E(P)$, an assignment of orientation to each edge in the subgraph induced by $\tilde{\phi}(u, v)$, such that:

1. For all $v \in V(P)$, $|\phi^{-1}(v)| \leq 1$.
2. For all $(u, v) \in E(P)$, for each vertex w in the subgraph of G induced by $\tilde{\phi}(u, v)$, at least one of the following holds:
   (a) $w \in X_t$, the outdegree of w is at most 1 (or at most 0 if $f(w) = v$) and the indegree of w is at most 1 (or at most 0 if $f(w) = u$),
   (b) $f(w) = u$ and w has outdegree 1, indegree 0,
   (c) $f(w) = v$ and w has indegree 1, outdegree 0,
   (d) w has indegree and outdegree 1, $f(w) \neq u$ and $f(w) \neq v$.

Note that, by virtue of the degree constraints, for any $(u, v) \in E(P)$, $\tilde{\phi}(u, v)$ induces a set of disjoint paths and cycles.

It is easy to see that P is a topological minor of G if and only if there exists a partial solution relative to the root bag of G such that $V(P) \subseteq \phi(V(G))$ and for each $(u, v) \in E(P)$, $\tilde{\phi}(u, v)$ induces a subgraph for which for every vertex w, one of the conditions b-d holds. If a partial solution meets these conditions, we say it is a *solution*.

The degree conditions are such that in a solution, we end up with one path from $\phi^{-1}(u)$ and $\phi^{-1}(v)$, and possibly a set of cycles. These cycles are not relevant to the solution and can be ignored. However, allowing the possible creation of cycles allows us to attain a better running time (since we do not need to track connectivity).

Say that a partial solution ϕ relative to t is *extensible* if there exists a solution ψ such that ψ is an extension of ϕ and the orientations assigned in ϕ match those assigned in ψ.

Definition 3.15 (Characteristic of a Partial Solution (Topological Minor)). The *characteristic* (f, S) *of a partial solution* $\phi : V(G[t]) \to V(P) \cup \{\square\} \cup E(P)$ relative to a node $t \in T$ for the Topological Minor problem is a function $f : X_t \to V(P) \cup \{\square\} \cup (E(P) \times \{A, B, C, D\})$, together with a subset S, such that:

1. If $v \in X_t$ and $\phi(v) = (u, w)$ and v is such that in $\tilde{\phi}(u, v)$

(a) v has outdegree 1 and $f(v) = u$, then $f(v) = ((u, w), A)$,

(b) v has indegree 1 and $f(v) = w$, then $f(v) = ((u, w), A)$,

(c) v has indegree 1 and outdegree 1, then $f(v) = ((u, w), A)$,

(d) Otherwise, if $f(v) = u$, then $f(v) = ((u, w), B)$,

(e) Otherwise, if $f(v) = v$, then $f(v) = ((u, w), C)$,

(f) Otherwise, if v has indegree 1, then $f(v) = ((u, w), B)$,

(g) Otherwise, if v has outdegree 1, then $f(v) = ((u, w), C)$,

(h) Else (if v has outdegree and indegree 0), $f(v) = ((u, w), D)$.

2. Otherwise, if $v \in X_t$, $f(v) = \phi(v)$.
3. S is the set of vertices in $\phi(G[t])$ but not (the endpoint of an edge) in $\phi(X_t)$.

Intuitively, state A denotes the degree constraints are satisfied, B denotes a vertex still needs to get an out-edge, C denotes the vertex still needs an in-edge, and D denotes a vertex still needs both an in-edge and an out-edge.

The following lemma is once again the counterpart of Lemma 3.3 and shows that we can apply dynamic programming:

Lemma 3.16. *If partial solutions for the* Topological Minor *problem f and g have the same characteristic and are both relative to t, then f can be extended to a solution if and only if g can be extended to a solution.*

Proof. We will prove that if f can be extended to a solution, then so can g. By symmetry, this is sufficient to prove the lemma. Let both f and g have characteristic (f', S) and let f_* be an extension of f to a solution. Let $g_* : V(G) \to V(P) \cup \{\square\} \cup E(P)$ be an extension of g, such that $g_*(v) = g(v)$ for all $v \in V(G[t])$ and let $g_*(v) = f(v)$ otherwise. For each $(u, v) \in E(P)$, we let the orientation of edges in the subgraph induced by $g_*(\tilde{u, v})$ equal that of the corresponding edge in $g(\tilde{u, v})$, or, if this edge does not exist, the corresponding edge in the subgraph induced by $f_*(\tilde{u, v})$. We claim that g_* is a solution:

- For all $v \in V(P)$, it clearly holds that $|g_*^{-1}(v)| = 1$: because $g(G[t]) = f(G[t])$, if $v \in g(G[t])$ then $v \notin f(V(G) \setminus G[t])$, if $v \notin g(G[t])$ then $v \in f(V(G) \setminus G[t])$.
- For each $(u, v) \in E(P)$, and each vertex w in the subgraph induced by $\tilde{g_*}(u, v)$, the degree conditions are met: if $w \in G[t] \setminus X_t$ then they are already met in g, and since the image of the neighbourhood of w is already fixed, this remains unchanged in g_*. Otherwise, either $u \in X_t$ or $u \notin X_t$. In both cases u has the same indegree and outdegree in f and g, and since extending g to g_* in the same way we extend f to f_* changes the degrees in the same way, the degree conditions must be satisfied in g_* (since they are in f_*).

$\square$

The following lemma shows that we can apply our technique of reducing the number of partial solution characteristics by using isomorphism:

Lemma 3.17. *A partial solution for* Topological Minor f *with characteristic* (f, S) *can be extended to a solution only if* S *is a union of connected components of* $G[V(P) \setminus Z]$*, where* $Z = \{v \in V(P) \mid v \in f(X_t) \text{ or } \exists u \in Nb(v), t \in \{A, B, C, D\} : ((u, v), t) \in f(X_t) \text{ or } ((v, u), t) \in f(X_t)\}$.

Proof. Suppose there exist adjacent vertices $v_1, v_2 \in V(P) \setminus Z$ and $v_1 \in S, v_2 \notin S \cup Z$. Since v_1, v_2 are adjacent, there must be a path from $f^{-1}(v_1)$ to the vertex that (in an extension of f) is mapped to v_2. Since $v_2 \notin f(G[t])$, this path must contain a vertex of X_t. However, this contradicts the fact that $v_1, v_2 \in V(P) \setminus Z$. □

The analysis of the number of cases of f and S remains unchanged: P has $O(n)$ edges, so the fact that both edges and vertices can now appear in the range of f does not affect the asymptotic bound, and since there are only four possible vertex states, the size of the range of f is still $O(n)$. Thus, we obtain the following result:

Theorem 3.18. *For any graph* H *and* $\epsilon > 0$, Topological Minor *can be solved in time* $2^{O(k^\epsilon tw + k/\log k)} n^{O(1)}$ *if the host graph has treewidth* tw *and the pattern graph is* H*-minor free.*

3.7. Conclusions

We have presented algorithms for (Induced) Subgraph, (Induced) Minor and Topological Minor that, by taking advantage of isomorphic structures in the pattern graph, run in subexponential time on H-minor free graphs. These algorithms are essentially optimal since, in the next chapter, we will show that the existence of $2^{o(n/\log n)}$-time algorithms would contradict the Exponential Time Hypothesis. This thus settles the (traditional) complexity of these problems on (general) H-minor free graphs.

Our result applies to a wide range of graphs: we require P to be H-minor free and G to have truly sublinear treewidth. Some restriction on G is indeed necessary, since if G is an arbitrary graph then Hamiltonian Path is a special case (in which P is a path) and a $2^{o(n)}$ algorithm would contradict the ETH [86].

By combining our result with one by Fomin et al. [50], it is possible to obtain a fixed-parameter tractable algorithm with running time $2^{O(k/\log k)} n^{O(1)}$ for the case where P is connected and G is apex-minor free. However, this algorithm is randomized. Very recently [96] it has been shown that if G is planar, a deterministic $2^{O(k/\log k)} n^{O(1)}$-time algorithm is possible (where P need not necessarily be connected). An interesting open question is whether this latter result can be generalized to the case of H-minor free graphs.

We note that our lower bound proof also works for Immersion. However, our algorithmic technique does not seem to work for Immersion. Does the Immersion problem also have a $2^{O(n/\log n)}$ algorithm, or is a stronger lower bound possible?

Lemma 3.7 holds for a more general class of graphs, and we believe it may be possible to extend our result to patterns from a graph class with expansion $O(1)$ or perhaps expansion $O(\sqrt{r})$. We note that for different graph classes, a tradeoff between the size of the small connected components and the factor $k/\log k$ in the exponent is possible:

it might be possible to obtain a $2^{O(n/\log\log n)}$-time algorithm for less restrictive graph classes.

Together with the Minimum Size Tree/Path Decomposition problems ([25] – we will also study these problems in the next chapter), these problems are amongst the first for which a $2^{\Theta(n/\log n)}$ upper and lower bound is known. Our work shows that the techniques from [25] can be adapted to other problems, and we suspect there may be many more problems for which identifying isomorphic components can speed up dynamic programming algorithms.

4 Lower Bounds

4.1. Introduction

In this chapter, we give tight subexponential lower bounds for a number of graph embedding problems, showing that the algorithms presented in the previous chapter are optimal. We introduce two related combinatorial problems, which we call String Crafting and Orthogonal Vector Crafting, and show that these cannot be solved in time $2^{o(|s|/\log|s|)}$, unless the Exponential Time Hypothesis fails.

Previously, Bodlaender et al. [26] showed that several of these graph embedding problems could not be solved in $2^{o(n/\log n)}$ time, even on very restricted classes of graphs. We use our results on these string problems to obtain simplified hardness results for several graph embedding problems, on more restricted graph classes than previously known: assuming the Exponential Time Hypothesis, there do not exist algorithms that run in $2^{o(n/\log n)}$ time for

- Subgraph Isomorphism on graphs of pathwidth 1,
- Induced Subgraph Isomorphism on graphs of pathwidth 1,
- Graph Minor on graphs of pathwidth 1,
- Induced Graph Minor on graphs of pathwidth 1,
- Intervalizing 5-Coloured Graphs on trees,
- and finding a tree or path decomposition with width at most c with a minimum number of bags, for any fixed $c \geq 16$.

$2^{\Theta(n/\log n)}$ appears to be the "correct" running time for many packing and embedding problems on restricted graph classes, and we think String Crafting and

Orthogonal Vector Crafting form a useful framework for establishing lower bounds of this form.

As noted in the introduction, many $\mathcal{NP}$-complete graph problems admit faster algorithms when restricted to planar graphs. In almost all cases, these algorithms have running times that are exponential in a square root function (e.g. $2^{O(\sqrt{n})}$, $n^{O(\sqrt{k})}$ or $2^{O(\sqrt{k})}n^{O(1)}$) and most of these results are tight, assuming the Exponential Time Hypothesis. This seemingly universal behaviour has been dubbed the “Square Root Phenomenon” [88].

These results answer an open question [87] of whether the Square Root Phenomenon holds for Subgraph Isomorphism in planar graphs negatively[1]: assuming the Exponential Time Hypothesis, there is no $2^{o(n/\log n)}$-time algorithm for Subgraph Isomorphism, even when restricted to (planar) graphs of pathwidth 2 [26]. The same lower bound holds for Induced Subgraph and (Induced) Minor and is in fact tight: as shown in the previous chapter, the problems admit $2^{O(n/\log n)}$-time algorithms on H-minor free graphs.

The original proofs of this type of lower bound [26] followed by reductions from a problem called String 3-Groups. We introduce a new problem, String Crafting, and show a $2^{\Omega(|s|/\log |s|)}$-time lower bound under the ETH for this problem by giving a direct reduction from 3-Satisfiability. Using this result, we show that the $2^{\Omega(|s|/\log |s|)}$-time lower bounds for (Induced) Subgraph and (Induced) Minor hold even on graphs of pathwidth 1.

Alongside String Crafting, we introduce the related Orthogonal Vector Crafting problem. Using this problem, we show $2^{\Omega(|n|/\log |n|)}$-time lower bounds for deciding whether a 5-coloured tree is the subgraph of an interval graph (for which the same colouring is proper) and for deciding whether a graph admits a tree (or path) decomposition of width 16 with at most a given number of bags.

For any fixed k, Intervalizing k-Coloured Graphs can be solved in time $2^{O(n/\log n)}$ [28]. Bodlaender and Nederlof [25] conjecture a lower bound (under the Exponential Time Hypothesis) of $2^{\Omega(n/\log n)}$ time for $k \geq 6$; we settle this conjecture and show that it in fact holds for $k \geq 5$, even when restricted to trees. To complement this result for a fixed number of colours, we also show that there is no algorithm solving Intervalizing Coloured Graphs (with an arbitrary number of colours) in time $2^{o(n)}$, even when restricted to trees.

Minimum Size Tree Decomposition and Minimum Size Path Decomposition can also be solved in $2^{O(n/\log n)}$ time on graphs of bounded treewidth. This is known to be tight under the Exponential Time Hypothesis for $k \geq 39$ [25]. We improve this to $k \geq 16$; our proof is also simpler than that in [25].

Our results show that String Crafting and Orthogonal Vector Crafting are a useful framework for establishing lower bounds of the form $2^{\Omega(n/\log n)}$ under the Exponential Time Hypothesis. It appears that for many packing and embedding problems on restricted graph classes, this bound is tight.

[1]This was first shown in [26], however, the proofs in this chapter give a somewhat stronger result.

4.2. Additional Notation

Strings. We work with the alphabet $\{0,1\}$; i.e., strings are elements of $\{0,1\}^*$. The length of a string s is denoted by $|s|$. The i^{th} character of a string s is denoted by $s(i)$. Given a string $s \in \{0,1\}^*$, $\overline{s}$ denotes binary complement of s, that is, each occurrence of a 0 is replaced by a 1 and vice versa; i.e., $|s| = |\overline{s}|$, and for $1 \leq i \leq |s|$, $\overline{s}(i) = 1 - s(i)$. E.g., if $s = 100$, then $\overline{s} = 011$. With s^R, we denote the string s in reverse order; e.g., if $s = 100$, then $s^R = 001$. The concatenation of strings s and t is denoted by $s \cdot t$. A string s is a *palindrome*, when $s = s^R$. By 0^n (resp. 1^n) we denote the string that consists of n 0's (resp. 1's).

We say a graph is a *caterpillar tree* if it is connected and has pathwidth 1. Such graphs consist of a central path, to each of its vertices one or more leaves can be attached. Figure 4.1 shows an example of a caterpillar tree.

4.3. String Crafting and Orthogonal Vector Crafting

We now formally introduce the String Crafting problem:

String Crafting
Given: String s, and n strings $t_1, \ldots, t_n$, with $|s| = \sum_{i=1}^{n} |t_i|$.
Question: Is there a permutation $\Pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$, such that the string $t^\Pi = t_{\Pi(1)} \cdot t_{\Pi(2)} \cdots t_\Pi$ fulfils that for each i, $1 \leq i \leq |s|$, $s(i) \geq t^\Pi(i)$?

I.e., we ask whether it is possible to permute the collection of strings $\{t_1, t_2, \ldots t_n\}$, such that if we then concatenate these, we obtain a resulting string t^Π (that necessarily has the same length as s) such that on every position where t^Π has a 1, s also has a 1.

Given Π, $1 \leq i \leq |s|$, we let $idx_\Pi(i) = \max\{1 \leq j \leq n : \Sigma_{k=1}^{j} |t_k| \geq i\}$ and let $pos_\Pi(i) = i - \Sigma_{k=1}^{idx_\Pi(i)-1} |t_k|$.

We also introduce the following variation of String Crafting, where, instead of requiring that whenever t^Π has a 1, s has a 1 as well, we require that whenever t^Π has a 1, s has a 0 (i.e. the strings t^Π and s, viewed as vectors, are orthogonal).

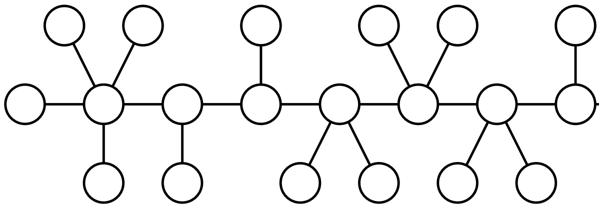


Figure 4.1. Example of a caterpillar tree.

Orthogonal Vector Crafting
Given: String s, and n strings $t_1, \ldots, t_n$, with $|s| = \sum_{i=1}^{n} |t_i|$.
Question: Is there a permutation $\Pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$, such that the string $t^\Pi = t_{\Pi(1)} \cdot t_{\Pi(2)} \cdots t_\Pi$ fulfils that for each i, $1 \leq i \leq |s|$, $s(i) \cdot t^\Pi(i) = 0$, i.e., when viewed as vectors, s is orthogonal to t^Π?

Theorem 4.1. *Suppose the Exponential Time Hypothesis holds. Then there is no algorithm that solves the* String Crafting *problem in* $2^{o(|s|/\log|s|)}$ *time, even when all strings* t_i *are palindromes and start and end with a* 1*.*

Proof. Suppose we have an instance of 3-Satisfiability with n variables and m clauses. We number the variables x_1 to x_n and for convenience, we number the clauses C_{n+1} to C_{n+m+1}.

We assume by the Sparsification Lemma that $m = O(n)$ [68].

Let $q = \lceil \log(n+m) \rceil$, and let $r = 4q+2$. We first assign a unique r-bit number to each variable and clause; more precisely, we give an injective mapping $id : \{1, \ldots, n+m\} \to \{0,1\}^r$. Let $nb(i)$ be the q-bit binary representation of i, such that $0 \leq nb(i) \leq 2^r - 1$. We set, for $1 \leq i \leq n+m$:

$$id(i) = 1 \cdot nb(i) \cdot \overline{nb(i)} \cdot \overline{nb(i)}^R \cdot nb(i)^R \cdot 1$$

Note that each $id(i)$ is an r-bit string that is a palindrome, ending and starting with a 1.

We first build the string s. We let s be the concatenation of $2n$ strings, each representing one of the literals.

Suppose the literal x_i appears c_i times in a clause, and the literal $\neg x_i$ appears d_i times in a clause. Set $f_i = c_i + d_i$. Assign the following strings to the pair of literals x_i and $\neg x_i$:

- a^{x_i} is the concatenation of the id's of all clauses in which x_i appears, followed by d_i copies of the string $1 \cdot 0^{r-2} \cdot 1$.
- $a^{\neg x_i}$ is the concatenation of the id's of all clauses in which $\neg x_i$ appears, followed by c_i copies of the string $1 \cdot 0^{r-2} \cdot 1$.
- $b^i = id(i) \cdot a^{x_i} \cdot id(i) \cdot a^{\neg x_i} \cdot id(i)$.

Now, we set $s = b^1 \cdot b^2 \cdots b^{n-1} \cdot b^n$.

We now build the collection of strings t_i. We have three different types of strings:

- *Variable selection:* For each variable x_i we have one string of length $(f_i + 2)r$ of the form $id(i) \cdot 0^{r \cdot f_i} \cdot id(i)$.
- *Clause verification:* For each clause C_i, we have a string of the form $id(i)$.
- *Filler strings:* A filler string is of the form $1 \cdot 0^{r-2} \cdot 1$. We have $n + 2m$ filler strings.

Thus, the collection of strings t_i consists of n variable selection strings, m clause verification strings, and $n + 2m$ filler strings. Notice that each of these strings is a palindrome and ends and starts with a 1.

The idea behind the reduction is that s consists of a list of variable identifiers followed by which clauses a true/false assignment to that variable would satisfy. The variable selection gadget can be placed in s in two ways: either covering all the clauses satisfied by assigning true to the variable, or covering all the clauses satisfied by assigning false. The clause verification strings then fit into s only if we have not covered all of the places where the clause can fit with variable selection strings (corresponding to that we have made some assignment that satisfies the clause).

Furthermore, note that since $\Sigma_{i=1}^{n} f_i = 3m$, the length of s is $(3n + 6m)r$, the combined length of the variable selection strings is $(2n + 3m)r$, the combined length of the clause verification strings is mr, and the filler strings have combined length $(n + 2m)r$.[2]

In the following, we say a string t_i is mapped to a substring s' of s if s' is the substring of s corresponding to the position (and length) of t_i in t^Π.

Lemma 4.2. *The instance of* 3-Satisfiability *is satisfiable, if and only if the constructed instance of* String Crafting *has a solution.*

Proof. First, we show the forward implication. Suppose we have a satisfying assignment to the 3-Satisfiability instance. Consider the substring of s formed by b^i, which is of the form $id(i) \cdot a^{x_i} \cdot id(i) \cdot a^{\neg x_i} \cdot id(i)$. If in the satisfying assignment x_i is true, we choose the permutation Π so that variable selection string $id(i) \cdot 0^{r \cdot f_i} \cdot id(i)$ corresponding to x_i is mapped to the substring $id(i) \cdot a^{\neg x_i} \cdot id(i)$; if x_i is false, we map the variable selection string onto the substring $id(i) \cdot a^{x_i} \cdot id(i)$. A filler string is mapped to the other instance of $id(i)$ in the substring.

Now, we show how the clause verification strings can be mapped. Suppose clause C_j is satisfied by the literal x_i (resp. $\neg x_i$). Since x_i is true (resp. false), the substring a^{x_i} (resp. $a^{\neg x_i}$) of s is not yet used by a variable selection gadget and contains $id(j)$ as a substring, to which we can map the clause verification string corresponding to C_j.

Note that in s now remain a number of strings of the form $1 \cdot 0^{r-2} \cdot 1$ and a number of strings corresponding to id's of clauses, together $2m$ such strings, which is exactly the number of filler strings we have left. These can thus be mapped to these strings, and we obtain a solution to the String Crafting instance. It is easy to see that with this construction, s has a 1 whenever the string constructed from the permutation does.

Next, for the reverse implication, consider a solution Π to the String Crafting instance. We require the following lemma:

Lemma 4.3. *Suppose that $t_i = id(j)$. Then the substring w of s corresponding to the position of t_i in t^Π is $id(j)$.*

[2]Note that a single given variable may appear as literal in many clauses, but since the total number of clauses is linear (by the Sparsification Lemma), we still obtain the claimed lower bound bound. In the next chapter, we will see a variation on this proof that reduces from Satisfiability with few occurrences per variable, further simplifying the proof (in this proof, the variable selection gadgets all have the same length).

Proof. Because the length of each string is a multiple of r, w is either $id(k)$ for some k, or the string $1 \cdot 0^{r-2} \cdot 1$. Clearly, w cannot be $1 \cdot 0^{r-2} \cdot 1$ because the construction of $id(i)$ ensures that it has more than 2 non-zero characters, so at some position w would have a 1 where w' does not. Recall that $id(i) = 1 \cdot nb(i) \cdot \overline{nb(i)} \cdot \overline{nb(i)}^R \cdot nb(i)^R \cdot 1$. If $j \neq k$, then either at some position $nb(k)$ has a 0 where $nb(j)$ has a 1 (contradicting that Π is a solution) or at some position $\overline{nb(k)}$ has a 0 where $\overline{nb(j)}$ has a 1 (again contradicting that Π is a solution). Therefore $j = k$. □

Clearly, for any i, there are only two possible places in t^Π where the variable selection string $id(i) \cdot 0^{r \cdot f_i} \cdot id(i)$ can be mapped to: either in the place of $id(i) \cdot a^{x_i} \cdot id(i)$ in s or in the place of $id(i) \cdot a^{\neg x_i} \cdot id(i)$, since these are the only (integer multiple of r) positions where $id(i)$ occurs in s. If the former place is used we set x_i to false, otherwise we set x_i to true.

Now, consider a clause C_j, and the place where the corresponding clause verification gadget $id(j)$is mapped to. Suppose it is mapped to some substring of $id(i) \cdot a^{x_i} \cdot id(i) \cdot a^{\neg x_i} \cdot id(i)$. If $id(j)$ is mapped to a substring of a^{x_i} then (by construction of a^{x_i}) x_i appears as a positive literal in C_j and our chosen assignment satisfies C_j (since we have set x_i to true). Otherwise, if $id(j)$ is mapped to a substring of $a^{\neg x_i}$ x_i appears negated in C_j and our chosen assignment satisfies C_j (since we have set x_i to false).

We thus obtain a satisfying assignment for the 3-Satisfiability instance. □

Since in the constructed instance, $|s| = (3n + 6m)r$ and $r = O(\log n), m = O(n)$, we have that $|s| = O(n \log n)$. A $2^{o(|s|/\log|s|)}$-time algorithm for String Crafting would give a $2^{o(n \log n / \log(n \log n))} = 2^{o(n)}$-time algorithm for deciding 3-Satisfiability, violating the ETH. □

Note that we can also restrict all strings t_i to start and end with a 0 by a slight modification of the proof.

Theorem 4.4. *Assuming the Exponential Time Hypothesis,* Orthogonal Vector Crafting *cannot be solved in* $2^{o(|s|/\log|s|)}$ *time, even when all strings* t_i *are palindromes and start and end with a* 1.

Proof. This follows from the result for String Crafting, by taking the complement of the string s. □

Again, we can also restrict all strings t_i to start and end with a 0.

As illustrated by the following theorem, these lower bounds are tight. The algorithm is a simpler example of the techniques used in [25, 28] and those in Chapter 3. There, the authors use isomorphism tests on graphs; here, we use equality of strings.

Theorem 4.5. *There exists algorithms, solving* String Crafting *and* Orthogonal Vector Crafting *in* $2^{O(|s|/\log|s|)}$ *time.*

Proof. The brute-force algorithm of trying all $n!$ permutations of the strings $t_1, \ldots, t_n$ would take $O(|s|!s)$ time in the worst case. This can be improved to $O(2^{|s|}s^2)$ by simple Held-Karp [66] dynamic programming: for each (multi-)subset $K \subseteq \{t_1, \ldots, t_n\}$ and $l = \Sigma_{t \in K}|t|$ we memoize whether the substring $s(1) \cdots s(l)$ of s together with K forms a positive instance of String Crafting (resp. Orthogonal Vector Crafting).

The number of such (multi-)subsets K is $2^{|s|}$ in the worst case. However, in this case, each string $t \in K$ is of length 1 and we can instead store the multiplicity of each string, making for only $O(|s|^2)$ cases (since each string is either 0 or 1).

More generally, call a string t_i *long* if $|t_i| \geq \log_2(|s|)/2$ and *short* otherwise. There are at most $2|s|/\log|s|$ long strings, and as such we can store explicitly what subset of the long strings is in K (giving $2^{O(|s|/\log|s|)}$ cases). Since there are at most $2^{\log|s|/2} = \sqrt{|s|}$ distinct short strings, storing the multiplicity of each one contributes at most $|s|^{\sqrt{|s|}} = 2^{\sqrt{|s|}\log|s|}$ cases. □

4.4. Lower Bounds for Graph Embedding Problems

Theorem 4.6. *Suppose the Exponential Time Hypothesis holds. Then there is no algorithm solving* Subgraph Isomorphism *in* $2^{o(n/\log n)}$ *time, even if* G *is a caterpillar tree of maximum degree* 3 *or* G *is connected, planar, has pathwidth* 2 *and has only one vertex of degree greater than* 3 *and* P *is a tree.*

Proof. By reduction from String Crafting. We first give the proof for the case that G is a caterpillar tree of maximum degree 3, We construct G from s as follows: we take a path of vertices $v_1, \ldots, v_{|s|}$ (*path vertices*). If $s(i) = 1$, we add a *hair vertex* h_i and edge (v_i, h_i) to G (obtaining a caterpillar tree). We construct P from the strings t_i by, for each string t_i repeating this construction, and taking the disjoint union of the caterpillars created in this way (resulting in a graph that is a forest of caterpillar trees, i.e., a graph of pathwidth 1). An example of this construction is depicted in Figure 4.2.

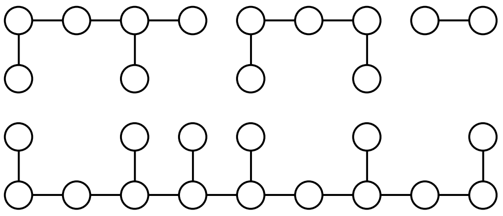


Figure 4.2. Simplified example of the graphs created in the hardness reduction for Theorem 4.6. The bottom caterpillar represents the host graph (corresponding to string s), the top caterpillars represent the strings t_i and form the guest graph. Depicted is an instance where $s = 101110101$ and $t_1 = 1010, t_2 = 101$ and $t_3 = 00$. (Note that these strings do not satisfy the requirements of the theorem, and are purely for illustration of the construction).

Lemma 4.7. *The constructed instance of* G *contains* P *as a subgraph only if the instance of* String Crafting *has a solution.*

Proof. Suppose P contains G as a subgraph. Since $\Sigma_i |t_i| = |s|$ and each string t_i starts and ends with a 1, the path vertices of P and G must be in one-to-one correspondence (we cannot map a hair vertex of P to a path vertex of G since otherwise we would not be able to fit all connected components of P into G). The order in which the connected components of P appear as we traverse the path of G gives a permutation Π of the strings t_i. We claim that this permutation is a solution to the String Crafting instance, since G must have a hair vertex whenever P has one (or else we would not have found P as a subgraph) we have that s has a 1 whenever t^Π has a 1. Note that it does not matter that we can flip a component of P (embed the vertices of the path in the reverse order) since the strings t_i are palindromes. □

Lemma 4.8. *The constructed instance of G contains P as a subgraph if the instance of* String Crafting *has a solution.*

Proof. Let Π be a solution for the String Crafting instance. We can map the path vertices of the connected components of P to the path vertices of G in the order the corresponding strings appear in the permutation Π (e.g. the first path vertex of the connected component corresponding to $\Pi(1)$ gets mapped to the first path vertex v_1, the first path vertex of the component corresponding to $\Pi(2)$ gets mapped to the $|t_{\Pi(1)}|+1^{\text{th}}$ path vertex,...). Whenever a path vertex in P is connected to a hair vertex, t^Π has a 1 in the corresponding position, and therefore s has a 1 in the corresponding position as well and thus G also has a hair vertex in the corresponding position. We can thus appropriately map the hair vertices of P to the hair vertices of G, and see that P is indeed a subgraph of G. □

Since the constructed instance has $O(|s|)$ vertices, this establishes the first part of the lemma. For the case that G is connected, we add to the graph G constructed in the first part of the proof a vertex u and, for each path vertex v_i, an edge (v_i, u). To P we add a vertex u' that has an edge to some path vertex of each component. By virtue of their high degrees, u must be mapped to u' and the remainder of the reduction proceeds in the same way as in the first part of the proof. □

We now show how to adapt this hardness proof to the case of Induced Subgraph:

Theorem 4.9. *Suppose the Exponential Time Hypothesis holds. Then there is no algorithm solving* Induced Subgraph *in $2^{o(n/\log n)}$ time, even if G is a caterpillar tree of maximum degree 3 or G is connected, planar, has pathwidth 2 and has only one vertex of degree greater than 3 and P is a tree.*

Proof. Matoušek and Thomas [92] observe that by subdividing each edge once, a subgraph problem becomes an induced subgraph problem. Due to the nature of our construction, we do not need to subdivide the hair edges. We can adapt the proof of Theorem 4.6 by subdividing every path edge (but not the hair edges) and for the connected case, also subdividing the edges that connect to the central vertices u and u'. □

We now show how to adapt this proof to (Induced) Minor, Shallow Minor and Topological Minor:

Theorem 4.10. *Suppose the Exponential Time Hypothesis holds. Then there is no algorithm solving* (Induced) Minor, Shallow Minor *or* Topological Minor *in $2^{o(n/\log n)}$ time, even if G is a caterpillar tree of maximum degree 3 or G is connected, planar, has pathwidth 2 and has only one vertex of degree greater than 3 and P is a tree.*

Proof. We can use the same reduction as for (induced) subgraph. Clearly, if P is an (induced) subgraph of G then it is also an (induced/shallow/topological) minor. Conversely, if P is not a subgraph of G then allowing contractions in G or subdivisions in P do not help: contracting a hair edge simply removes that edge and vertex from the graph, while contracting a path edge immediately makes the path too short to fit all the components. □

4.5. Tree and Path Decompositions with Few Bags

In this section, we study the minimum size tree and path decomposition problems:

> MINIMUM SIZE TREE DECOMPOSITION OF WIDTH k (k-MSTD)
> **Given:** A graph G, integers k, n.
> **Question:** Does G have a tree decomposition of width at most k, that has at most n bags?

The Minimum Size Path Decomposition (k-MSPD) problem is defined analogously. The following theorem is an improvement over Theorem 3 of [25], where the same was shown for $k \geq 39$; our proof is also simpler.

Theorem 4.11. *Let $k \geq 16$. Suppose the Exponential Time Hypothesis holds, then there is no algorithm for k-MSPD or k-MSTD using $2^{o(n/\log n)}$ time.*

Proof. By reduction from ORTHOGONAL VECTOR CRAFTING. We begin by showing the case for MSPD, but note the same reduction is used for MSTD.

For the string s, we create a connected component in the graph G as follows: for $1 \leq i \leq |s|+1$ we create a clique C_i of size 6, and (for $1 \leq i \leq |s|$) make all vertices of C_i adjacent to all vertices of C_{i+1}. For $1 \leq i \leq |s|$, if $s(i) = 1$, we create a vertex s_i and make it adjacent to the vertices of C_i and C_{i+1}.

For each string t_i, we create a component in the same way as for s, but rather than using cliques of size 6, we use cliques of size 2: for each $1 \leq i \leq n$ and $1 \leq j \leq |t_i|+1$ create a clique $T_{i,j}$ of size 2 and (for $1 \leq j \leq |t_i|$) make all vertices of $T_{i,j}$ adjacent to all vertices of $T_{i,j+1}$. For $1 \leq j \leq |t_i|$, if $t_i(j) = 1$, create a vertex $t_{i,j}$ and make it adjacent to the vertices of $T_{i,j}$ and $T_{i,j+1}$.

An example of the construction (for $s = 10110$ and $t_1 = 01001$) is shown in Figure 4.3. We now ask whether a path decomposition of width 16 exists with at most $|s|$ bags.

Lemma 4.12. *If there exists a solution Π to the* ORTHOGONAL VECTOR CRAFTING *instance, then G has a path decomposition of width 16 with at most $|s|$ bags.*

Proof. Given a solution Π, we show how to construct such a decomposition with bags $X_i, 1 \leq i \leq |s|$. In bag X_i we take the vertices C_i, C_{i+1} and (if it exists) the vertex s_i. We also take the cliques $T_{idx_\Pi(i),pos_\Pi(i)}$, $T_{idx_\Pi(i+1),pos_\Pi(i+1)}$ and the vertex $t_{idx_\Pi(i),pos_\Pi(i)}$ (if it exists).

Each bag contains two cliques of size 6 and two cliques of size 2, adding up to 16 vertices in each bag. Each bag may additionally contain a vertex s_i or a vertex $t_{idx_\Pi(i),pos_\Pi(i)}$, but, by nature of a solution to ORTHOGONAL VECTOR CRAFTING, not

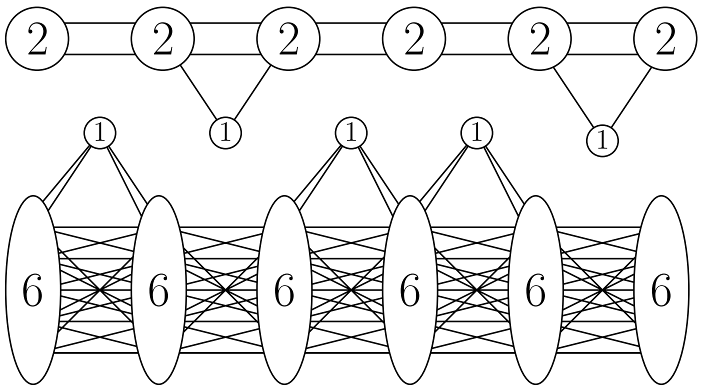


Figure 4.3. Simplified example of the graph created in the hardness reduction for Theorem 4.11. The circles and ellipses represent cliques of various sizes. The component depicted in the top of the picture corresponds to $t_1 = 01001$, while the component at the bottom corresponds to $s = 10110$.

both, showing that each bag contains at most 17 vertices and as such the decomposition indeed has width 16. □

Lemma 4.13. *If G has a tree decomposition of width 16 with at most $|s|$ bags, then the instance of* ORTHOGONAL VECTOR CRAFTING *has a solution.*

Proof. Suppose we have a tree decomposition of width 16 with at most $|s|$ bags. Since for any $1 \leq i \leq |s|$, $C_i \cup C_{i+1}$ induces a clique, there exists a bag that contains both C_i and C_{i+1}. Moreover, since each clique C_i contains six vertices, each bag of the decomposition can contain at most two such cliques. Moreover, since the decomposition has at most (and thus we can assume exactly) $|s|$ bags, there exists exactly one bag containing both C_i and C_{i+1}.

Since (for $1 < i < |s|$) the bag containing C_i and C_{i+1} must be adjacent to the bag containing C_{i-1} and C_i and to the bag containing C_{i+1} and C_{i+2} we see that all but two bags have degree at least two and the remaining two bags have degree at least 1. Since a tree/path decomposition cannot have cycles, we see that the tree/path decomposition must take the form of a path, where the bag containing C_i and C_{i+1} is adjacent to the bag containing C_{i-1} and C_i and to the bag containing C_{i+1} and C_{i+2}.

Assume the bags of the decomposition are $X_1, \dots X_s$, and $C_i, C_{i+1} \subseteq X_i$.

Since in each of the bags we now have capacity for five more vertices, we see that each bag contains exactly two cliques $T_{i,j}, 1 \leq i \leq n, 1 \leq j \leq |t_i| + 1$ and that there exists a bag that contains both $T_{i,j}$ and $T_{i,j+1}$ for all $1 \leq i \leq n, 1 \leq j \leq |t_i|$. Moreover, for each i, the bags containing $\{T_{i,j} \cup T_{i,j+1} : 1 \leq j \leq |t_i|\}$ must be consecutive and appear in that order (or in the reverse order, but by the palindromicity of t_i this case is symmetric).

Note that at this point, each bag contains exactly 16 vertices and has room for exactly 1 more vertex (which can be either an s_i or a t_{ij}).

Thus, if we were to list the intersection of each bag X_i with $\{T_{i,j}, 1 \leq i \leq n, 1 \leq j \leq |t_i|\}$, we would obtain the following sequence:

$$\{T_{\Pi(1),1}, T_{\Pi(1),2}\}, \dots, \{T_{\Pi(1),|t_{\Pi(1)}|}, T_{\Pi(1),|t_{\Pi(1)}|+1}\}$$
$$\{T_{\Pi(2),1}, T_{\Pi(2),2}\}, \dots, \{T_{\Pi(2),|t_{\Pi(2)}|}, T_{\Pi(2),|t_{\Pi(2)}|+1}\}$$
$$\dots$$
$$\{T_{\Pi(n),1}, T_{\Pi(n),2}\}, \dots, \{T_{\Pi(n),|t_{\Pi(n)}|}, T_{\Pi(n),|t_{\Pi(n)}|+1}\}$$

Which gives us the permutation Π and string t^{Π} we are after. Note that $s(i)$ and $t^{\Pi}(i)$ cannot both be 1, because otherwise the vertex s_i (being adjacent to the vertices of C_i and C_{i+1}) must go in bag X_i, but so must the vertex $t_{idx_{\Pi}(i),pos_{\Pi}(i)}$ which would exceed the capacity of the bag. □

The size of the graph created in the reduction is $O(|s|)$, so we obtain a $2^{\Omega(n/\log n)}$ lower bound for 16-MSPD under the Exponential Time Hypothesis. We can extend the reduction to $k > 16$ by adding universal vertices to the graph.

For the tree decomposition case, note that a path decomposition is also a tree decomposition. For the reverse implication, we claim that a tree decomposition of G of width 16 with at most $|s|$ bags must necessarily be a path decomposition. This is because for each $1 \leq i \leq |s|$, there exists a unique bag containing C_i and C_{i+1} which is adjacent to the bag containing $C_i - 1$ and C_i (if $i > 1$) and to the bag containing C_{i+1} and C_{i+2} (if $i < |s|$). All but two bags are thus adjacent to at least two other ones, and a simple counting argument shows that there therefore is no bag with degree greater than two (or we would not have enough edges). □

4.6. Intervalizing Coloured Graphs

In this section, we consider the problem of intervalizing coloured graphs:

Intervalizing Coloured Graphs
Given: A graph $G = (V, E)$ together with a proper colouring $c : V \to \{1, 2, \dots, k\}$.
Question: Is there an interval graph G' on the vertex set V, for which c is a proper colouring, and which is a supergraph of G?

Intervalizing Coloured Graphs is known to be $\mathcal{NP}$-complete, even for 4-coloured caterpillars (with hairs of unbounded length) [5]. In contrast with this result we require five colours instead of four, and the result only holds for trees instead of caterpillars. However, we obtain a $2^{\Omega(n/\log n)}$ lower bound under the Exponential Time Hypothesis, whereas the reduction in [5] is from Multiprocessor Scheduling and to our knowledge, the best lower bound obtained from it is $2^{\Omega(\sqrt[5]{n})}$ (the reduction is weakly polynomial in the length of the jobs, which following from the reduction from 3-Partition in [70] is $\Theta(n^4)$). In contrast to these hardness results, for the case with 3 colours there is an $O(n^2)$ time algorithm [16, 17].

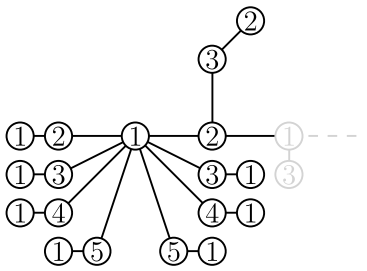
Theorem 4.14. Intervalizing Coloured Graphs *does not admit a* $2^{o(n/\log n)}$*-time algorithm, even for* 5*-coloured trees, unless the Exponential Time Hypothesis fails.*

Proof. Let $s, t_1, \ldots, t_n$ be an instance of Orthogonal Vector Crafting. We construct $G = (V, E)$ in the following way:

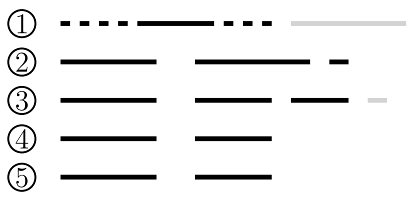
S-String Path. We create a path of length $2|s| - 1$ with vertices $p_0, \ldots p_{2|s|-2}$, and set $c(p_i) = 1$ if i is even and $c(p_i) = 2$ if i is odd. Furthermore, for even $0 \leq i \leq 2|s| - 2$, we create a neighbour n_i with $c(n_i) = 3$.

Barriers. To each endpoint of the path, we attach the *barrier gadget*, depicted in Figure 4.4. The gray vertices are not part of the barrier gadget itself, and represent p_0 and n_0 (resp. $p_{2|s|-2}$ and $n_{2|s|-2}$). Note that the barrier gadget operates on similar principles as the barrier gadget due to Alvarez et al. [5]. We shall refer to the barrier attached to p_0 as the left barrier, and to the barrier attached to $p_{2|s|-2}$ as the right barrier.

The barrier consists of a central vertex with colour 1, to which we connect eight neighbours (*clique vertices*), two of each of the four remaining colours. Each of the clique vertices is given a neighbour with colour 1. To one of the clique vertices with colour 2 we connect a vertex with colour 3, to which a vertex with colour 2 is connected (*blocking vertices*). This clique vertex shall be the barrier's *endpoint*. Note that the neighbour with colour 1 of this vertex is not considered part of the barrier gadget, as it is instead a path vertex. We let C_l (e_l) denote the center (endpoint) of the left barrier, and C_r (e_r) the center (endpoint) of the right barrier.



(a) Barrier Gadget



(b) Interval Representation

Figure 4.4. (a) Barrier Gadget. The gray vertices are not part of the barrier gadget itself, and show how it connects to the rest of the graph. (b) How the barrier gadget may (must) be intervalized.

T-String Paths. Now, for each string t_i, we create a path of length $2|t_i| + 1$ with vertices $q_{i,0}, \ldots, q_{i,2|t_i|}$ and set $c(q_{i,j}) = 3$ if j is odd and set $c(q_{i,j}) = 2$ if j is even. We make $q_{i,1}$ adjacent to U. Furthermore, for odd $1 \leq j \leq 2|t_i| - 1$, we create a neighbour m_j with $c(m_j) = 1$. We also create two *endpoint vertices* of colour 3, one of which is adjacent to $q_{i,0}$ and the other to $q_{i,2|t_i|}$,

Connector Vertex. Next, we create a *connector vertex* of colour 5, which is made adjacent to p_1 and to $q_{i,1}$ for all $1 \leq i \leq n$. This vertex serves to make the entire graph connected.

Marking Vertices. Finally, for each $1 \leq i \leq |s|$ (resp. for each $1 \leq i \leq n$ and $1 \leq j \leq |t_i|$), if $s(i) = 1$ (resp. $t_i(j) = 1$), we give p_{2i-1} (resp. $q_{i,2j-1}$) two neighbours (called the *marking vertices*) with colour 4. For each of the marking vertices, we create a neighbour with colour 3.

This construction is depicted in Figure 4.5. In this example $s = 10100$, $t_1 = 01$ and $t_2 = 001$. Note that this instance of Orthogonal Vector Crafting is illustrative, and does not satisfy the restrictions required in the proof.

Informally, the construction works as follows: the barriers at the end of the path of p-vertices cannot be passed by the remaining vertices, meaning we have to "weave" the shorter q-paths into the long p-path. The colours enforce that the paths are in "lockstep", that is, we have to traverse them at the same speed. We have to map every q-vertex with colour 3 to a p-vertex with colour 2, but the marking vertices prevent us from doing so if both bitstrings have a 1 at that particular position.

Lemma 4.15. *G can be intervalized if the* Orthogonal Vector Crafting *instance has a solution.*

Proof. As an example, Figure 4.6 shows how the graph from Figure 4.5 may be intervalized. Let Π be a solution to the instance of Orthogonal Vector Crafting. We can intervalize the barriers as depicted in Figure 4.4b, noting that the right barrier should be intervalized in a mirrored fashion. The connector vertex (which is the only remaining vertex of colour 5) can be assigned an interval that covers the entire interval between the two barrier gadgets. If no marker vertices are present, then we can weave the q-paths into the p-path as depicted in Figure 4.6, whereby each interval corresponding to a q-vertex of colour 3 completely contains the interval of a p-vertex of colour 2 (note that the endpoint vertices of the q-paths are treated differently from the q-path vertices with colour 3). If we intervalize the q-paths in the same order the corresponding strings appear in Π, then we can also intervalize the marking vertices: Figure 4.6 shows that the marker vertices can also be intervalized, so long as a p-vertex and its corresponding q-vertex are not both adjacent to marker vertices, but this is guaranteed by the orthogonality of s and t^{Π}. □

Lemma 4.16. *G can be intervalized only if the* Orthogonal Vector Crafting *instance has a solution.*

Proof. Suppose we are given a properly coloured interval supergraph of G that assigns to each vertex $v \in V$ its left endpoint $l(v)$ and right endpoint $r(v)$. For vertices $u, v \in V$, we write $v \subset u$ if $l(u) < l(v) < r(v) < r(u)$, and we write $v < u$ if $r(v) < l(u)$. We write $v \lessdot u$ if $l(v) < l(u) < r(v) < r(u)$ - that is, the interval of v starts to the left of the interval of u and the two intervals overlap.

We may without loss of generality assume that $C_l < C_r$ and that no two endpoints of intervals coincide.

Claim. For any non-barrier vertex v, we have that $r(C_l) < l(v) < r(v) < l(C_r)$.

Proof. Examining the situation for the left barrier, we see that a clique vertex cannot be contained completely inside the interval of the center vertex C_l since it is adjacent to another vertex with colour 1 (whose interval may not intersect that of the center).

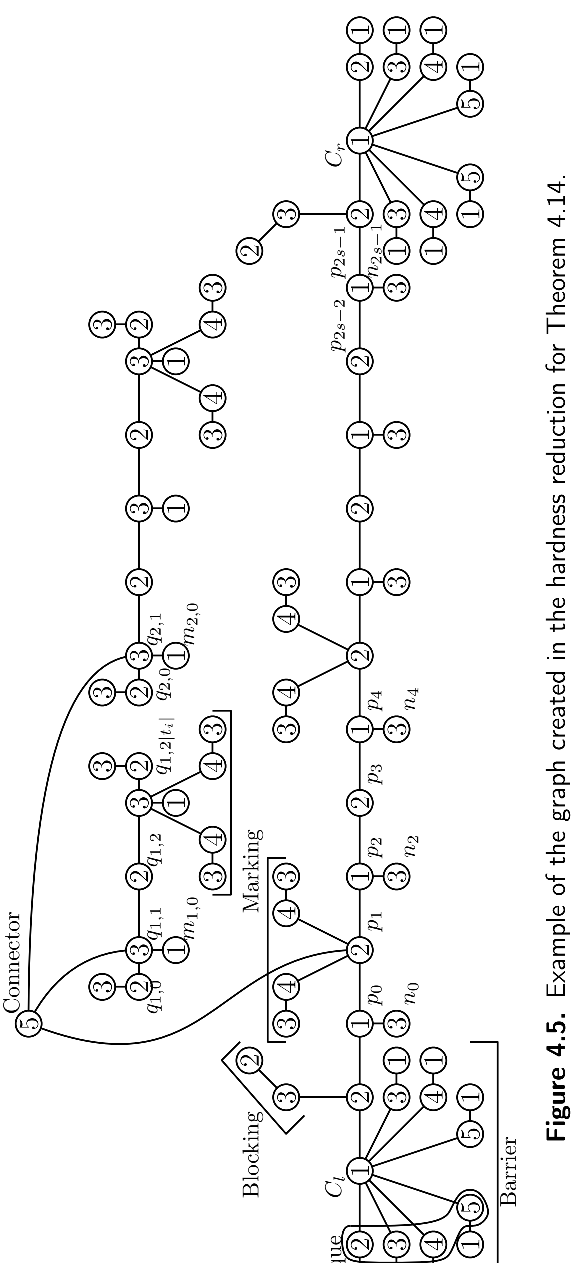


Figure 4.5. Example of the graph created in the hardness reduction for Theorem 4.14.

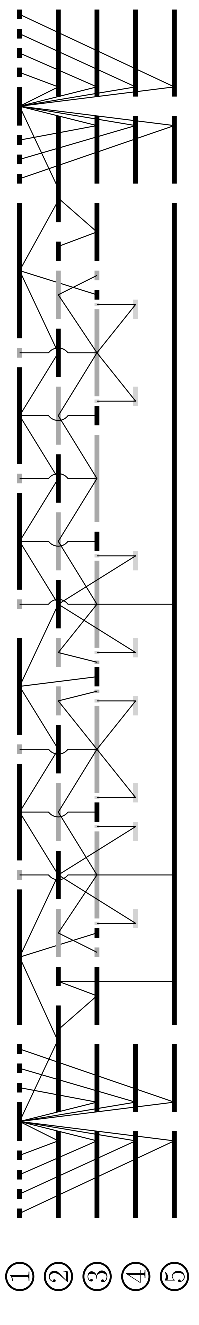


Figure 4.6. How the graph from Figure 4.5 may be intervalized. Thick lines represent intervals and are arranged vertically based on the colour of the associated vertex. The thinner lines indicate the edges of the original graph. Black intervals correspond to the barriers, p, n-vertices and the connector vertex, light gray intervals to marker vertices and dark gray intervals to end point vertices and to q, m-vertices.

Since there are two clique vertices of each colour, for each colour, the interval of one clique vertex of that colour must extend to the left of the central vertex' interval and the other must extend to the right. Therefore, these intervals that contain $r(c_l)$ induce a clique of size 5. Since we are looking for a 5-coloured supergraph, no other intervals can contain $r(c_l)$.

Note that the clique vertices are interchangeable, except the ones coloured 2: the clique vertex that is adjacent to p_0 must go to the right of the center vertex, since otherwise the path from it to C_r could not pass the clique at $r(c_l)$.

Suppose that for some non-barrier vertex v, it holds that $l(v) \leq r(c_r)$. This is not possible, since the path from v to c_r cannot pass the clique induced at $r(c_l)$. Therefore, $r(c_l) < l(v)$.

The case for the right barrier is symmetric. □

Claim. For all $0 \leq i < 2|s| - 1$, we have that $p_i \lessdot p_{i+1}$. Furthermore, $C_l \lessdot e_l \lessdot p_0$ and $p_{2|s|-1} \lessdot e_r \lessdot C_r$.

Proof. The fact that $C_l \lessdot e_l \lessdot p_0$ (resp. $p_{2|s|-1} \lessdot e_r \lessdot C_r$) follows from the analysis of the barrier gadget in the previous claim. We now proceed by induction, for the purposes of which we shall write $e_l = p_{-1}$ and $e_r = p_{2|s|}$: suppose the claim holds for $i-1$, i.e. $p_{i-1} \lessdot p_i$. It cannot hold that $p_{i+1} \subset p_i$, since it is adjacent to p_{i+2}, nor can it hold that $p_{i+1} \lessdot p_i$ (since then it would intersect the interval of p_{i-1}. □

For any $1 \leq i \leq n$, a similar fact holds for the path $q_{i,0}, \ldots, q_{i,2|t_i|}$:

Claim. We may without loss of generality assume that $q_{i,0} \lessdot \ldots \lessdot q_{i,2|t_i|}$.

Proof. By a similar induction argument as above, it must hold that either $q_{i,0} \lessdot \ldots \lessdot q_{i,2|t_i|}$ or $q_{i,0} \gtrdot \ldots \gtrdot q_{i,2|t_i|}$. However, since t_i is palindromic, these two cases are equivalent. □

Claim. Let $1 \leq i \leq n, 1 \leq j \leq |t_i|$. Then there exists $1 \leq k \leq |s|$ such that $p_{2k-1} \subset q_{i,2j-1}$.

Proof. The interval of $q_{i,2j-1}$ cannot be completely contained in the interval of a vertex with colour 1, since $q_{i,2j-1}$ is adjacent to $m_{i,2j-1}$ which has colour 1 as well. Therefore (since $r(C_l) < l(q_{i,2j-1}) < r(q_{i,2j-1}) < l(C_r)$), the interval of $q_{i,2j-1}$ must intersect the interval of either a barrier endpoint or a path vertex p_{2k-1} for some $1 \leq k \leq |s|$. It is not possible that $q_{i,2j-1}$ intersects the interval of a barrier endpoint, since (due to the blocking and clique vertices with colour 3, see Figure 4.4) it would have to be completely contained inside this interval, which is impossible since $q_{i,2j-1}$ is adjacent to vertices with colour 2. Therefore there exists a $1 \leq k \leq |s|$ such that the interval of $q_{i,2j-1}$ intersects that of p_{2k-1}.

Since $q_{i,2j-2} \lessdot q_{i,2j-1} \lessdot q_{i,2j}$ and $c(q_{i,2j-2}) = c(q_{i,2j}) = c(p_{2k-1}) = 2$ we must have that $q_{i,2j-2} < q_{2k-1} < q_{i,2j}$. It now follows that $p_{2k-1} \subset q_{i,2j-1}$. □

This allows us to define the *position* $1 \leq P(i,j) \leq |s|$ for each $1 \leq i \leq n, 1 \leq j \leq |t_i|$, which is equal to the k from the previous claim. Note that P is a bijection, since each interval p_{2k-1} is the subset of (the interval of) exactly one q-vertex (each p-vertex interval cannot be the subset of more than one q-vertex interval, and the number of p-vertices is such that no q-vertex can completely contain more than one q-vertex interval).

Claim. Let $1 \leq i \leq n, 1 \leq j \leq |t_i| - 1$. Then $P(i, j+1) = P(i, j) + 1$.

Proof. This follows from the fact that a q-vertex with colour 3 cannot completely contain a p-vertex with colour 1 (since it has an n-vertex as neighbour that has colour 1 as well) and the fact that a q-vertex with colour 2 cannot overlap a p-vertex with colour 2.

Formally, let $P(i, j) = k$, then $p_{2k-1} \subset q_{i,2j-1}$. Since $q_{i,2j-1} \lessdot q_{i,2j} \lessdot q_{i,2j+1}$, if the claim does not hold, we must have $p_{2k+1+2m} \subset q_{i,2j+1}$ for some $m > 0$. We must have $q_{i,2j} \subset p_{2k+2r}$ for some $r \geq 0$. If $r = 0$, then $q_{i,2j} < q_{i,2j+1}$ which is a contradiction. On the other hand, if $r > 0$ then $q_{i,2j-1} < q_{i,2j}$ which is also a contradiction. □

Claim. Let Π be the permutation of $\{1, \ldots, n\}$ such that $P(idx_\Pi(i), pos_\Pi(i)) = i$. Then Π exists and is a solution to the ORTHOGONAL VECTOR CRAFTING instance.

Proof. The existence of Π follows from the previous claim. Suppose that Π is not a solution. Then there exists an i, such that $S(i) = T_{idx_\Pi(i)}(pos_\Pi(i)) = 1$. However, this means that $p_{2i-1} \subset q_{idx_\Pi(i), 2pos_\Pi(i)-1}$. Since both p_{2i-1} and $q_{idx_\Pi(i), 2pos_\Pi(i)-1}$ have marking vertices, this is impossible as the marking vertices with colour 4 would have overlapping intervals. □

This completes the proof of Lemma 4.16. □

The number of vertices of G is linear in $|s|$ and we thus obtain a $2^{\Omega(n/\log n)}$ lower bound under the Exponential Time Hypothesis. □

Note that the graph created in this reduction only has one vertex of super-constant degree. This is tight, since the problem is polynomial-time solvable for bounded degree graphs (for any fixed number of colours) [72].

To complement this result for a bounded number of colours, we also show a $2^{\Omega(n)}$-time lower bound for graphs with an unbounded number of colours, assuming the ETH. Note that this result implies that the algorithm from [28] is optimal. A complication in the proof is that to obtain the stated bound, one can only use (on average) a constant number of vertices of each colour (when using $O(n)$ colours). A variation on the previous proof whereby instead of using bitstrings, colours are used to identify clauses and variables is thus unlikely to work since one would need to repeat each colour many times (in each place where a particular bitstring does not fit).

Theorem 4.17. *Assuming the Exponential Time Hypothesis, there is no algorithm solving* INTERVALIZING COLOURED GRAPHS *in time $2^{o(n)}$, even when restricted to trees.*

Proof. By reduction from EXACT COVER BY 3-SETS (X3C). X3C is the following problem: given a set X with $|X| = n$ and a collection M of subsets $X_1, \ldots, X_m$ of size 3, decide whether there exists a subset M' of size $n/3$ such that $\bigcup M' = X$. Assuming the Exponential Time Hypothesis, there does not exist an algorithm solving X3C in time $2^{o(m)}$ (See e.g. [25, 9, 55]. Note that 3-Dimensional Matching is a special case of X3C.).

The intuition behind the reduction is that, to leverage the large number of colours, one needs to force the creation of a very large clique in the intervalized graph. For each element of $X_i \in M$, we create a component of which the vertices have colours that correspond to the elements of X_i. The graph created in the reduction will be so that all but $n/3$ of the components corresponding to elements of M can be placed in distinct intervals, but the remaining $n/3$ components will have to overlap. This, in turn, is only possible if no two components in this latter collection contain duplicated colours, that is, each element is represented at most (and thus exactly) once in the selected $n/3$ components.

We assume the elements of X are labelled $1, \dots, n$. We may assume that n is a multiple of 3 (or we may return a trivial no-instance) and that $m > n/3$ (or we can check immediately whether M covers X).

The graph created in the reduction has $n + 4$ colours: two colours e_i and f_i for each $1 \leq i \leq n$ and four additional colours a, b, c, d.

We construct the graph G as follows: we start with a path of $2(m-n/3)+1$ vertices $p_0, \dots, p_{2(m-n/3)}$, where p_i has colour a if i is even and p_1 has colour b if i is odd. To $p_{2(m-n/3)}$ we make adjacent a vertex with colour d, which we shall refer to as the *right barrier* vertex.

Next, for each $1 \leq i \leq n$, we create a vertex v_i with colour e_i, that is made adjacent to p_0. For each i, we create an additional vertex with colour d that is made adjacent to v_i. These vertices (with colour d) are called the *left barrier* vertices.

Next, for each 3-set $X_i \in M$, we create a *set component*, consisting of a *central vertex* with colour a, to which are made adjacent: two vertices with colour c, each of which is adjacent to a vertex with colour b and, for each of the three elements of X_i, two vertices with the corresponding (to that element) colour f_i, each of which is made adjacent to a vertex with colour e_i.

Finally, we connect all the components together with a *connector vertex* of colour d, which is made adjacent to each set gadget and to a vertex of the path (for instance to p_0). Figure 4.7 provides an example of the construction.

As with the case for five colours, a solution somehow has to "pack" the set components into the part of the graph between the barriers: $m - n/3$ of the set components can be packed into the path (each interval corresponding to a b-vertex can hold at most one component); the remaining $n/3$ components have to be packed between the left barrier vertices and p_0 and this is possible only if the corresponding sets are disjoint (otherwise intervals of the corresponding colours would have to overlap).

Lemma 4.18. *If G can be intervalized, then the* X3C *instance has a solution.*

Proof. In any interval supergraph of G, the interval of the connector vertex should lie between the intervals of the left barrier vertices and the right barrier vertex. Let $X_i \in M$. The interval of the central vertex of the set component corresponding to X_i must either be contained in the interval corresponding to some vertex p_i for some odd $1 \leq i < 2(m - n/3)$ or it should be contained in the intersection of the intervals corresponding to the vertices $\{v_j \mid 1 \leq j \leq n\}$ (since its interval cannot intersect any interval of a vertex with colour a, nor can its interval be contained in the interval of a

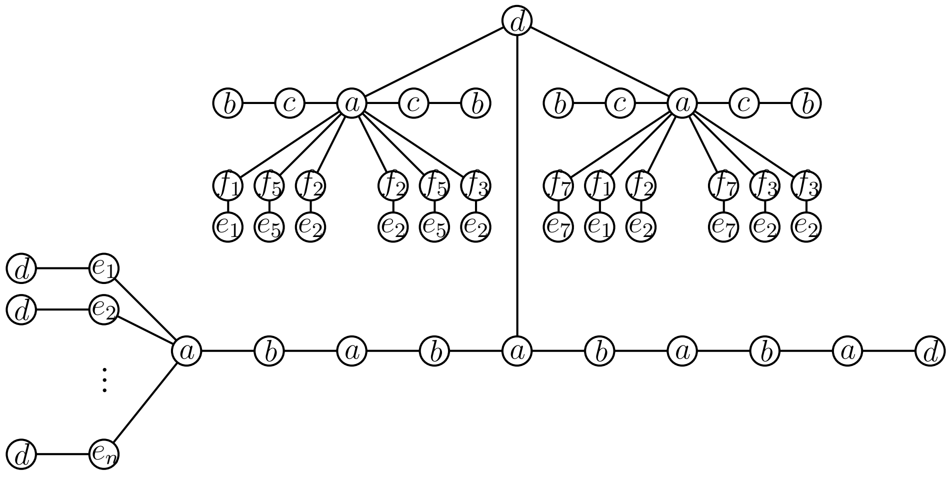


Figure 4.7. The construction used in the proof of Theorem 4.17. In this example, $X = \{\{1, 5, 2\}, \{7, 1, 2\}, \ldots\}$. The topmost vertex with colour d is the connector vertex, and its degree would increase as more elements are added to X.

left or right barrier vertex).

Claim. At most one interval corresponding to a central vertex can be contained in the interval of any vertex p_i (for odd i).

Proof. Each central vertex is adjacent to two vertices with colour c. Since these vertices in turn have neighbours with colour b, it follows that (if some central vertex is contained in the interval of p_i) the interval of one of the vertices with colour c must contain the left endpoint of the interval of p_i, and the other must contain the right endpoint. Therefore the interval of p_i cannot contain another central vertex. □

Claim. Let $X_i \neq X_j \in M$. If $X_i \cap X_j \neq \emptyset$, then the intervals of the central vertices of the set components corresponding to X_i and X_j cannot both be contained in the intersection of the intervals corresponding to the vertices $\{v_k \mid 1 \leq k \leq n\}$.

Proof. Since $X_i \cap X_j \neq \emptyset$, both central vertices have two neighbours with colour f_m for some $1 \leq m \leq n$. Each of these vertices has a neighbour with colour e_m, and thus the interval of a vertex with colour f_m must contain either the left or the right endpoint of the interval that is the intersection of the intervals corresponding to the vertices $\{v_k \mid 1 \leq k \leq n\}$. This is not possible, since either the left or the right endpoint of this intersection will be contained in more than one interval corresponding to a vertex with colour f_m. □

We thus see that the elements of M that correspond to set components whose intervals are contained in the intersection of the intervals corresponding to the vertices $\{v_k \mid 1 \leq k \leq n\}$ form a solution: no element of X is contained in more than one of them. As at most $|X| - n/3$ intervals of set components are contained in intervals corresponding to some vertex v_i, we have that at least $n/3$ set components have intervals that are contained in the aforementioned intersection. These must thus form a solution to the X3C instance. □

Lemma 4.19. *If the* X3C *instance has a solution, then G can be intervalized.*

Proof. The v-vertices are assigned to identical intervals. Their d-coloured neighbours can be assigned arbitrarily small intervals that are placed in the left half of the v-vertex interval. The p-vertices can then be placed from left to right, so that p_0 overlaps a small portion of the right end of the v-vertex intervals, and each p vertex interval overlaps the interval of the preceding p-vertex slightly. Finally the right barrier vertex (with colour d) should be placed in the right half of the interval corresponding to $p_2 2(m - n/3)$.

Next, the connector vertex (with colour d) can be assigned a long interval between the left and right barrier vertices, that overlaps the v-vertex intervals and all of the p-vertex intervals. This placement of the connector vertex allows us to place the intervals of set components anywhere between the left and right barriers.

Let $M' \subseteq M$ be a solution to the X3C instance. Since $|M'| = n/3$, and there are $m - n/3$ p-vertices with colour b, we can assign each element of M that is not in the solution a unique p-vertex with colour a. We assign the central vertex (which has colour a) of the set component of each such element an interval inside the interval of its p-vertex, such that it does not intersect neighbouring p-vertices (which have colour a). The vertices with f- or e-colours of the set component can be assigned similar intervals (not intersecting neighbouring p-vertices). One of the vertices with colour c is assigned an interval that extends past the left endpoint of the p-vertex interval (and thus intersects the preceding p-vertex interval), which allows us to assign its neighbour with colour b an interval that is contained in the preceding p-vertex interval (and does not intersect any other p-vertex interval). The other vertices with colours b and c can be placed similarly on the right.

Finally, for the set components corresponding to elements of M', we can assign the vertices with colours a, b, c arbitrarily small intervals in the right half of the v-vertex intervals; the f-coloured vertices can be placed so that their intervals stick out beyond the right and left endpoints of the v-vertex intervals, so that the e-coloured vertices can be placed not overlapping the v-vertex intervals. The fact that M' is a solution guarantees this can be done without any e, f-colours overlapping each other, since each such colour occurs exactly twice (one such pair of vertices can be placed on the left, the other on the right). □

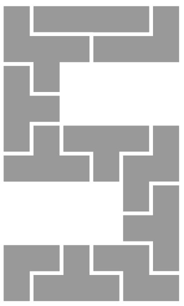
This completes the reduction. Since the number of vertices of G is linear in $|M|$ and $|X|$, we see that Intervalizing Coloured Graphs does not admit a $2^{o(n)}$-time algorithm, unless the Exponential Time Hypothesis fails. □

4.7. Conclusions

In this chapter, we have shown for several problems that, under the Exponential Time Hypothesis, $2^{\Theta(n/\log n)}$ is the best achievable running time – even when the instances are very restricted (for example in terms of pathwidth or planarity). For each of these problems, algorithms that match this lower bound are known and thus $2^{\Theta(n/\log n)}$ is (likely) the asymptotically optimal running time.

For problems where planarity or bounded treewidth of the instances (or, through bidimensionality, of the solutions) can be exploited, the optimal running time is often $2^{\Theta(\sqrt{n})}$ (or features the square root in some other way). On the other hand, each of problems studied in this chapter exhibits some kind of "packing" or "embedding" behaviour. For such problems, $2^{\Theta(n/\log n)}$ is often the optimal running time. We have introduced two artificial problems, String Crafting and Orthogonal Vector Crafting, that form a useful framework for proving such lower bounds.

It would be interesting to study which other problems exhibit such behaviour, or to find yet other types of running times that are "natural" under the Exponential Time Hypothesis. The loss of the $\log n$-factor in the exponent is due to the fact that $\log n$ bits or vertices are needed to "encode" n distinct elements; it would be interesting to see if there are any problems or graph classes where a more compact encoding is possible (for instance only $\log^{1-\epsilon} n$ vertices required, leading to a tighter lower bound) or where an encoding is less compact (for instance $\log^2 n$ vertices required, leading to a weaker lower bound) and whether this can be exploited algorithmically.



Intermezzo: Polyomino Packing

5.1. Introduction

In this chapter, we show that the problem of deciding whether a collection of polyominoes, each fitting in a $2 \times O(\log n)$ rectangle, can be packed into a $3 \times n$ box does not admit a $2^{o(n/\log n)}$-time algorithm, unless the Exponential Time Hypothesis fails. We also give an algorithm that attains this lower bound, solving any instance of polyomino packing with total area n in $2^{O(n/\log n)}$. This establishes a tight bound on the complexity of Polyomino Packing, even in a very restricted case. In contrast, for a $2 \times n$ box, we show that the problem can be solved in strongly subexponential time.

The complexity of games and puzzles is a widely studied topic, and the complexity of most games and puzzles in terms of completeness for a particular complexity class ($\mathcal{NP}$, $\mathcal{PSPACE}$, $\mathcal{EXPTIME}$, ...) is generally well-understood (see e.g. [64] for an overview). Results in this area are not only mathematically interesting and fun, but are also a great educational tool for teaching hardness reductions. However, knowing that a game or puzzle is $\mathcal{NP}$-complete does not provide a very detailed picture: it only tells us that there is unlikely to be a polynomial-time algorithm, but leaves open the possibility that there might be a very fast superpolynomial but subexponential-time algorithm. This issue was precisely the motivation for introducing the Exponential Time Hypothesis [67].

We study the Polyomino Packing problem from the viewpoint of exact complexity. We give a reduction from 3-SAT, showing that Polyomino Packing cannot be solved in $2^{o(n/\log n)}$ time, even if the target shape is a $3 \times n$ rectangle and each piece fits in a $2 \times O(\log n)$ rectangle. As the reduction is self-contained, direct from 3-SAT and rather elegant, it could be an excellent example to use for teaching. We also show that this is tight: Polyomino Packing can be solved in $2^{O(n/\log n)}$ time for any set of polyominoes of total area n that have to be packed into any shape.

Polyomino Packing appears to behave similarly to Subgraph Isomorphism on planar graphs, which, as shown in the previous chapters, has exact complexity $2^{\Theta(n/\log n)}$ (i.e., there exists an algorithm solving the problem in $2^{O(n/\log n)}$ time on n-vertex graphs, and unless the ETH fails there is no $2^{o(n/\log n)}$-time algorithm).

Demaine and Demaine [40] showed that packing n polyominoes of size $\Theta(\log n) \times \Theta(\log n)$ into a square box is $\mathcal{NP}$-complete. This result left open a gap, namely of whether the problem remained $\mathcal{NP}$-complete for polyominoes of *area* $O(\log n)$. This gap was recently closed by Brand [30], who showed that Polyomino Packing is $\mathcal{NP}$-complete even for polyominoes of size $3 \times O(\log n)$ that have to be packed into a square. However, Brand's construction effectively builds up larger (more-or-less square) polyominoes by forcing smaller (rectangular) polyominoes to be packed together in a particular way, by using jagged edges that correspond to binary encodings of integers to enforce that certain pieces are placed together.

Our reduction also uses binary encoding of integers to force that various pieces are placed together. However, in contrast, it gives hardness for a much more restricted case (packing polyomino pieces of size $2 \times O(\log n)$ into a rectangle of height 3) and also reduces directly from 3-SAT, avoiding the polynomial blowup incurred by Brand's reduction from 3-Partition, thus giving a tight (under the Exponential Time Hypothesis) lower bound. As 3-Partition is a frequently used tool for showing hardness of various types of packing puzzles and games, we believe that these techniques could be used to give (tight, or at least strong) lower bounds on the complexity of other games and puzzles.

This result is tight in another sense: we show that Polyomino Packing where the target shape is a $2 \times n$ rectangle admits a $2^{O(n^{3/4}\log n)}$-time algorithm, so $3 \times n$ is the smallest rectangle in which a $2^{\Omega(n/\log n)}$-time lower bound can be attained.

Note that our results are agnostic to the type (free, fixed or one-sided) of polyomino used. That is, it does not matter whether we are able to rotate (one-sided), rotate and flip (free) or not (fixed) our polyominoes. Our reduction creates instances whose solvability is preserved when changing the type of polyomino, while the algorithms can easily be adapted to work with any type of polyomino. In the following, we consider the Polyomino Packing problem, which asks whether a given set of polyominoes can be packed to fit inside a given target shape. If we include the additional restriction that the area of the target shape is equal to the total area of the pieces, we obtain the Exact Polyomino Packing problem.

5.2. Lower Bounds

Theorem 5.1. *Unless the Exponential Time Hypothesis fails, there exists no $2^{o(n/\log n)}$-time algorithm for* Polyomino Packing, *even if the target shape is a $3 \times n$ box, and the bounding box of each polyomino is of size $2 \times \Theta(\log n)$.*

Proof. A weaker version of the statement follows by a simple reduction from the Orthogonal Vector Crafting problem from the previous chapter. However, because obtaining the bound on the piece size requires a deeper understanding of the proof, and to illustrate the technique, we give a different proof that is nevertheless somewhat similar to the reduction from the previous chapter.

We proceed by reduction from n-variable 3-SAT, which, unless the Exponential

Time Hypothesis fails, does not admit a $2^{o(n)}$-time algorithm. By the Sparsification Lemma [68], we can assume that the number of clauses $m = O(n)$.

Using the following well-known construction, we can furthermore assume that each variable occurs as a literal at most 3 times: replace each variable x_i that occurs $k > 3$ times by k new variables $x_{i,1}, \ldots, x_{i,k}$ and add the clauses $(\neg x_{i,1} \vee x_{i,2}) \wedge (\neg x_{i,2} \vee x_{i,3}) \wedge \ldots \wedge (\neg x_{i,k-1} \vee x_{i,k}) \wedge (\neg x_{i,k} \vee x_{i,1})$. This only increases the total number of variables and clauses linearly (assuming we start with a linear number of clauses).

We remark that our construction works for general SAT formulas. The Sparsification Lemma is only needed to achieve the stated $2^{\Omega(n/\log n)}$ lower bound, and the bound on the number of occurrences of a variable is only needed to obtain the bound on the piece size.

Our construction will feature three types of polyomino: *n formula-encoding polyominoes*, *n variable-setting* polyominoes and *m clause-checking* polyominoes. We number the variables of the input formula $1, \ldots, n$ and the clauses $n+1, \ldots, n+m$. With every clause or variable we associate a bitstring of length $22 + 4\lceil \log(n+m) \rceil$, which is obtained by taking the binary representation of that clause/variable's number, padding it with 0's to obtain a bitstring of length $\lceil \log(n+m) \rceil$, replacing every 0 by 01 and every 1 by 10 (thus ensuring the number of 1's in the bitstring is equal to the number of 0's, and that the bitstring contains at most 2 consecutive zeroes or ones) and then appending a reversed copy of the bitstring to itself (making it palindromic). Finally, we prepend 11110001111 and append 11110001111 (note that thus the start and end of the bitstring is the only place to feature 3 or more consecutive 0's).

For any bitstring, we can create a *corresponding polyomino*: given a bitstring of length k, its corresponding polyomino fits in a $2 \times k$ rectangle, whose top row consists of k squares, and whose bottom row has a square whenever the bitstring has a 1 in that position. For each such polyomino, we can also create a *complementary polyomino* that mates with it to form a $3 \times k$ rectangle (which can also be seen as a flipped version of the polyomino corresponding to the complement of the bitstring, i.e., the bitstring with all zeroes replaced by ones and vice-versa). Figure 5.1 shows several example corresponding polyominoes and their complements. Note that since the bitstrings are palindromic, the thus created polyominoes are achiral, i.e., invariant over being flipped.

We can *concatenate* two polyominoes corresponding to bitstrings b_1, b_2 by taking the polyomino corresponding to the concatenation of the two bitstrings $b_1 b_2$.

Note that the polyomino corresponding to a variable or clause can only mate with its complementary polyomino, it cannot fit together with any polyomino corresponding

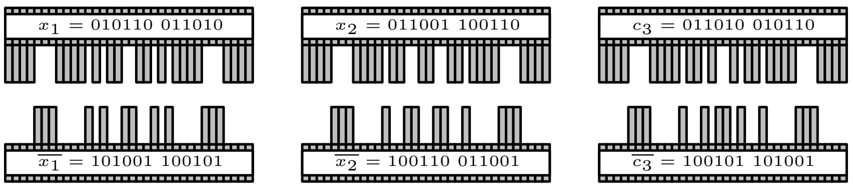

x_1 = 010110 011010
x_2 = 011001 100110
c_3 = 011010 010110
$\overline{x_1}$ = 101001 100101
$\overline{x_2}$ = 100110 011001
$\overline{c_3}$ = 100101 101001

Figure 5.1. Top: polyominoes corresponding to variables x_1, x_2 and clause c_3. Bottom: the complementary polyominoes, that mate with the polyominoes above them to form a $3 \times k$ square. Note that the polyominoes are depicted compressed horizontally.

to any other variable or clause or the complement thereof. Our construction uses as building blocks two more polyominoes: the *wildcard polyomino*, which is obtained as the polyomino corresponding to the bitstring 00001110000000 . . . 00000001110000 ($4\lceil \log{(n+m)} \rceil$ zeroes surrounded by 00001110000) and the *blocking polyomino*, which is the complementary polyomino for the wildcard. Note that the wildcard polyomino fits together with any clause or variable polyomino, while the blocking polyomino only fits together with the wildcard polyomino.

Since each variable occurs as a literal at most three times, we can assume that it appears at most twice in positive form, and at most twice negated (if the variable occurs exclusively positively or negated we can simply remove the clauses that contain it to obtain an equivalent instance).

We are now ready to define the *formula-encoding polyominoes*. The construction will have n variable-encoding polyominoes, one for each variable x_i, and each consists of the concatenation of 7 polyominoes: we start with a polyomino corresponding to the bitstring of x_i. Next, for each time (at most two) x_i occurs positively in a clause, we take a polyomino corresponding to (the bitstring of) that clause. If x_i occurs only once in positive form, then we take (for padding) a copy of the blocking polyomino. Then, we take another copy of the polyomino for x_i. Next, we take the polyominoes corresponding to clauses in which x_i occurs negated. Again, we add the blocking polyomino if x_i only occurs negated once. Finally, we take another copy of the polyomino corresponding to x_i.

The *variable-setting polyomino* for x_i is the polyomino formed by concatenating, in the following order: (a) the complement polyomino for the variable, (b) 2 copies of the wildcard polyomino, (c) another copy of the complement polyomino.

The *clause-checking polyominoes* are simply the following: for each clause, we take a polyomino corresponding to the complement of its bitstring.

This completes the construction. An example of the construction is shown in Figure 5.2. Note that if fixed or one-sided polyominoes are used, the formula-encoding ones are provided with the solid row of squares on top, and the remaining polyominoes are provided with the solid row on the bottom. We claim this set of polyominoes can be packed into a $3 \times 7n(22 + 4\lceil \log{(n+m)} \rceil)$ box if and only if the formula is satisfiable.

($\Rightarrow$). Suppose the polyominoes can be packed in a $3 \times 7n(22 + 4\lceil \log{(n+m)} \rceil)$ box. We first examine the placement of the formula-encoding polyominoes. Because each formula-encoding polyomino starts with a row of four ones, and the largest "gap" of zeroes occurring in one is of length three, they cannot overlap vertically; each formula-encoding polyomino must be fully to the right of the previous. Moreover, since the width of the target rectangle matches exactly the total width of the formula-encoding polyominoes, they must be placed back-to-back in some arbitrary permutation.

Consider the placement of a single complementary polyomino for a clause or variable. Because wherever two formula-encoding polyominoes touch back-to-back there are 8 consecutive rows in which 2 squares are already occupied, and the longest "gap" in a complementary polyomino is of length at most 5 (and at the left and right edges, there is a gap of length exactly 4, we see that the rows in which this polyomino are placed can contain only a single formula-encoding polyomino. This rules out any undesirable shifts: no complementary polyomino can overlap (vertically) more than one formula-encoding polyomino. Moreover, note that this same phenomenon forces the vertical alignment of polyominoes corresponding to variables or clauses in the

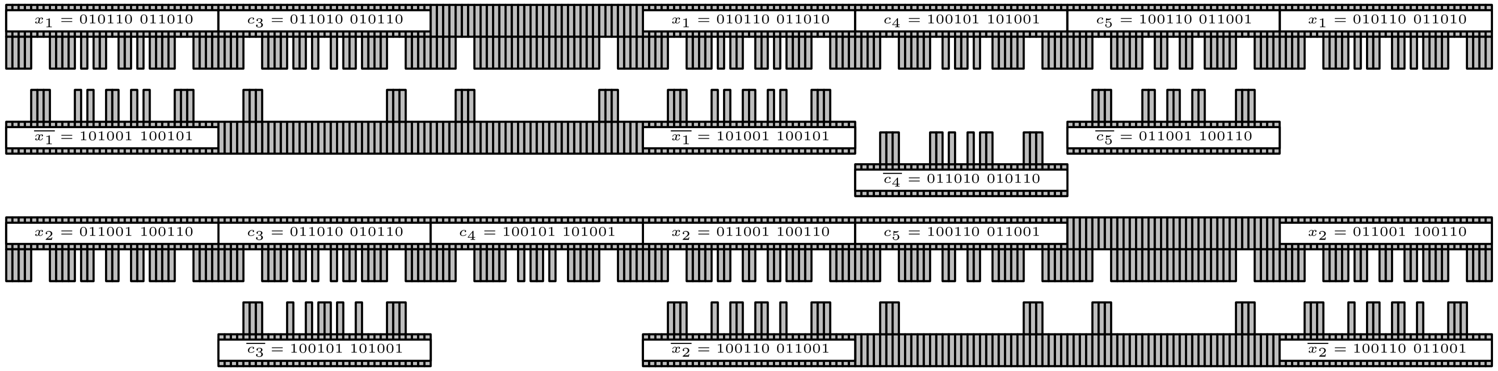


Figure 5.2. Example of our reduction for the formula $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$. Top-to-bottom, left-to-right: formula encoding polyomino for x_1, variable-setting polyomino for x_1, clause-checking polyomino for c_4, clause checking-polyomino for c_5, formula-encoding polyomino for x_2, clause-checking polyomino for c_3, variable-setting polyomino for x_2. The polyominoes are arranged in a way that suggests the solution $x_1 = false$, $x_2 = true$.

formula-encoding polyominoes with the complementary polyominoes in variable-setting and clause-checking polyominoes.

Now, consider the placement of a variable-setting polyomino (for variable x_i). Since it starts with a complementary polyomino for x_i, and also ends with one x_i, it too much be placed such that it only overlaps at most (and exactly) one formula-encoding polyomino, namely the one for x_i. It thus suffices to consider each formula-encoding polyomino in isolation. Note that then, there are only two possible placements for the variable-setting polyomino for variable x_i: either overlapping the first half of the formula-encoding polyomino, with the wildcard polyominoes used as building blocks in the variable-setting polyomino overlapping (and thus blocking) the polyominoes corresponding to clauses that are satisfied by setting x_i to $true$, or, overlapping the second half of the formula-encoding polyomino, overlapping (and thus blocking) the polyominoes corresponding to clauses that are satisfied by setting x_i to $false$.

Thus, the placement of the variable-setting polyominoes (unsurprisingly) corresponds to an assignment for the variables of the formula. It is easy to see that the clause-checking polyominoes can then be packed into the space left only if the assignment is satisfying: if the assignment does not satisfy some clause, then all the places where the respective clause-checking polyomino could fit are blocked by variable-setting polyominoes.

($\Leftarrow$). We can consider each formula-encoding polyomino in isolation. An assignment for the formula immediately tells us how to pack the variable-setting polyomino for x_i into the formula-encoding polyomino for x_i (namely: if x_i is true we place the variable-setting polyomino in the second half, otherwise, we place it in the first half of the formula-encoding polyomino). It is easy to see that if the assignment is satisfying, then for each clause-checking polyomino there is at least one possible placement inside a formula-encoding polyomino. For an example of how the pieces fit together for a satisfying assignment, see Figure 5.2. □

Remark that our reduction leaves gaps inside the packing. If we consider the variant of the problem where total area of the pieces is equal to the area of the target shape, and thus the entire rectangle must be filled (Exact Polyomino Packing), the instance can be padded with several 1×1 polyominoes to make the total area of the pieces equal to the area of the target rectangle.

Corollary 5.2. *Unless the Exponential Time Hypothesis fails, there exists no $2^{o(n/\log n)}$-time algorithm for* Exact Polyomino Packing, *even if the target shape is a $3 \times n$ box, and the bounding box of each polyomino is of size $2 \times O(\log n)$.*

This raises an interesting open problem: does Exact Polyomino Packing still admit a $2^{\Omega(n/\log n)}$-time lower bound when the pieces are similarly sized, that is, each piece must have area $\Theta(\log n)$ (or even just $\Omega(n)$). This seems to greatly limit the number of possible interactions between two polyomino pieces, since they cannot be combined in a way that creates small gaps.

Note that in the previous reduction we can fix the position of the formula-encoding polyominoes in advance. The problem then reduces to packing variable-setting and clause-checking polyominoes into the shape left when subtracting the formula-encoding polyominoes from the $3 \times n$ rectangle, which fits inside a $2 \times n$ rectangle. Doing so we obtain the following corollary:

Corollary 5.3. *Unless the Exponential Time Hypothesis fails, there exists no $2^{o(n/\log n)}$-time algorithm for* Polyomino Packing *(resp.,* Exact Polyomino Packing*), even if the target shape fits inside a $2 \times n$ box, and the bounding box of each polyomino is of size $2 \times \Theta(\log n)$ (resp., $2 \times O(\log n)$).*

5.3. Algorithms

Our lower bound applies in a rather constrained case: even for packing polyominoes with a bounding box of size $2 \times O(\log n)$ into a rectangle of size $3 \times n$, there is no $2^{o(n/\log n)}$-time algorithm. As we will show later, a similar lower bound cannot be established when the pieces are $1 \times k$ or $2 \times k$ rectangles (since the number of distinct such polyominoes is linear in their area rather than exponential). An interesting question, which we answer negatively, is whether a $2^{\Omega(n/\log n)}$-time lower bound can be obtained for packing polyominoes with a bounding box of size $2 \times O(\log n)$ into a rectangle of size $2 \times n$. Thus, the case for which we have derived our lower bound is essentially the most restrictive possible. Note that, while solvable in strongly subexponential time, this problem is $\mathcal{NP}$-complete, as can be seen by a simple reduction from 3-Partition.

We say that a polyomino is *Y-monotone* if every row consists of a number of contiguous squares, that is, there are no gaps.

Theorem 5.4. Polyomino Packing *for fixed, free or one-sided polyominoes can be solved in $2^{O(n^{3/4}\log n)}$ time if the target shape is a $2 \times n$ rectangle.*

Proof. First, consider a simple $O(2^n n^{O(1)})$-time dynamic programming algorithm that decides whether m polyominoes $p_1, \ldots, p_m$ can be packed into a target polyomino of area n: for any subset S of (the squares of) the target polyomino (there are 2^n such subsets) and $1 \leq k \leq m$, let $B(S, k)$ be the proposition "the polyominoes $p_k, p_{k+1}, \ldots, p_m$ can be packed into S". $B(S, m)$ is simply the proposition that S is the same polyomino as p_m; if $B(S, i-1)$ is known for all S then $B(S', i)$ can be computed by trying all (polynomially many) placements of p_i within S'.

If we are dealing with free or one-sided polyominoes we first guess how many (if any) of the 1×2 polyominoes should be used in the vertical orientation, and how many in the horizontal orientation. This thus converts them to fixed 1×2 or 2×1 polyominoes, and only increases the running time of the algorithm by a factor n.

We augment the previously presented algorithm with the following observation: when the target polyomino is a $2 \times n$ rectangle, and if we process the polyominoes in a fixed order, with the polyominoes that are $1 \times k$ rectangles being processed last (thus after the 2×1 polyominoes and any other polyominoes), then the target shapes considered by the dynamic programming algorithm are always the disjoint union of

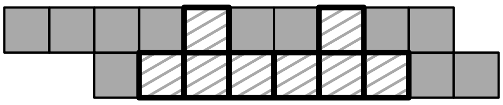


Figure 5.3. Packing an arbitrary $2 \times k$ polyomino into a Y-monotone polyomino results in several pieces that are again Y-monotone.

several Y-monotone polyominoes (c.f. Figure 5.3). Such polyominoes can be described by 3 integers: one giving the number of squares in the bottom row, one giving the number of squares in the top row, and one giving the shift of the top row relative to the bottom row. Note that this observation crucially depends on processing the $1 \times k$ polyominoes last, since removing them from a $2 \times k$ polyomino does not necessarily result in a shape that is Y-monotone, however, if only $1 \times k$ polyominoes remain, we can ensure this requirement remains satisfied because we can consider the top and bottom row of each polyomino in the target shape separately.

If each of these integers is (in absolute value) at most $n^{1/4} - 1$ we call the resulting polyomino *small*, otherwise, the polyomino is *large*. We can use the following more efficient description of the target shape: for each polyomino in the shape that is small, we give the number of such polyominoes in the target shape and we simply list each large polyomino. Since there are at most $2n^{3/4}$ distinct small polyominoes[1], giving the quantity for each leads to at most $(2n)^{2n^{3/4}} \leq 2^{2n^{3/4}(\log n+1)}$ cases. There are at most $2n^3$ distinct large polyominoes, but the target shape contains at most $2n^{3/4}$ of them (since each has area at least $n^{1/4}$), thus contributing $(2n^3)^{2n^{3/4}} \leq 2^{6n^{3/4}(\log n+1)}$ cases. Thus, if we identify equivalent target shapes, the dynamic programming algorithm needs to consider at most $2^{8n^{3/4}(\log n+1)}n = 2^{O(n^{3/4}\log n)}$ subproblems, and each subproblem can be handled in polynomial time. □

Note that this algorithm only works when the target shape is a $2 \times n$ rectangle. Corollary 5.3 shows that we should not expect a similar algorithm for packing polyominoes into an arbitrary target shape, even if that target shape fits in a $2 \times n$ box.

Finally, we show that our $2^{\Omega(n/\log n)}$-time lower bound is tight:

Theorem 5.5. Polyomino Packing *for free, fixed or one-sided polyominoes can be solved in $2^{O(n/\log n)}$ time if the target shape has area n.*

Proof. The problem can be modelled as Subgraph Isomorphism for an $O(n)$-vertex planar graph, for which a preceding chapter gave a $2^{O(n/\log n)}$-time algorithm. The construction is as follows: for every square in a polyomino, we take a cycle on four vertices, to which we add a fifth, universal vertex (which can be embedded in a planar embedding in the middle of this cycle). This fifth vertex is marked by adding a number of degree 1 vertices to it, to bring its degree up to (at least) 9. Each edge of this cycle is associated with an edge of the square in the polyomino. We make adjacent the endpoints of edges corresponding to adjacent edges in the polyomino. Both the host graph and the guest graph are constructed in this way, the host graph from the target shape (when viewed as a polyomino) and the guest graph from the set of input polyominoes (which will thus have one connected component corresponding to each separate polyomino that must be packed). An example for packing 3 polyominoes into a 3×4 rectangle is shown in Figure 5.4. The special (degree 9) vertices must be mapped to other vertices that are also degree 9, and this means that the cycles corresponding to squares can only be mapped to cycles corresponding to other squares (and not to cycles created by making cycles adjacent since those vertices have degree less than 9).

[1]The top and bottom rows can consist of $0, 1, \ldots, n^{1/4}-1$ squares, while the shift can be $-(n^{1/4}-2)$ to $n^{1/4} - 2$.

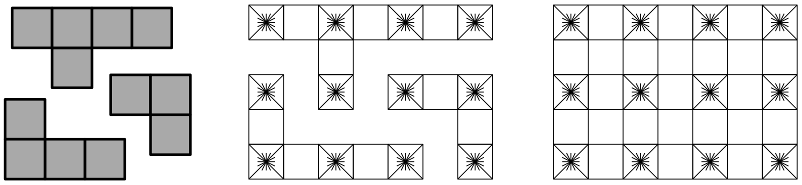


Figure 5.4. Polyomino Packing problem (left) modelled as Subgraph Isomorphism from pattern (middle) into host graph (right).

This construction works for free polyominoes. To restrict to fixed or one-sided polyominoes, we can modify the construction slightly to make the structure used to represent a square asymmetric. For one-sided polyominoes, we create a structure that is rotationally symmetric but achiral. To this end, we subdivide each edge of the cycle twice and identify one of the two vertices created by this subdivision, add another vertex, adjacent to this vertex, to its neighbours, and to the central vertex. For fixed polyominoes, we can add one additional edge (from the center to one of the vertices of the cycle to also remove the rotational symmetry. These constructions are depicted in Figure 5.5. □

To make this chapter more instructional, we give a direct proof of the following weaker version of Theorem 5.5 — which illustrates in a simpler way the principles from the previous chapters.

Theorem 5.6. Polyomino Packing *for free, fixed or one-sided polyominoes can be solved in* $2^{O(n/\log n)}$ *time if the target shape is a rectangle of area* n.

Proof. If the rectangle is higher than it is wide, rotate it (and, if the polyominoes are fixed, the polyominoes as well) 90 degrees. Consider a scanline passing over the rectangle from left to right. At any given time, the scanline intersects at most $O(\sqrt{n})$ squares of the rectangle. We can specify how the intersection of the solution with the scanline looks by, for each square, specifying the polyomino (if any) that is placed there, along with its rotation and translation with respect to the square. This gives at most $O(n^3)$ cases for each square, and, since the scanline intersects at most $\sqrt{n}$ squares, $2^{O(\sqrt{n}\log n)}$ cases total.

We furthermore need to specify which polyominoes have already been used in the solution (to the left of the scanline) and which ones still need to be packed. Similar to [26], a polyomino is *large* if it has area greater than $c \log n$, and small otherwise.

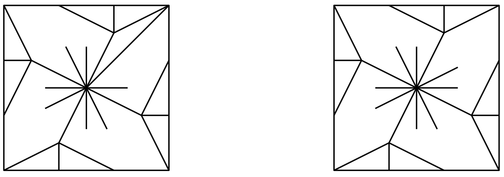


Figure 5.5. Alternative constructions to use with fixed (left) or one-sided (right) polyominoes.

Since the number of polyominoes with area k is bounded by 4.65^k [77], the number of distinct small polyominoes it at most $4.65^{c \log n}$. For $c \leq 0.22$, this is at most $\sqrt{n}$. We can specify the *quantity* of each small polyomino left with a single number from 0 to n, giving $(n+1)^{\sqrt{n}} = 2^{O(\sqrt{n} \log n)}$ cases. Meanwhile, the number of large polyominoes is at most $n/(c \log n)$, and thus there are $2^{O(n/\log n)}$ possible subsets of them.

The problem can now be solved by dynamic programming. For each position of the scanline, we have $2^{O(n/\log n)}$ subproblems: can a given subset of pieces ($2^{O(n/\log n)}$ cases) be packed entirely to the left of the scanline (with only the pieces intersecting the scanline possibly sticking out to the right of it), such that the intersection with the scanline looks as specified ($2^{O(\sqrt{n} \log n)}$ cases) (and, in the case of Exact Polyomino Packing, leaving no gaps)? For each such subproblem, we can find its answer by deleting the pieces whose leftmost square(s) intersect the scanline, and checking whether the instance thus obtained is compatible with some subproblem with the scanline moved one position to the left. $\square$

There is an interesting contrast between these algorithms. Whereas the strongly subexponential algorithm for the case of the $2 \times n$ rectangle works by considering the input polyominoes in a fixed order (so that we always know which subset we have used) and uses a bound on the number of subsets of the target shape that have to be considered, the algorithm for the general case works the opposite way around: it considers subsets of the target shape in a (more-or-less) fixed order (by the scanline approach) and bounds the number of possible subsets of the input polyominoes.

Note that our $2^{\Omega(n/\log n)}$-time lower bound exploits the fact that we can construct exponentially many polyominoes that fit inside a $2 \times O(\log n)$ rectangle. If we consider polyominoes with simpler shapes, that is, polyominoes that are $a \times b$ rectangles, then the problem can be solved in strongly subexponential time:

Corollary 5.7. Polyomino Packing *can be solved in* $2^{O(\sqrt{n} \log n)}$ *time if the polyominoes are rectangular and the target shape is a rectangle with area* n.

Proof. Consider the algorithm presented in the proof of Theorem 5.6. The running time is dominated by the number of cases for tracking a subset of the polyominoes. If the polyominoes are rectangles, then note that the number of distinct rectangles of area at most n is also at most n. Call a polyomino *large* if it has area $\geq \sqrt{n}$ and *small* otherwise: there are at most $\sqrt{n}$ large polyominoes in the input, and thus at most $2^{\sqrt{n}}$ subsets of them. The number of distinct small polyominoes is at most $\sqrt{n}$, and thus specifying the quantity for each leads to at most $n^{\sqrt{n}} = 2^{\sqrt{n} \log n}$ cases. $\square$

5.4. Conclusions

We have given a precise characterization of the complexity of (Exact) Polyomino Packing. For a set of polyominoes of total area n, the problem can be solved in $2^{O(n/\log n)}$ time. Even when restricted to the case where the pieces are of size $2 \times O(\log n)$ and they have to be packed into a $3 \times n$ rectangle or into a given shape which fits inside a $2 \times n$ rectangle, there is no faster (up to the base of the exponentiation) algorithm unless the Exponential Time Hypothesis fails. In contrast, in the case where the target shape is a $2 \times n$ rectangle, a strongly subexponential algorithm exists.

We conclude by listing several interesting open problems:

- Exact polyomino packing with excess pieces: we are given some target shape, and a set of polyominoes with total area possibly exceeding the target shape. Is it possible to use a subset of the polyominoes to build the target shape? Clearly this problem is at least as hard as (exact) polyomino packing; however, considering the set of pieces may be much larger than the target shape, it would be interesting to study this problem from a parameterized perspective (where the parameter k is the area of the target shape). The problem can be solved in $2^k n^{O(1)}$-time (using the simple dynamic programming algorithm of Section 5.3); is there a $2^{o(k)} n^{O(1)}$-time (or even $2^{o(k)} 2^{o(n/\log n)}$-time) algorithm?
- What is the complexity of Exact Polyomino Packing when every piece has area $\Omega(\log n)$ or $\Theta(\log n)$?
- We do not believe that our algorithm for packing polyominoes into a $2 \times n$ rectangle is tight. What is the exact complexity of this problem? This is closely related to the exact complexity of 3-Partition with the input given in unary, which (to our knowledge) is also an open problem.

II

Problems in Geometric Intersection Graphs

6 Problems in Geometric Intersection Graphs

6.1. Introduction

As noted in the introduction, many hard graph problems that seem to require $2^{\Omega(n)}$ time on general graphs (where n is the number of vertices) can be solved in subexponential time on planar graphs. In particular, many of these problems can be solved in $2^{O(\sqrt{n})}$ time on planar graphs — with a notable exception being the graph embedding problems we saw in the previous part.

In this chapter, we explore a different direction: rather than investigate whether *other problems* exhibit a "square root phenomenon" in planar graphs, we study whether classical problems (ones that are known to have square root behaviour in planar graphs, such as Independent Set, Vertex Cover and Hamiltonian Cycle) also exhibit square root behaviour in *other graph classes*.

One promising class of graphs where square root (or at least: strongly subexponential time) behaviour has also been observed, are geometric intersection graphs, with running times of the form $n^{O(n^{1-1/d})}$ (or in one case $2^{O(n^{1-1/d})}$) in the d-dimensional case having been obtained [91, 102].

The planar separator theorem [84, 85] and treewidth-based algorithms [34] could be considered a "framework" for obtaining subexponential algorithms on planar graphs or, more generally, on H-minor free graphs. In this chapter we aim to give a similar framework to obtain algorithms for problems in a wide class of geometric intersection graphs, while guaranteeing the running time $2^{O(n^{1-1/d})}$.

The *intersection graph* $G[F]$ of a set F of objects in $\mathbb{R}^d$ is the graph whose vertex set is F and in which two vertices are adjacent when the corresponding objects intersect. *(Unit-)disk graphs*, where F consists of (unit) disks in the plane are a widely studied class of intersection graphs. Disk graphs form a natural generalization of planar graphs, since any planar graph can be realized as the intersection graph of a set of disks in the plane. In this chapter we consider intersection graphs of a set F of *fat objects*, where an object $o \subseteq \mathbb{R}^d$ is *α-fat*, for some $0 < \alpha \leq 1$ if there are balls B_{in} and B_{out} in $\mathbb{R}^d$ such that $B_{\text{in}} \subseteq o \subseteq B_{\text{out}}$ and $\text{radius}(B_{\text{in}})/\,\text{radius}(B_{\text{out}}) \geq \alpha$. For example, disks are 1-fat and squares are $(1/\sqrt{2})$-fat. From now on we assume that α is an absolute constant, and often simply speak of fat objects. Note that we do not require the objects in F to be convex, or even connected. Thus our definition is very general. In most of our results we furthermore assume that the objects in F are *similarly sized*, meaning that the ratio of their diameters is bounded by a fixed constant.

As mentioned, many subexponential results for planar graphs rely on planar separators. Our first contribution is a generalization of this result to intersection graphs of (arbitrarily-sized) fat objects in $\mathbb{R}^d$. Since these graphs can have large cliques we cannot bound the number of vertices in the separator. Instead, we build a separator consisting of cliques. We then define a weight function γ on these cliques — in our applications it suffices to define the weight of a clique C as $\gamma(|C|) := \log(|C| + 1)$. We define the weight of a separator as the sum of the weights of its constituent cliques C_i, which is useful since for many problems a separator can intersect the solution vertex set in $2^{O(\sum_i \gamma(|C_i|))}$ many ways. Formally, the theorem can be stated this way:

Theorem 6.1. *Let F be a set of n α-fat objects in $\mathbb{R}^d$ and let γ be a weight function such that $\gamma(t) = O(t^{1-1/d-\varepsilon})$, for constants $d \geq 2$, $\alpha > 0$, and $\varepsilon > 0$. Then the intersection graph $G[F]$ has a $(6^d/(6^d + 1))$-balanced separator and a clique partition $\mathcal{C}(F_{\text{sep}})$ of F_{sep} with weight $O(n^{1-1/d})$. Such a separator and a clique partition $\mathcal{C}(F_{\text{sep}})$ can be computed in $O(n^{d+2})$ time if the objects have constant complexity.*

A direct application of our separator theorem is a $2^{O(n^{1-1/d})}$ algorithm for Independent Set. For general fat objects, only the 2-dimensional case was known to have such an algorithm [90].

Our separator theorem can be seen as a generalization of the work of Fu [53] who considers a weighting scheme similar to ours. However, Fu's result is significantly less general as it only applies to unit balls and his proof is arguably more complicated. Our result can also be seen as a generalization of the separator theorem of Har-Peled and Quanrud [62] which gives a small separator for constant ply—indeed, our proof borrows some ideas from theirs.

Finally, the technique employed by Fomin et al. [51] in two dimensions has also similar qualities; in particular, the idea of using cliques as a basis for a separator can also be found there, and leads to subexponential parameterized algorithms, even for some problems that we do not tackle here.

After proving the weighted separator theorem for arbitrarily-sized fat objects, we switch to similarly-sized objects. Here the idea is as follows: We find a suitable clique-decomposition $\mathcal{P}$ of the intersection graph $G[F]$, contract each clique to a single vertex, and then work with the contracted graph $G_{\mathcal{P}}$ where the node corresponding to a clique C gets weight $\gamma(|C|)$. We then prove that the graph $G_{\mathcal{P}}$ has constant degree and, using our separator theorem, we prove that $G_{\mathcal{P}}$ has weighted treewidth $O(n^{1-1/d})$. Moreover, we can compute a tree decomposition of this weight in $2^{O(n^{1-1/d})}$ time.

These weighted tree decompositions can often be used (with a small adaptation) in "traditional" dynamic programming algorithms on tree decompositions. Thus we obtain a framework that gives $2^{O(n^{1-1/d})}$-time algorithms for intersection graphs of similarly-sized fat objects for many problems for which treewidth-based algorithms are known. Our framework recovers and often slightly improves the best known results for several problems[1], including Independent Set, Hamiltonian Cycle and Feedback Vertex Set. Our framework also gives the first subexponential algorithms in geometric intersection graphs for, among other problems, r-Dominating Set for constant r, Steiner Tree and Connected Dominating Set.

Furthermore, we show that our approach can be combined with the *rank-based approach* [15], a technique to speed up algorithms for connectivity problems.

A desirable property of algorithms for geometric graphs is that they are *robust*, meaning that they can work directly on the graph without knowledge of the underlying geometry. Most of the known algorithms are in fact non-robust, which could be a problem in applications, since finding a geometric representation of a given geometric intersection graph is NP-hard [31] (and many recognition problems for geometric graphs are ER-complete [71]). One of the advantages of our framework is that it yields robust algorithms for many problems. To this end we need to generalize our scheme slightly: We no longer work with a clique partition to define the contracted graph $G_{\mathcal{P}}$, but with a partition whose classes are the union of constantly many cliques. We show that such a partition can be found efficiently without knowing the set F defining the given intersection graph. Thus we obtain robust algorithms for many of the problems mentioned above, in contrast to known results which almost all need the underlying set F as input.

6.2. Separators for Arbitrarily-Sized Fat Objects

Let F be a set of n α-fat objects in $\mathbb{R}^d$ for some constant $\alpha > 0$, and let $G[F] = (F, E)$ be the intersection graph induced by F. For a given decomposition $\mathcal{C}(F_{\text{sep}})$ of F_{sep} into cliques and a given weight function γ we define the *weight* of F_{sep}, denoted by weight(F_{sep}), as weight$(F_{\text{sep}}) := \sum_{C \in \mathcal{C}(F_{\text{sep}})} \gamma(|C|)$. Next we prove that $G[F]$ admits a balanced separator of weight $O(n^{1-1/d})$ for any cost function $\gamma(t) = O(t^{1-1/d-\varepsilon})$ with $\varepsilon > 0$. Our approach borrows ideas from Har-Peled and Quanrud [62], who show the existence of small separators for low-density sets of objects, although our arguments are significantly more involved.

Step 1: Finding candidate separators. Let H_0 be a minimum-size hypercube containing at least $n/(6^d + 1)$ objects from F, and assume without loss of generality that H_0 is the unit hypercube centered at the origin. Let $H_1, \dots, H_m$ be a collection of $m := n^{1/d}$ hypercubes, all centered at the origin, where H_i has edge length $1 + \frac{2i}{m}$. Note that the largest hypercube, H_m, has edge length 3, and that the distance between consecutive hypercubes H_i and H_{i+1} is $1/n^{1/d}$.

Each hypercube H_i induces a partition of F into three subsets: a subset $F_{\text{in}}(H_i)$ containing all objects that lie completely in the interior of H_i, a subset $F_\partial(H_i)$ containing

[1]Note that most of the earlier results are in the parameterized setting, but we do not consider parameterized algorithms here.

all objects that intersect the boundary ∂H_i of H_i, and a subset $F_{\text{out}}(H_i)$ containing all objects that lie completely in the exterior of H_i. Obviously an object from $F_{\text{in}}(H_i)$ cannot intersect an object from $F_{\text{out}}(H_i)$, and so $F_\partial(H_i)$ defines a separator in a natural way. It will be convenient to add some more objects to these separators, as follows. We call an object *large* when its diameter is at least 1/4, and *small* otherwise. We will add all large objects that intersect H_m to our separators. Thus our candidate separators are the sets $F_{\text{sep}}(H_i) := F_\partial(H_i) \cup F_{\text{large}}$, where F_{large} is the set of all large objects intersecting H_m. We show that our candidate separators are balanced:

Lemma 6.2. *For any $0 \leq i \leq m$ we have*

$$\max\left(|F_{\text{in}}(H_i) \setminus F_{\text{large}}|, |F_{\text{out}}(H_i) \setminus F_{\text{large}}|\right) < \frac{6^d}{6^d+1}n.$$

Proof. Consider a hypercube H_i. Because H_0 contains at least $n/(6^d+1)$ objects from F, we immediately obtain

$$\left|F \cap (F_{\text{out}}(H_i) \setminus F_{\text{large}})\right| \leq |F \cap F_{\text{out}}(H_0)| \leq |F \setminus F_{\text{in}}(H_0)| < \left(1 - \frac{1}{6^d+1}\right)n = \frac{6^d}{6^d+1}n.$$

To bound $\left|F_{\text{in}}(H_i) \setminus F_{\text{large}}\right|$, consider a subdivision of H_i into 6^d sub-hypercubes of edge length $\frac{1}{6}(1+\frac{2i}{m}) \leq 1/2$. We claim that any sub-hypercube H_{sub} intersects fewer than $n/(6^d+1)$ small objects from F. To see this, recall that small objects have diameter less than 1/4. Hence, all small objects intersecting H_{sub} are fully contained in a hypercube of edge length less than 1. Since H_0 is a smallest hypercube containing at least $n/(6^d+1)$ objects from F, H_{sub} must thus contain fewer than $n/(6^d+1)$ objects from F, as claimed. Each object in $F_{\text{in}}(H_i)$ intersects at least one of the 6^d sub-hypercubes, so we can conclude that $\left|F_{\text{in}}(H_i) \setminus F_{\text{large}}\right| < \left(6^d/(6^d+1)\right)n$. $\square$

Step 2: Defining the cliques and finding a low-weight separator. Define $F^* := F \setminus (F_{\text{in}}(H_0) \cup F_{\text{out}}(H_m) \cup F_{\text{large}})$. Note that $F_\partial(H_i) \subseteq F^*$ for all i. We partition F^* into *size classes* F^*_s, based on the diameter of the objects. More precisely, for integers s with $1 \leq s \leq s_{\max}$, where $s_{\max} := \lceil (1-1/d)\log n \rceil - 2$, we define

$$F^*_s := \left\{ o \in F^* : \frac{2^{s-1}}{n^{1/d}} \leq \text{diam}(o) < \frac{2^s}{n^{1/d}} \right\}.$$

We furthermore define F^*_0 to be the subset of objects $o \in F^*$ with $\text{diam}(o) < 1/n^{1/d}$. Note that $2^{s_{\max}}/n^{1/d} \geq 1/4$, which means that every object in F^* is in exactly one size class.

Each size class can be decomposed into cliques, as follows. Fix a size class F^*_s, with $1 \leq s \leq s_{\max}$. Since the objects in F are α-fat for a fixed constant $\alpha > 0$, each $o \in F^*_s$ contains a ball of radius $\alpha \cdot (\text{diam}(o)/2) = \Omega(\frac{2^s}{n^{1/d}})$. Moreover, each object $o \in F^*_s$ lies fully or partially inside the outer hypercube H_m, which has edge length 3. This implies we can stab all objects in F^*_s using a set P_s of $O((\frac{n^{1/d}}{2^s})^d)$ points. Thus there exists a decomposition $\mathcal{C}(F^*_s)$ of F^*_s consisting of $O(\frac{n}{2^{sd}})$ cliques. In a similar way we can argue that there exists a decomposition $\mathcal{C}(F_{\text{large}})$ of F_{large} into $O(1)$ cliques. For F^*_0 the argument does not work since objects in F^*_0 can be arbitrarily small. Hence, we create a singleton clique for each object in F_0^*. Together with the decompositions of the size classes F_s^* and of F_{large} we thus obtain a decomposition $\mathcal{C}(F^*)$ of F^* into cliques.

A decomposition of $F_{\text{sep}}(H_i)$ into cliques is induced by $\mathcal{C}(F^*)$, which we denote by $\mathcal{C}(F_{\text{sep}}(H_i))$. Thus, for a given weight function γ, the weight of $F_{\text{sep}}(H_i)$ is equal to $\sum_{C\in\mathcal{C}(F_{\text{sep}}(H_i))}\gamma(|C|)$. Our goal is now to show that at least one of the separators $F_{\text{sep}}(H_i)$ has weight $O(n^{1-1/d})$, when $\gamma(t) = O(t^{1-1/d-\varepsilon})$ for some $\varepsilon > 0$. To this end we will bound the total weight of all separators $F_{\text{sep}}(H_i)$ by $O(n)$. Using that the number of separators is $n^{1/d}$ we then obtain the desired result.

Lemma 6.3. *If* $\gamma(t) = O(t^{1-1/d-\varepsilon})$ *for some* $\varepsilon > 0$ *then* $\sum_{i=1}^{m}\text{weight}(F_{\text{sep}}(H_i)) = O(n)$.

Proof. First consider the cliques in $\mathcal{C}(F_0^*)$, which are singletons. Since objects in F_0^* have diameter less than $1/n^{1/d}$, which is the distance between consecutive hypercube H_i and H_{i+1}, each such object is in at most one set $F_\partial(H_i)$. Hence, its contribution to the total weight $\sum_{i=1}^{m}\text{weight}(F_{\text{sep}}(H_i))$ is $\gamma(1) = O(1)$. Together, the cliques in $\mathcal{C}(F_0^*)$ thus contribute $O(n)$ to the total weight.

Next, consider $\mathcal{C}(F_{\text{large}})$. It consists of $O(1)$ cliques. In the worst case each clique appears in all sets $F_\partial(H_i)$. Hence, their total contribution to $\sum_{i=1}^{m}\text{weight}(F_{\text{sep}}(H_i))$ is bounded by $O(1)\cdot\gamma(n)\cdot n^{1/d} = O(n)$.

Now consider a set $\mathcal{C}(F_s^*)$ with $1 \le s \le s_{\max}$. A clique $C \in \mathcal{C}(F_s^*)$ consists of objects of diameter at most $2^s/n^{1/d}$ that are stabbed by a common point. Since the distance between consecutive hypercubes H_i and H_{i+1} is $1/n^{1/d}$, this implies that C contributes to the weight of $O(2^s)$ separators $F_{\text{sep}}(H_i)$. The contribution to the weight of a single separator is at most $\gamma(|C|)$. (It can be less than $\gamma(|C|)$ because not all objects in C need to intersect ∂H_i.) Hence, the total weight contributed by all cliques, which equals the total weight of all separators, is

$$\sum_{s=1}^{s_{\max}}\sum_{C\in\mathcal{C}(F_s^*)}(\text{weight contributed by } C) \le \sum_{s=1}^{s_{\max}}\sum_{C\in\mathcal{C}(F_s^*)}2^s\gamma(|C|) = \sum_{s=1}^{s_{\max}}\left(2^s\sum_{C\in\mathcal{C}(F_s^*)}\gamma(|C|)\right).$$

Next we wish to bound $\sum_{C\in\mathcal{C}(F_s^*)}\gamma(|C|)$. Define $n_s := |F_s^*|$ and observe that $\sum_{s=1}^{s_{\max}}n_s \le n$. Recall that $\mathcal{C}(F_s^*)$ consists of $O(n/2^{sd})$ cliques, that is, of at most $cn/2^{sd}$ cliques for some constant c. To make the formulas below more readable we assume $c = 1$ (so we can omit c), but it is easily checked that this does not influence the final result asymptotically. Similarly, we will be using $\gamma(t) = t^{1-1/d-\varepsilon}$ instead of $\gamma(t) = O(t^{1-1/d-\varepsilon})$. Because γ is positive and concave, the sum $\sum_{C\in\mathcal{C}(F_s^*)}\gamma(|C|)$ is maximized when the number of cliques is maximal, namely $\min(n_s, n/2^{sd})$, and when the objects are distributed as evenly as possible over the cliques. Hence,

$$\sum_{C\in\mathcal{C}(F_s^*)}\gamma(|C|) \le \begin{cases} n_s & \text{if } n_s \le n/2^{sd},\\ (n/2^{sd})\cdot\gamma\left(\frac{n_s}{n/2^{sd}}\right) & \text{otherwise.}\end{cases}$$

We now split the set $\{1,\ldots,s_{\max}\}$ into two index sets S_1 and S_2, where S_i contains all indices s such that $n_s \le n/2^{sd}$, and S_2 contains all remaining indices. Thus

$$\sum_{s=1}^{s_{\max}}\left(2^s\sum_{C\in\mathcal{C}(F_s^*)}\gamma(|C|)\right) = \sum_{s\in S_1}\left(2^s\sum_{C\in\mathcal{C}(F_s^*)}\gamma(|C|)\right) + \sum_{s\in S_2}\left(2^s\sum_{C\in\mathcal{C}(F_s^*)}\gamma(|C|)\right)$$

The first term in the previous equation can be bounded by

$$\sum_{s \in S_1} \left(2^s \sum_{C \in \mathcal{C}(F_s^*)} \gamma(|C|) \right) \leq \sum_{s \in S_1} 2^s n_s \leq \sum_{s \in S_1} 2^s (n/2^{sd}) = n \sum_{s \in S_1} 1/2^{s(d-1)} = O(n),$$

where the last step uses that $d \geq 2$. For the second term we get

$$\begin{aligned}
\sum_{s \in S_2} \left(2^s \sum_{C \in \mathcal{C}(F_s^*)} \gamma(|C|) \right) &\leq \sum_{s \in S_2} \left(2^s (n/2^{sd}) \cdot \gamma\left(\frac{n_s}{n/2^{sd}} \right) \right) \\
&\leq \sum_{s \in S_2} \left(\frac{n}{2^{s(d-1)}} \cdot \left(\frac{n_s 2^{sd}}{n} \right)^{1-1/d-\varepsilon} \right) \\
&\leq n \sum_{s \in S_2} \left(\frac{n_s}{n} \right)^{1-1/d-\varepsilon} \frac{1}{2^{sd\varepsilon}} \\
&\leq n \sum_{s \in S_2} \left(\frac{1}{2^{d\varepsilon}} \right)^s \\
&= O(n).
\end{aligned}$$
□

We are now ready to prove Theorem 6.1.

Proof of Theorem 6.1. Each candidate separator $F_{\text{sep}}(H_i)$ is $(6^d/(6^d+1))$-balanced by Lemma 6.2. Their total weight is $O(n)$ by Lemma 6.3, and since we have $n^{1/d}$ candidates one of them must have weight $O(n^{1-1/d})$. This separator can be found in $O(n^{d+2})$ time by brute force. Indeed, to find the hypercube $H_0 = [x_1, x_1'] \times \cdots \times [x_d, x_d']$ in $O(n^{d+2})$ time we first guess the object defining x_i, for all $1 \leq i \leq d$, then guess the object defining x_1' (and, hence, the size of the hypercube), and finally determine the number of objects inside the hypercube. Once we have H_0, we can generate the hypercubes $H_1, \cdots, H_{n^{1/d}}$, generate the cliques as described above, and then compute the weights of the separators $F_{\text{sep}}(H_i)$ by brute force within the same time bound. □

Corollary 6.4. *Let F be a set of n fat objects in $\mathbb{R}^d$, where $d \geq 2$ is a constant. Then* Independent Set *on the intersection graph $G[F]$ can be solved in $2^{O(n^{1-1/d})}$ time.*

Proof. Let $\gamma(t) := \log(t+1)$, and compute a separator F_{sep} for $G[F]$ using Theorem 6.1. For each subset $S_{\text{sep}} \subseteq F_{\text{sep}}$ of independent (that is, pairwise non-adjacent) vertices we find the largest independent set S of G such that $S \supseteq S_{\text{sep}}$, by removing the closed neighbourhood of S_{sep} from G and recursing on the remaining connected components. Finally, we report the largest of all these independent sets. Because a clique $C \in \mathcal{C}(F_{\text{sep}})$ can contribute at most one vertex to S_{sep}, we have that the number of candidate sets S_{sep} is at most

$$\prod_{C \in \mathcal{C}(F_{\text{sep}})} (|C|+1) = 2^{\sum_{C \in \mathcal{C}(F_{\text{sep}})} \log(|C|+1)} = 2^{O(n^{1-1/d})}.$$

Since all components on which we recurse have at most $(6^d/(6^d+1))n$ vertices, the running time $T(n)$ satisfies

$$T(n) = 2^{O(n^{1-1/d})} T((6^d/(6^d+1))n) + \text{poly}(n),$$

which solves to $T(n) = 2^{O(n^{1-1/d})}$. □

6.3. An Algorithmic Framework for Similarly-Sized Fat Objects

We restrict our attention to *similarly-sized* fat objects. More precisely, we consider intersection graphs of sets F of objects such that, for each $o \in F$, there are balls B_{in} and B_{out} in $\mathbb{R}^d$ such that $B_{\text{in}} \subseteq F \subseteq B_{\text{out}}$, and $\text{radius}(B_{\text{in}}) = \alpha$ and $\text{radius}(B_{\text{out}}) = 1$ for some fatness constant $\alpha > 0$. The restriction to similarly-sized objects makes it possible to construct a clique cover of F with the following property: if we consider the intersection graph $G[F]$ where the cliques are contracted to single vertices, then the contracted graph has constant degree. Moreover, the contracted graph admits a tree decomposition whose weighted treewidth is $O(n^{1-1/d})$. This tool allows us to solve many problems on intersection graphs of similarly-sized fat objects.

Our tree-decomposition construction uses the separator theorem from the previous subsection. That theorem also states that we can compute the separator for $G[F]$ in polynomial time, provided we are given F. However, finding the separator if we are only given the graph and not the underlying set F is not easy. Note that deciding whether a graph is a unit-disk graph is already ER-complete [71]. Nevertheless, we show that for similarly-sized fat objects we can find certain tree decompositions with the desired properties, purely based on the graph $G[F]$.

κ-partitions, $\mathcal{P}$-contractions, and separators. Let $G = (V, E)$ be the intersection graph of an (unknown) set F of similarly-sized fat objects, as defined above. The separators in the previous section use cliques as basic components. We need to generalize this slightly, by allowing connected unions of a constant number of cliques as basic components. Thus we define a *κ-partition* of G as a partition $\mathcal{P} = (V_1, \dots, V_k)$ of V such that every partition class V_i induces a connected subgraph that is the union of at most κ cliques. Note that a 1-partition corresponds to a clique cover of G.

Given a κ-partition $\mathcal{P}$ of G we define the *$\mathcal{P}$-contraction* of G, denoted by $G_{\mathcal{P}}$, to be the graph obtained by contracting all partition classes V_i to single vertices and removing loops and parallel edges. In many applications it is essential that the $\mathcal{P}$-contraction we work with has maximum degree bounded by a constant. From now on, when we speak of the degree of a κ-partition $\mathcal{P}$ we refer to the degree of the corresponding $\mathcal{P}$-contraction.

The following theorem and its proof are very similar to Theorem 6.1, but it applies only for similarly-sized objects because of the degree bound on $G_{\mathcal{P}}$. The other main difference is that the separator is defined on the $\mathcal{P}$-contraction of a given κ-partition, instead of on the intersection graph G itself.

Theorem 6.5. *Let $G = (V, E)$ be the intersection graph of a set of n similarly-sized fat objects in $\mathbb{R}^d$, and let γ be a weight function such that $\gamma(t) = O(t^{1-1/d-\varepsilon})$, for constants $d \geq 2$ and $\varepsilon > 0$. Suppose we are given a κ-partition $\mathcal{P}$ of G such that $G_{\mathcal{P}}$ has maximum degree at most Δ, where κ and Δ are constants. Then there exists a $(6^d/(6^d+1))$-balanced separator for $G_{\mathcal{P}}$ of weight $O(n^{1-1/d})$.*

The following lemma shows that a partition $\mathcal{P}$ as needed in Theorem 6.5 can be computed even in the absence of geometric information.

Lemma 6.6. *Let $G = (V, E)$ be the intersection graph of an (unknown) set of n similarly-sized fat objects in $\mathbb{R}^d$ for some constant $d \geq 2$. There there exist constants κ and Δ such that a κ-partition $\mathcal{P}$ for which $G_{\mathcal{P}}$ has maximum degree Δ can be computed in polynomial time.*

Proof. Let $S \subseteq V$ be a maximal independent set in G (e.g., it is inclusion-wise maximal). We assign each vertex $v \in V \setminus S$ to an arbitrary vertex $s \in S$ that is a neighbour of v; such a vertex s always exists since S is maximal. For each vertex $s \in S$ define $V_s := \{s\} \cup \{v \in V \setminus S : v \text{ is assigned to } s\}$. We prove that the partition $\mathcal{P} := \{V_s : s \in S\}$, which can be computed in polynomial time, has the desired properties.

Let o_v denote the (unknown) object corresponding to a vertex $v \in V$, and for a partition class V_s define $U(V_s) := \bigcup_{v \in V_s} o_v$. We call $U(V_s)$ a *union-object.* Let $\mathcal{U}_S := \{U(V_s) : s \in S\}$. Because the objects defining G are similarly-sized and fat, there are balls $B_{\text{in}}(o_v)$ of radius $\alpha = \Omega(1)$ and $B_{\text{out}}(o_v)$ of radius 1 such that $B_{\text{in}}(o_v) \subseteq o_v \subseteq B_{\text{out}}(o_v)$.

Now observe that each union-object $U(V_s)$ is contained in a ball of radius 3. Hence, we can stab all balls $B_{\text{in}}(o_v)$, $v \in V_s$ using $O(1)$ points, which implies that $\mathcal{P}$ is a κ-partition for some $\kappa = O(1)$.

To prove that the maximum degree of $G_{\mathcal{P}}$ is $O(1)$, we note that any two balls $B_{\text{in}}(s)$, $B_{\text{in}}(s')$ with $s, s' \in S$ are disjoint (because S is an independent set in G). Since all union-objects $U(s')$ that intersect $U(s)$ are contained in a ball of radius 9, an easy packing argument now shows that $U(s)$ intersects $O(1)$ union-objects $U(s)$. Hence, the node in $G_{\mathcal{P}}$ corresponding to V_s has degree $O(1)$. □

Weighted tree decompositions for $\mathcal{P}$-contractions. We now introduce the notion of *weighted treewidth* [108]. Recall that the *width* of a tree decomposition is the size of its largest bag minus 1, and the *treewidth* of a graph G equals the minimum width of a tree decomposition of G. In weighted treewidth, each vertex has a weight, and the *weighted width* of a tree decomposition is the maximum over the bags of the sum of the weights of the vertices in the bag (note: without the -1). The *weighted treewidth* of a graph is the minimum weighted width over its tree decompositions.

Now let $\mathcal{P} = (V_1, \ldots, V_k)$ be a κ-partition of a given graph G which is the intersection graph of similarly-sized fat objects, and let γ be a given weight function on partition classes. We apply the concept of weighted treewidth to $G_{\mathcal{P}}$, where we assign each vertex V_i of $G_{\mathcal{P}}$ a weight $\gamma(|V_i|)$. Because we have a separator for $G_{\mathcal{P}}$ of low weight by Theorem 6.5, we can prove a bound on the weighted treewidth of $G_{\mathcal{P}}$ using standard techniques.

Lemma 6.7. *Let $\mathcal{P}$ be a κ-partition of a family of similarly-sized fat objects such that $G_{\mathcal{P}}$ has maximum degree at most Δ, where κ and Δ are constants. Then the weighted treewidth of $G_{\mathcal{P}}$ is $O(n^{1-1/d})$ for any weight function γ with $\gamma(t) = O(t^{1-1/d-\varepsilon})$.*

Proof. The lemma follows from Theorem 6.5 by a minor variation on standard techniques — see for example [14, Theorem 20]. Take a separator S of $G_{\mathcal{P}}$ as indicated by

Theorem 6.5. Recursively, make tree decompositions of the connected components of $G_{\mathcal{P}} \setminus S$. Take the disjoint union of these tree decompositions, add an edge between the two trees and then add S to all bags. We now have a tree decomposition of $G_{\mathcal{P}}$. As base case, when we have a subgraph of $G_{\mathcal{P}}$ with $O(n^{1-1/d})$ vertices, then we take one bag with all vertices in this subgraph.

The weight of bags for subgraphs of $G_{\mathcal{P}}$ with r vertices fulfils $w(r) = O(r^{1-1/d}) + w(6^d/(6^d+1)r)$, which gives that the weighted width of this tree decomposition is $w(n) = O(n^{1-1/d})$. □

By combining Lemmas 6.6 and 6.7 we can obtain a κ-partition such that $G_{\mathcal{P}}$ has constant degree, and such that the weighted treewidth of $G_{\mathcal{P}}$ is as desired. In what follows, we work towards finding a suitable weighted tree decomposition.

A *blowup* of a vertex v by an integer t results in a graph where we replace the vertex v with a clique of size t (called the clique of v), in which we connect every vertex to the neighbourhood of v. The vertices in these cliques all have weight 1.

Lemma 6.8 ([24]). *Let $W \subseteq V$ form a clique in $G = (V, E)$. Every tree decomposition $(T_G, \{X^G_t \mid t \in T_G\})$ of G has a bag $X^G_t \in \{X^G_t \mid t \in T_G\}$ with $W \subseteq X^G_t$.*

Lemma 6.9. *The weighted treewidth of a graph G with weight function $w : V(G) \to \mathbb{N}$ is equal to 1 plus the treewidth of H that is gained from G by blowing up each vertex v by $\gamma(v)$. Let $(T_H, \{X^H_t \mid t \in T_H\})$ be a tree decomposition of H. Then we can create a tree decomposition $(T_G, \{X^G_t \mid t \in T_G\})$ of G where T_G is isomorphic to T_H the following way: a vertex $v \in G$ is added to a bag if and only if the corresponding bag in T_H contains all vertices from the clique of v. Furthermore, the treewidth of $(T_H, \{X^H_t \mid t \in T_H\})$ is at most the weighted width of $(T_G, \{X^G_t \mid t\, inT_G\})$ minus 1.*

Proof. The proof we give below is a simple modification of folklore insights on treewidth; For related results see [27, 19]. First, we notice that $(T_G, \{X^G_t \mid t \in T_G\})$ is a tree decomposition of G. From Lemma 6.8, we have that for each vertex v and edge $\{v, w\}$ there is a bag in $(T_G, \{X^G_t \mid t \in T_G\})$ that contains v, respectively $\{v, w\}$. For the third condition of tree decompositions, suppose j_2 is in T_G on the path from j_1 to j_3. If v belongs to the bags of j_1 and j_3, then all vertices in the clique resulting of blowing up v belong in $(T_H, \{X^H_t \mid t \in T_H\})$ to the bags of j_1 and j_3, hence by the properties of tree decompositions to the bag of j_2, and hence $v \in X^G_(j_2)$. It follows that the preimage of each vertex in V_G is a subtree of T_G. The total weight of vertices in a bag in $(T_G, \{X^G_t \mid t \in T_G\})$ is never larger than the size of the corresponding bag in $(T_H, \{X^H_t \mid t \in T_H\})$. Thus, by taking for $(T_G, \{X^G_t \mid t \in T_G\})$ a tree decomposition with minimum weighted treewidth, we see that the weighted treewidth of G is at most the treewidth of H plus 1; the additive term of 1 comes from the -1 in the definition of treewidth.

In the other direction, if we take a tree decomposition $(T_G, \{X^G_t \mid t \in T_G\})$ of G, we can obtain one of H by replacing in each bag each vertex v by the clique that results from blowing up G. The size of a bag in the tree decomposition of H now equals the total weight of the vertices in G; hence the width of $(T_G, \{X^G_t \mid t \in T_G\})$ equals the weighted width of the obtained tree decomposition of H; it follows that the weighted treewidth of G is at least the treewidth of H minus 1. □

We are now ready to prove our main theorem for algorithms.

Theorem 6.10. *Let $G = (V, E)$ be the intersection graph of an (unknown) set of n similarly-sized α-fat objects in $\mathbb{R}^d$, and let γ be a weight function such that $1 \le \gamma(t) = O(t^{1-1/d-\varepsilon})$, for constants $d \ge 2$, $\alpha > 0$, and $\varepsilon > 0$. Then there exist constants κ and Δ such that there is a κ-partition $\mathcal{P}$ with the following properties: (i) $G_{\mathcal{P}}$ has maximum degree at most Δ, and (ii) $G_{\mathcal{P}}$ has weighted treewidth $O(n^{1-1/d})$. Moreover, such a partition $\mathcal{P}$ and a corresponding tree decomposition of weight $O(n^{1-1/d})$ can be computed in $2^{O(n^{1-1/d})}$ time.*

Proof. Lemma 6.6 provides a partition $\mathcal{P}$ built around a maximal independent set. By Lemma 6.7, the weighted treewidth of $G_{\mathcal{P}}$ is $O(n^{1-1/d})$.

To get a tree decomposition, consider the above partition again, with a weight function $\gamma(t) = O(t^{1-1/d-\varepsilon})$. We work on the contracted graph $G_{\mathcal{P}}$; we intend to simulate the weight function by modifying $G_{\mathcal{P}}$. Let H be the graph we get from $G_{\mathcal{P}}$ by blowing up each vertex v_C by an integer that is approximately the weight of the corresponding class, more precisely, we blow up v_C by $\lceil \gamma(|C|) \rceil$. By Lemma 6.9, its treewidth (plus one) is a 2-approximation of the weighted treewidth of G (since $\gamma(t) \ge 1$). Therefore, we can run a treewidth approximation algorithm that is single exponential in the treewidth of H. We can use the algorithm from either [100] or [18] for this, both have running time $2^{O(tw(H))}|V(H)|^{O(1)} = 2^{O(n^{1-1/d})}(n\,\gamma(n))^{O(1)} = 2^{O(n^{1-1/d})}$, and provide a tree decomposition whose width is a c-approximation of the treewidth of H, from which we gain a tree decomposition whose weighted treewidth is a $2c$-approximation of the weighted treewidth of $G_{\mathcal{P}}$. □

6.4. Basic Algorithmic Applications

In this section, we give examples of how κ-partitions and weighted tree decompositions can be used to obtain subexponential-time algorithms for classical problems on geometric intersection graphs.

Given a κ-partition $\mathcal{P}$ and a weighted tree decomposition of $G_{\mathcal{P}}$ of width τ, we note that there exists a nice tree decomposition of G (i.e., a "traditional", non-partitioned tree decomposition) with the property that each bag is a subset of the union of a number of partition classes, such that the total weight of those classes is at most τ. This can be seen by creating a nice version of the weighted tree decomposition of $G_{\mathcal{P}}$, and then replacing every introduce/forget bag (that introduces/forgets a class of the partition) by a series of introduce/forget bags (that introduce/forget the individual vertices). We call such a decomposition a *traditional tree decomposition*. Using such a decomposition, it becomes easy to give algorithms for problems for which we already have dynamic-programming algorithms operating on nice tree decompositions. We can re-use the algorithms for the leaf, introduce, join and forget cases, and either show that the number of partial solutions remains bounded (by exploiting the properties of the underlying κ-partition) or show that we can discard some irrelevant partial solutions.

We present several applications for our framework, resulting in $2^{O(n^{1-1/d})}$-time algorithms for various problems. In addition to the (non-robust) Independent Set algorithm for fat objects based on our separator, we also give a robust algorithm for similarly sized fat objects. This adds robustness compared to the state of the art [91]. In the rest of the applications, our algorithms work on intersection graphs of d-dimensional similarly sized fat objects; this is usually a larger graph class than what has been studied. We have non-robust algorithms for Hamiltonian Path and Hamiltonian Cycle; this is a simple generalization of the previously known [51, 76] algorithm for unit disks. For Feedback Vertex Set, we give a robust algorithm with the same running time improvement, over a non-robust algorithm that works in 2-dimensional unit disk graphs [51]. For r-Dominating Set, we give a robust algorithm for $d \geq 2$, which is the first subexponential algorithm in dimension $d \geq 3$, and the first robust subexponential algorithm for $d = 2$ [90]. (The algorithm in [90] is for Dominating Set in unit disk graphs.) Finally, we give robust algorithms for Steiner Tree, r-Dominating Set, Connected Vertex Cover, Connected Feedback Vertex Set and Connected Dominating Set, which are – to our knowledge – also the first subexponential algorithms in geometric intersection graphs for these problems.

In the following, we fix our weight function to be $\gamma(k) = \log(k+1)$.

Theorem 6.11. *If a κ-partition and a weighted tree decomposition of width at most τ is given,* Independent Set *and* Vertex Cover *can be solved in time $2^{\kappa\tau}n^{O(1)}$.*

Proof. A well-known algorithm (see, e.g., [34]) for solving Independent Set on graphs of bounded treewidth, computes, for each bag t and subset $S \subseteq X_t$, the maximum size $c[t, S]$ of an independent subset $\hat{S} \subset G[t]$ such that $\hat{S} \cap X_t = S$.

An independent set never contains more than one vertex of a clique. Therefore, if we have a traditional tree decomposition $(T, \{X_t \mid t \in T\})$, obtained from a κ-partitioned weighted tree decomposition, X_t is a subset of the union of partition classes $\{V_i \mid i \in \overline{X}_t\}$ (where $\overline{X}_t$ denotes the corresponding bag of the weighted tree decomposition). Since from each partition class we can select at most κ vertices (one vertex from each clique), the number of subsets $\hat{S}$ that need to be considered is at most $\prod_{i\in\overline{X}_t}(|V_i| + 1)^\kappa = \exp\left(\sum_{i\in\overline{X}_t} \kappa \log\left(|V_i| + 1\right)\right) = 2^{\kappa\tau}$.

Applying the standard algorithm for Independent Set on a traditional tree decomposition, using the fact that only solutions that select at most one vertex from each clique get a non-zero value, we obtain the claimed algorithm. Minimum Vertex Cover can be solved by finding a solution I to Maximum Independent Set and returning the complement $V \setminus I$, which is a minimum vertex cover. □

Corollary 6.12. *For any constant $d \geq 2$,* Independent Set *and* Vertex Cover *can be solved in $2^{O(n^{1-1/d})}$ time on intersection graphs of similarly-sized d-dimensional fat objects, even if the geometric representation is not given.*

In the remainder of this section, because we need additional assumptions that are derived from the properties of intersection graphs, we state our results in terms of algorithms operating directly on intersection graphs. However, note that underlying each of these results is an algorithm operating on a weighted tree decomposition of the contracted graph.

To obtain the algorithm for Independent Set, we exploited the fact that we can select at most one vertex from each clique, and that thus, we can select at most κ vertices from each partition class. For Dominating Set, our bound for the treewidth is however not enough. Instead, we need the following, stronger result, which states that the weight of a bag in the decomposition can still be bounded by $O(n^{1-1/d})$, even if we take the weight to be the total weight of the classes in the bag *and* that of their distance-r neighbours:

Theorem 6.13. *Let G be an intersection graph of n similarly-sized d-dimensional fat objects, and let $r \geq 1$ be a constant. For any weight function γ, there exists a constant $\kappa = O(1)$ such that G has a κ-partition $\mathcal{P}$ and a corresponding $G_{\mathcal{P}}$ of maximum degree at most Δ, where $G_{\mathcal{P}}$ has a weighted tree decomposition with the additional property that for any bag b, the total weight of the partition classes $\{V_i \in \mathcal{P} \mid$ (some vertex in) V_i is within distance r of some $V_j \in \sigma(b)\}$ is $O(n^{1-1/d})$.*

Proof. As per Theorem 6.10, there exist constants $\kappa, \Delta = O(1)$ such that G has a κ-partition in which each class of the partition is adjacent to at most Δ other classes.

We now create a new geometric intersection graph G', which is made by copying each vertex (and its corresponding object) at most κ^r times. We create the following κ^r-partition $\mathcal{P}^r$: for each class V_i of the original partition, create a class that contains a copy of the vertices from V_i and copies of the vertices from the classes within distance at most r from V_i. This graph G^r has at most $\kappa^r n = O(n)$ vertices, and it is an intersection graph of similarly-sized objects; furthermore, the set $\mathcal{P}^r$ has low union ply. Therefore, we can find a weighted tree decomposition of $G^r_{\mathcal{P}^r}$ of width $O(n^{1-1/d})$ by Lemma 6.7.

This decomposition can also be used as a decomposition for the original κ-partition, by replacing each partition class with the corresponding original partition class. □

Theorem 6.14. *Let $r \in \mathbb{Z}_+, d \geq 2$ be constants. Then r-DOMINATING SET can be solved in $2^{O(n^{1-1/d})}$ time on intersection graphs of similarly-sized d-dimensional fat objects.*

Proof. We first present the argument for DOMINATING SET. It is easy to see that from each partition class, we need to select at most $\kappa^2(\Delta+1)$ vertices: each partition class can be partitioned into at most κ cliques, and each of these cliques is adjacent to at most $\kappa(\Delta+1)$ other cliques. If we select at least $\kappa(\Delta+1)+1$ vertices from a clique, we can instead select only one vertex from the clique, and select at least one vertex from each neighbouring clique.

We once again proceed by dynamic programming on a traditional tree decomposition (see e.g. [34] for an algorithm solving DOMINATING SET using tree decompositions). However, rather than needing just two states per vertex (in the solution or not), we need three: a vertex can be either in the solution, not in the solution and not dominated, or not in the solution and dominated. After processing each bag, we discard partial solutions that select more than $\kappa^2(\Delta+1)$ vertices from any class of the partition. Note that all vertices of each partition class are introduced before any are forgotten, so we can guarantee we do indeed never select more than $\kappa^2(\Delta+1)$ vertices from each partition class.

The way vertices outside the solution are dominated or not is completely determined by the vertices that are in the solution and are neighbours of the vertices in the bag. While the partial solution does not track this explicitly for vertices that are forgotten, by using the fact that we need to select at most $\kappa\Delta$ vertices from each class of the partition, and the fact that Theorem 6.13 bounds the total weight of the neighbourhood of the partition classes in a bag, we see that there are at most $\Pi_i(|V_i| + 1)^{\kappa^2(\Delta+1)} = \exp(\kappa^2(\Delta + 1) \sum_i \log(|V_i| + 1)) = 2^{O(n^{1-1/d})}$, where the product (resp., sum) is taken over all partition classes V_i that appear in the current bag or are a neighbours of such a class.

For the generalization where $r > 1$, the argument that we need to select at most $\kappa(\Delta + 1)$ vertices from each clique still holds: moving a vertex from a clique with more than $\kappa(\Delta + 1)$ vertices selected to an adjacent clique only decreases the distance to any vertices it helps cover. The dynamic programming algorithm needs, in a partial solution, to track at what distance from a vertex in the solution each vertex is. This, once again, is completely determined by the solution in partition classes at distance at most r; the number of such cases we can bound using Theorem 6.13. □

6.5. Application of the Rank-Based Approach

To illustrate how our algorithmic framework can be combined with the rank-based approach, we now give an algorithm for Steiner Tree.

We only consider the unweighted variant of Steiner Tree, as, assuming the ETH, the Weighted Steiner Tree problem does not admit a subexponential-time algorithm, even on a clique (as Weighted Steiner Tree on a clique can encode Steiner Tree on an arbitrary graph by setting the weights of non-edges to a sufficiently large number). Therefore, we should not expect Theorem 6.15 to hold for the weighted case.

Theorem 6.15. *Let $d \geq 2$ be a constant. Then* Steiner Tree *can be solved in $2^{O(n^{1-1/d})}$ time on intersection graphs of d-dimensional similarly-sized fat objects.*

Proof. The algorithm works by dynamic programming on a traditional tree decomposition. The leaf, introduce, join and forget cases can be handled as they are in the conventional algorithm for Steiner Tree on tree decompositions, see e.g. [15]. However, after processing each bag, we can reduce the number of partial solutions that need to be considered by exploiting the properties of the underlying κ-partition.

To this end, we first need a bound on the number of vertices that can be selected from each class of the κ-partition $\mathcal{P}$.

Lemma 6.16. *Let C be a clique in a κ-sized clique cover of a partition class $V_i \in \mathcal{P}$. Then any optimal solution X contains at most $\kappa(\Delta + 1)$ vertices from C that are not also in K. Furthermore, any optimal solution thus contains at most $\kappa^2(\Delta + 1)$ vertices (that are not also in K) from each partition class.*

Proof. To every vertex $v \in (C \cap X) \setminus K$ we greedily assign a *private neighbour* $u \in X \setminus C$ such that u is adjacent to v and u is not adjacent to any other previously assigned private neighbour. If this process terminates before all vertices in $(C \cap X) \setminus K$ have been assigned a private neighbour, then the remaining vertices are redundant and can be removed from the solution.

We now note that since the neighbourhood of C can be covered by at most $\kappa(\Delta+1)$ cliques, this gives us an upper bound on the number of private neighbours that can be assigned and thus bounds the number of vertices that can be selected from any partition class. □

The algorithm for Steiner Tree presented in [15] is for the weighted case, but we can ignore the weights by setting them to 1. A partial solution is then represented by a subset $\hat{S} \subseteq X_t$ (representing the intersection of the partial solution with the vertices in the bag), together with an equivalence relation on $\hat{S}$ (which indicates which vertices are in the same connected component of the partial solution).

Since we select at most $\kappa^2(\Delta+1)$ vertices from each partition class, we can discard partial solutions that select more than this number of vertices from any partition class. Then the number of subsets S considered is at most

$$\prod_{i \in \overline{X}_t)} (|V_i|+1)^{\kappa^2(\Delta+1)} = \exp\left(\kappa^2(\Delta+1) \cdot \sum_{i \in \overline{X}_t} \log(|V_i|+1)\right) \leq \exp\left(\kappa^2(\Delta+1)\tau\right).$$

For any such subset $\hat{S}$, the number of possible equivalence relations is $2^{\Theta(|\hat{S}|\log|\hat{S}|)}$. However, the rank-based approach [15] provides an algorithm called "*reduce*" that, given a set of equivalence relations[2] on $\hat{S}$, outputs a representative set of equivalence relations of size at most $2^{|\hat{S}|}$. Thus, by running the reduce algorithm after processing each bag, we can keep the number of equivalence relations considered single exponential.

Since $|\hat{S}|$ is also $O(\kappa^2(\Delta+1)\tau)$ (we select at most $\kappa^2(\Delta+1)$ vertices from each partition class and each bag contains at most τ partition classes), for any subset $\hat{S}$, the rank-based approach guarantees that we need to consider at most $2^{O(\kappa^2(\Delta+1)\tau)}$ representative equivalence classes of $\hat{S}$ (for each set $\hat{S}$). □

Theorem 6.17. *For any constant $d \geq 2$,* Maximum Induced Forest *(and* Feedback Vertex Set*) can be solved in $2^{O(n^{1-1/d})}$ time on intersection graphs of d-dimensional similarly-sized fat objects.*

Proof. We once again proceed by dynamic programming on a traditional tree decomposition corresponding to the weighted tree decomposition of $G_{\mathcal{P}}$ of width τ, where $\mathcal{P}$ is a κ-partition, and the maximum degree of $G_{\mathcal{P}}$ is at most Δ. We describe the algorithm from the viewpoint of Maximum Induced Forest, but Feedback Vertex Set is simply its complement.

Using the rank-based approach with Maximum Induced Forest requires some modifications to the problem, since the rank-based approach is designed to get maximum connectivity, whereas in Maximum Induced Forest, we aim to "minimize" connectivity (i.e., avoid creating cycles). To overcome this issue, the authors of [15] add a special universal vertex v_0 to the graph (increasing the width of the decomposition by 1) and ask (to decide if a Maximum Induced Forest of size k exists in the graph) whether we can delete some of the edges incident to v_0 such that there exists an induced, connected subgraph, including v_0, of size $k+1$ in the modified graph that has exactly k edges. Essentially, the universal vertex allows us to arbitrarily glue together the trees of an induced forest into a single (connected) tree. This thus reformulates the problem such that we now aim to find a connected solution.

[2]What we refer to as "equivalence relation", [15] refers to as "partition".

The main observation that allows us to use our framework, is that from each clique we can select at most 2 vertices (otherwise, the solution would become cyclic), and that thus, we only need to consider partial solutions that select at most 2κ vertices from each partition class. The number of such subsets is at most $2^{O(\kappa\tau)}$. Since we only need to track connectivity among these 2κ vertices (plus the universal vertex), the rank-based approach allows us to keep the number of equivalence relations considered single-exponential in $\kappa\tau$. Thus, we obtain a $2^{O(\kappa\tau)}n^{O(1)}$-time algorithm. □

Additional Problems Our approach gives $2^{O(n^{1-1/d})}$-time algorithms on geometric intersection graphs of d-dimensional similarly-sized fat objects for almost any problem with the property that the solution (or the complement thereof) can only contain a constant (possibly depending on the "degree" of the cliques) number of vertices of any clique. We can also use our approach for variations of the following problems, that require the solution to be connected:

- Connected Vertex Cover and Connected Dominating Set: these problems may be solved similarly to their normal variants (which do not require the solution to be connected), using the rank-based approach to keep the number of equivalence classes considered single exponential. In case of Connected Vertex Cover, the complement is an independent set, therefore the complement may contain at most one vertex from each clique. In case of Connected Dominating Set, it can be shown that each clique can contain at most $O(\kappa^2\Delta)$ vertices from a minimum connected dominating set.
- Connected Feedback Vertex Set: the algorithm for Maximum Induced Forest can be modified to track that the complement of the solution is connected, and this can be done using the same connectivity-tracking equivalence relation that keeps the solution cycle-free.

Theorem 6.18. *For any constant dimension $d \geq 2$,* Connected Vertex Cover, Connected Dominating Set *and* Connected Feedback Vertex Set *can be solved in time $2^{O(n^{1-1/d})}$ on intersection graphs of similarly-sized d-dimensional fat objects.*

Hamiltonian Cycle. Our separator theorems imply that Hamiltonian Cycle/Path can be solved in $2^{O(n^{1-1/d})}$ time on intersection graphs of similarly-sized d-dimensional fat objects. However, in contrast to our other results, this requires that a geometric representation of the graph is given. Given a 1-partition $\mathcal{P}$ where $G_{\mathcal{P}}$ has constant degree, it is possible to show that a cycle/path only needs to use at most two edges between each pair of cliques; see e.g. [69, 76] and that we can obtain an equivalent instance with all but a constant number of vertices removed from each clique. Our separator theorem implies this graph has treewidth $O(n^{1-1/d})$, and Hamiltonian Cycle/Path can then be solved using dynamic programming on a tree decomposition.

Theorem 6.19. *For any constant dimension $d \geq 2$,* Hamiltonian Cycle *and* Hamiltonian Path *can be solved in time $2^{O(n^{1-1/d})}$ on the intersection graph of similarly-sized d-dimensional fat objects which are given as input.*

6.6. Conclusions

In this chapter, we have seen how a modified form of tree decomposition, with each bag partitioned into cliques, can be used to obtain (strongly) subexponential time algorithms for problems in geometric intersection graphs. Even though these graphs can be dense, the fact that the problems considered can be solved easily on cliques can be exploited.

For each of the previously presented algorithms, it is possible to obtain tight, ETH-based lower bounds. While the proofs of these lower bounds are not part of this thesis (and can be found in [36]), for completeness, Table 6.1 summarizes the algorithmic and lower bound results, showing the most inclusive class where the algorithm applies, and the most restrictive class where the lower bound has been shown.

We have thus also seen that $2^{\Theta(n^{1-1/d})}$ is the "right" running time (tight under the ETH) for many problems in d-dimensional geometric intersection graphs, providing an extension of the square root phenomenon to other classes of graphs. In the next and final part of this thesis, we will move away from purely theoretical results, and instead look at practical applications of treewidth.

We finish this chapter with some open problems:

- Is it possible to obtain clique decompositions without geometric information? Alternatively, how hard is it colour the complement of a small diameter geometric intersection graph of fat objects?
- Many of our applications require the low degree property (i.e., the fact that $G_{\mathcal{P}}$ has bounded degree). Is the low degree property really essential for these applications? Would having low average degree be sufficient?
- Is it possible to modify the framework to work without the similar size assumption?

Finally, it would be interesting to explore the potential consequences of this framework for parameterized and approximation algorithms.

Problem	Algorithm class	Robust	Lower bound class
Independent Set	Fat	no	Unit Ball, $d \geq 2$
Independent Set	Sim. sized fat	yes	Unit Ball, $d \geq 2$
r-Dominating Set, $r = \text{const}$	Sim. sized fat	yes	Induced Grid, $d \geq 2$
Steiner Tree	Sim. sized fat	yes	Induced Grid, $d \geq 2$
Feedback Vertex Set	Sim. sized fat	yes	Induced Grid, $d \geq 2$
Conn. Vertex Cover	Sim. sized fat	yes	Unit Ball, $d \geq 2$ **or** Induced Grid, $d \geq 3$
Conn. Dominating Set	Sim. sized fat	yes	Induced Grid, $d \geq 2$
Conn. Feedback Vertex Set	Sim. sized fat	yes	Unit Ball, $d \geq 2$ **or** Induced Grid, $d \geq 3$
Hamiltonian Cycle/Path	Sim. sized fat	no	Induced Grid, $d \geq 2$

Table 6.1. Summary of our results. In each case we list the most inclusive class where our framework leads to algorithms with $2^{O(n^{1-1/d})}$ running time, and the most restrictive class for which we have a matching lower bound. We also list whether the algorithm is robust.

III

Practical Applications of Treewidth

7 Computing Tree Decompositions on the GPU

7.1. Introduction

As seen in the previous parts, treewidth and (structures similar to) tree decompositions can be very useful theoretical tools. The fact that many otherwise hard graph problems are linear time solvable on graphs of bounded treewidth [22] can also be exploited in practical applications — we will see an example of this in the next chapter. For such applications, it is important to have efficient algorithms, that given a graph, determine the treewidth and find tree decompositions with optimal (or near-optimal) width.

The interest in practical algorithms to compute treewidth and tree decompositions is also illustrated by the fact that both the PACE 2016 and PACE 2017 challenges [37] included treewidth as one of the two challenge topics. Remarkably, while most tracks in the PACE 2016 challenge attracted several submissions [38], there were no submissions for the call for GPU-based programs for computing treewidth. Current sequential exact algorithms for treewidth are only practical when the treewidth is small (up to 4, see [65]), or when the graph is small (see [57, 20, 118, 44, 117]). As computing treewidth is NP-hard, an exponential growth of the running time is to be expected; unfortunately, the exact FPT algorithms that are known for treewidth are assumed to be impractical; e.g., the algorithm of [12] has a running time of $2^{O(tw^3)}n$ for an n-vertex graph of treewidth tw. This creates the need for good parallel algorithms, as parallelism can help to significantly speed up the algorithms, and thus deal with larger graph sizes.

We present a parallel algorithm for computing the treewidth of a graph on a GPU. We implement this algorithm in OpenCL, and experimentally evaluate its performance. Our algorithm is based on an $O^*(2^n)$-time algorithm that explores the elimination orderings of the graph using a Held-Karp like dynamic programming approach. We use Bloom filters to detect duplicate solutions.

GPU programming presents unique challenges and constraints, such as constraints on the use of memory and the need to limit branch divergence. We experiment with various optimizations to see if it is possible to work around these issues. We achieve a very large speed up (up to 77×) compared to running the same algorithm on the CPU.

The starting point of our algorithm is a sequential algorithm by Bodlaender et al. [20]. This algorithm exploits a characterization of treewidth in terms of the width of an *elimination ordering*, and gives a dynamic programming algorithm with a structure that is similar to the textbook Held-Karp algorithm for TSP [66].

Prior work on parallel algorithms for treewidth is limited to one paper, by Yuan [117], who implements a branch and bound algorithm for treewidth on a CPU with a (relatively) small number of cores. With the advent of relatively inexpensive consumer GPUs that offer more than an order of magnitude more computational power than their CPU counterparts do, it is very interesting to explore how exact and fixed-parameter algorithms can take advantage of the unique capabilities of GPUs. We take a first step in this direction, by exploring how treewidth can be computed on the GPU.

Our algorithm is based on the elimination ordering characterization of treewidth. Given a graph $G = (V, E)$, we may *eliminate* a vertex $v \in V$ from G by removing v and turning its neighbourhood into a clique, thus obtaining a new graph. One way to compute treewidth is to find an order in which to eliminate all the vertices of G, such that the maximum degree of each vertex (at the time it is eliminated) is minimized. This formulation is used by e.g. [57] to obtain a (worst-case) $O^*(n!)$-time algorithm. However, it is easy to obtain an $O^*(2^n)$-time algorithm by applying Held-Karp style dynamic programming as first observed by Bodlaender et al. [20]: given a set $S \subseteq V$, eliminating the vertices in S from G will always result in the same intermediate graph, regardless of the order in which the vertices are eliminated (and thus, the order in which we eliminate S only affects the degrees encountered during its elimination). This optimization is used in the algorithms of for instance [45] and [117].

We explore the elimination ordering space in a breadth-first manner. This enables efficient parallelization of the algorithm: during each iteration, a wavefront of states (consisting of the sets of vertices S of size k for which there is a feasible elimination order) is expanded to the wavefront of the next level, with each thread of the GPU taking a set S and considering which candidate vertices of the graph can be added to S. Since multiple threads may end up generating the same state, we then use a bloom filter to detect and remove these duplicates.

To reduce the number of states explored, we experiment with using the minor-min-width heuristic [57], for which we also provide a GPU implementation. Whereas normally this heuristic would be computed by operating on a copy of the graph, we instead compute it using only the original graph and a smaller auxiliary data structure, which may be more suitable for the GPU. We also experiment with several techniques unique to GPU programming, such as using shared/local memory (which can best be likened to the cache of a CPU) and rewriting nested loops into a single loop to attempt to improve parallelism.

We provide an experimental evaluation of our techniques, on a platform equipped with an Intel Core i7-6700 CPU (3.40GHz) with 32GB of RAM (4x8GB DDR4), and an NVIDIA GeForce GTX 1060 with 6GB GDDR5 memory (manufactured by Gigabyte, Part Number `GV-N1060WF2OC-6GD`). Our algorithm is implemented in OpenCL (and thus highly portable). We achieve a very large speedup compared to running the same algorithm on the CPU.

7.2. Additional Definitions

Treewidth. Throughout this thesis, we have been using the characterization of treewidth of a graph G in terms of the smallest width of a tree decomposition of G. In this chapter, we use an alternative characterization, in terms of elimination orderings. Our algorithm is based on the $O(2^n nm)$-time algorithm of Bodlaender et al. [20], which also uses this characterization.

Let $G = (V, E)$ be a graph with vertices $v_1, \dots v_n$. An *elimination ordering* is a permutation $\pi : V \to \{1, \dots, n\}$ of the vertices of G. The *treewidth* of G is equal to $\min_\pi \max_v |Q(\{u \in V \mid \pi(u) < \pi(v)\}, v)|$, where $Q(S, v)$ is the set of vertices $\{u \in V \setminus S \mid \text{there is a path } v, p_1, \dots, p_m, u \text{ such that } p_1, \dots, p_m \in S\}$, i.e., $Q(S, v)$ is the subset of vertices of $V \setminus S$ reachable from v by paths whose internal vertices are in S [20].

An alternative view of this definition is that given a graph G, we can *eliminate* a vertex v by removing it from the graph, and turning its neighbourhood into a clique. The treewidth of a graph is at most k, if there exists an elimination order such that all vertices have degree at most k at the time they are eliminated.

GPU Terminology. Parallelism on a GPU is achieved by executing many *threads* in parallel. These threads are grouped into *warps* of 32 threads. The 32 threads that make up a warp do not execute independently: they share the same program counter, and thus must always execute the same “line” of code (thus, if different threads need to execute different branches in the code, this execution is serialized - this phenomenon, called *branch divergence*, should be avoided). The unit that executes a single thread is called a *CUDA core*.

We used a GTX1060 GPU, which is based on the Pascal architecture [97]. The GTX1060 has 1280 CUDA cores, which are distributed over 10 Streaming Multiprocessors (SMs). Each SM thus has 128 CUDA cores, which can execute up to 4 warps of 32 threads simultaneously. However, a larger number of warps may be assigned to an SM, enabling the SM to switch between executing different warps, for instance to hide memory latency.

Each SM has 256KiB[1] of register memory (which is the fastest, but which registers are addressed must be known at compile time, and thus for example dynamically indexing an array stored in register memory is not possible), 96KiB of shared memory (which can be accessed by all threads executing within the thread block) and 48KiB of L1 cache.

Furthermore, we have approximately 6GB of global memory available which can be written to and read from by all threads, but is very slow (though this is partially alleviated by caching and latency hiding). Shared memory can, in the right circumstances, be read and written much faster, but is still significantly slower than register memory. Finally, there is also texture memory (which we do not use) and constant

[1]A *kibibyte* is 2^{10} bytes.

memory (which is a cached section of the global memory) that can be used to store constants that do not change over the kernel's execution (we use constant memory to store the adjacency lists of the graph).

Shared memory resides physically closer to the SM than global memory, and it would thus make sense to call it "local" memory (in contrast to the more remote global memory). Indeed, OpenCL uses this terminology. However, NVIDIA/CUDA confusingly use "local memory" to indicate a portion of the global memory dedicated to a single thread.

7.3. The Algorithm

7.3.1. Computing Treewidth

Our algorithm works with an iterative deepening approach: for increasing values of k, it repeatedly runs an algorithm that tests whether the graph has treewidth at most k. This means that our algorithm is in practice much more efficient than the worst-case $O^*(2^n)$ behaviour shown by [20], since only a small portion of the 2^n possible subsets may be feasible for the target treewidth k. A similar approach (of solving the decision version of the problem for increasing values of k) was also used by Tamaki [104], who refers to it as *positive-instance driven dynamic programming*.

This algorithm lends itself very well to parallelization, since the subsets can be evaluated (mostly) independently in parallel. This comes at the cost of slightly reduced efficiency (in terms of the number of states expanded) compared to a branch and bound approach (e.g. [44, 117, 118]) since the states with treewidth $< k-1$ are expanded more than once. However, even a branch and bound algorithm needs to expand all of the states with treewidth $k-1$ before it can conclude that treewidth k is optimal, so the main advantage of branch and bound is that it can settle on a solution with treewidth k without expanding all such solutions (of width k).

To test whether the graph has treewidth at most k, we consider subsets $S \subseteq V$ of increasing size, such that the vertices of S can be eliminated in some order without eliminating a vertex of degree $> k$. For each k, the algorithm starts with an input list (that initially contains just the empty set) and then forms an output list by for each set S in the input list, attempting to add every vertex $v \notin S$ to S, which is feasible only if the degree of v in the graph that remains after eliminating the vertices in S is not too large. This is tested using a depth first search. Then, the input and output lists are swapped and the process is repeated. If after n iterations the output list is not empty, we can conclude that the graph has treewidth at most k. Otherwise, we proceed to test for treewidth $k+1$. Pseudocode for this algorithm is given in Algorithm 7.1.

We include three optimizations: first, if $C \subseteq V$ induces a clique, there is an elimination order that ends with the vertices in C [20]. We can thus precompute a maximum clique C, and on line 7 of Listing 7.1, skip any vertices in C. Next, if G has treewidth at most k and there are at least $k+1$ vertex-disjoint paths between vertices u and v, we may add the edge uv to G without increasing its treewidth [32]. Thus, we precompute for each pair of vertices u, v the number of vertex-disjoint paths between them, and when testing whether the graph has treewidth at most k we add edges between all vertices which have at least $k+1$ disjoint paths (note that this has

Algorithm 7.1. Algorithm for computing treewidth. Note that lines 7–19 compute the degree of v in the graph that remains after eliminating the vertices in S.

```
1: for k = 0 to n-1 do do
2:     let inp = $\emptyset$
3:     for i = 0 to n-k-2 do
4:         outp = {}
5:         for each set $S$ in inp do
6:             for each vertex $v \notin S$ do
7:                 let stack = $\emptyset$
8:                 let degree = 0
9:                 push $v$ to stack
10:                while stack $\neq \emptyset$ do
11:                    pop vertex $u$ from stack
12:                    for each unvisited neighbour $w$ of $u$ do
13:                        mark $w$ as visited
14:                        if $w \in S$ then
15:                            push $w$ to stack
16:                        else
17:                            let degree = degree+1
18:                if degree $\leq k$ then
19:                    let outp = outp $\cup$ $\{S \cup \{v\}\}$
20:        let inp = outp
21:    if inp $\neq \emptyset$ then
22:        report the treewidth of $G$ is $k$
```

diminishing returns, since in each iteration we can add fewer and fewer edges). Finally, if the graph has treewidth at least k, then the last $k+1$ vertices can be eliminated in any order so we can terminate execution of the algorithm earlier.

We note that our algorithm does not actually compute a tree decomposition or elimination order, but could easily be modified to do so. Currently, the algorithm stores with each (partial) solution one additional integer, which indicates which four vertices were the last to be eliminated. To reconstruct the solution, one could either store a copy of (one in every four of) the output lists on the disk, or repeatedly add the last four vertices to C and rerun the algorithm to obtain the next four vertices (with each iteration taking less time than the previous, since the size of C has increased).

7.3.2. Duplicate Elimination using Bloom Filters

Each set S may be generated in multiple ways by adding different vertices to subsets $S' \subseteq S$; if we do not detect whether a set S is already in the output list when adding it, we risk the algorithm generating $\Omega(n!)$ sets. To detect whether a set S is already in the output, we use a Bloom filter [11]: Bloom filters are a classical data structure in which an array A of m bits can be used to encode the presence of n elements by means of r hash functions. To insert an element S, we compute r independent hash functions $\{H_i | 1 \leq i \leq r\}$ each of which indicates one position in the array, $A[H_i(S)]$, which should be set to 1. If any of these bits was previously zero, then the element was not yet present in the filter, and otherwise, the probability of a false positive is approximately $(1 - e^{-kn/m})^r$.

In our implementation, we compute two 32-bit hashes $h_1(S), h_2(S)$ using Murmur3 [8], which we then combine linearly to obtain hashes $H_i(S) = h_1(S) + i \cdot h_2(S)$ (which is nearly as good as using r independent hash functions [75]).

In our experiments, we have used $\frac{m}{n} \geq 24$ and $r = 17$ to obtain a low (theoretical) false positive probability of around 1 in 100.000. We note that the possibility of false positives results in a Monte Carlo algorithm (the algorithm may inadvertently decide that the treewidth is higher than it really is). Indeed, given that many millions of states are generated during the search we are guaranteed that the Bloom filter will return some false positives, however, this does not immediately lead to incorrect results: it is still quite unlikely that all of the states leading to an optimal solution are pruned, since there are often multiple feasible elimination orders.

The Bloom filter is very suitable for implementation on a GPU, since our target architecture (and indeed, most GPUs) offers a very fast atomic OR operation [98]. We note that addressing a Bloom filter concurrently may also introduce false negatives if multiple threads attempt to insert the same element simultaneously. To avoid this, we use the initial hash value to pick one of 65.536 mutexes to synchronize access (this allows most operations to happen wait-free and only a collision on the initial hash value causes one thread to wait for another).

7.3.3. Minor-Min-Width

Search algorithms for treewidth are often enhanced with various heuristics and pruning rules to speed up the computation. One very popular choice (used by e.g. [57, 117, 118]) is minor-min-width (MMW) [57] (also known as MMD+(min-d)) [23]). MMW is based on the observation that the minimum degree of a vertex is a lower bound on the treewidth, and that contracting edges (i.e. taking minors) does not increase the treewidth. MMW repeatedly selects a minimum degree vertex, and then contracts it with a neighbour of minimum degree, in an attempt to obtain a minor with large minimum degree (if we encounter a minimum degree that exceeds our target treewidth, we know that we can discard the current state). As a slight improvement to this heuristic, the second smallest vertex degree is also a lower bound on the treewidth [23].

Given a subset $S \subseteq G$, we would like to compute the treewidth of the graphs that remains after eliminating S from G. The most straightforward method is to explicitly create a copy of G, eliminate the vertices of S, and then repeatedly perform the contraction as described above. However, storing e.g. an adjacency list representation of these intermediate graphs would exceed the available shared memory and size of the caches. As we would like to avoid transferring large amounts of data to and from global memory, we implemented a method to compute MMW without explicitly storing the intermediate graphs.

Our algorithm tracks the current degrees of the vertices (which, conveniently, we already have computed to determine which vertices can be eliminated). It is thus easy to select a minimum degree vertex v. Since we do not know what vertices it is adjacent to (in the intermediate graph), we must select a minimum degree neighbour by using a depth-first search, similarly to how we compute the vertex degrees in Algorithm 7.1. Once we have found a minimum degree neighbour u, we run a second dept-first search to compute the number of neighbours u has in common with v, allowing us to update the degree of v. To keep track of which vertices have been contracted, we use a disjoint set data structure.

The disjoint set structure and list of vertex degrees together use only two bytes per vertex (for a graph of up to 256 vertices), thus, they fit our memory constraints whereas an adjacency matrix or adjacency list (for dense graphs, noting that the graphs in question can quickly become dense as vertices are eliminated) would readily exceed it.

7.4. Experiments

7.4.1. Instances

We selected a number of instances from the PACE 2016 dataset [37] and libtw [113]. All instances were preprocessed using the preprocessing rules of our PACE submission [111], which split the graph using *safe* separators: we first split the graph into its connected components, then split on articulation points, then on articulation pairs (making the remaining components 3-connected) and finally – if we can establish that this is safe – on articulation triplets (resulting in the 4-connected components of the graph). We then furthermore try to detect and split on (almost) clique separators in the graph. For a more detailed treatment of these preprocessing rules, we refer to [21].

7.4.2. General Benchmark

We first present an experimental evaluation of our algorithm (without using MMW) on a set of benchmark graphs. Table 7.1 shows the number of vertices, computed treewidth, time taken (in seconds) on the GPU and the number of sets S explored. Note that the time does not include the time taken for preprocessing, and that the vertex count is that of the preprocessed graph (and thus, the original graph may have been larger).

The size of the input and output lists were limited by the memory available on our GPU. With the current configuration (limited to graphs of at most 64 vertices - though the code is written to be flexible and can easily be changed to support up to 256 vertices), these lists could hold at most 180 million states (i.e., subsets $S \subseteq V$ that have a feasible partial elimination order) each. If at any iteration this number was exceeded, the excess states were discarded. The algorithm was allowed to continue execution for the current treewidth k, but was terminated when trying the next higher treewidth (since we might have discarded a state that would have led to a solution with treewidth k, the answer would no longer be exact). The states where the capacity of the lists was exceed are marked with `*`, if the algorithm was terminated then the treewidth is stricken through (and represents the candidate value for treewidth at which the algorithm was terminated, and *not* the treewidth of the graph, which is likely higher).

For instance, for graph `1ubq` the capacity of the lists was first exceeded at treewidth 10, and the algorithm was terminated at treewidth 11 (and thus the actual treewidth is at least 10, but likely higher). For graph `myciel5`, the capacity of the lists was first exceeded at treewidth 19, but still (despite discarding some states) a solution of treewidth 19 was nevertheless found (which we thus know is the exact treewidth).

Name	$\|V\|$	tw	Time (sec.)		Exp
			GPU	CPU	
1e0b_graph	55	24	779	-	1730 $\times 10^6$
1fjl_graph*	57	26	1730	-	3680 $\times 10^6$
1igd_graph	59	25	107	5120	261 $\times 10^6$
1ku3_graph	60	22	235	-	542 $\times 10^6$
1ubq*	47	~~11~~	1130	-	2300 $\times 10^6$
8x6_torusGrid*	48	~~7~~	1110	-	2100 $\times 10^6$
BN_97	48	18	1020	-	2310 $\times 10^6$
BN_98	47	21	689	-	1590 $\times 10^6$
contiki_dhcpc_handle_dhcp*	39	6	1490	-	2930 $\times 10^6$
DoubleStarSnark	30	6	34,5	873	87,6 $\times 10^6$
DyckGraph	32	7	280	-	639 $\times 10^6$
HarborthGraph*	40	5	698	-	1540 $\times 10^6$
KneserGraph_8_3*	56	~~24~~	1710	-	4130 $\times 10^6$
McGeeGraph	24	7	1,30	25,3	3,85 $\times 10^6$
myciel4	23	10	0,234	0,460	0,0978 $\times 10^6$
myciel5*	47	19	2000	70.600	4000 $\times 10^6$
NonisotropicUnitaryPolarGraph_3_3	63	53	1,16	60,4	1,56 $\times 10^6$
queen5_5	25	18	0,212	0,0230	0,00313 $\times 10^6$
queen6_6	36	25	0,254	0,389	0,0360 $\times 10^6$
queen7_7	49	35	0,966	43,5	1,90 $\times 10^6$
queen8_8	64	45	26,3	2040	57,9 $\times 10^6$
RandomBarabasiAlbert_100_2*	41	12	1610	-	3280 $\times 10^6$
RandomBoundedToleranceGraph_60	59	30	0,274	0,635	0,0560 $\times 10^6$
SylvesterGraph	36	15	248	-	632 $\times 10^6$
te*	62	~~7~~	1170	-	2160 $\times 10^6$
water	21	9	0,197	0,00600	0,00124 $\times 10^6$

Table 7.1. Performance of the algorithm on several benchmark graphs, using global memory and a work size of 128.

For several graphs (those where the GPU version of the algorithm took at most 5 minutes), we also benchmarked a sequential version of the same algorithm on the CPU. In some cases, the algorithm achieves a very large speedup compared to the CPU version (up to 77×, in the case of `queen8_8`). Additionally, for `myciel5`, we also ran the CPU-based algorithm, which took more than 19 hours to finish. The GPU version only took 34 minutes.

The GPU algorithm can process a large number of states in a very short time. For example, for the graph `1fjl`, 3680 million states were explored in just 1730 seconds, i.e., over 2 million states were processed each second (and for each state, a $\Theta(|V||E|)$-time algorithm is executed). The highest throughput (2.5 million states/sec.) is achieved on `SylvesterGraph`, but this graph has relatively few vertices.

We caution the reader that the graph names are somewhat ambiguous. For instance, the `queen7_7` instance is from libtw and has treewidth 35. The 2016 PACE instances include a graph called `dimacs_queen7_7` which only has treewidth 28. The instances used in our evaluation are available from our GitHub repository [112].

7.4.3. Work Size and Global v.s. Shared Memory

In this section, we study the effect of work size and whether shared or global memory is used on the running time of our implementation.

Recall that shared memory is a small amount (in our case, 96KiB) of memory that is physically close to each Streaming Multiprocessor, and is therefore in principle faster than the (much larger, off-chip) global memory. We would therefore expect that our implementation is faster when used with shared memory.

Each SM contains 128 CUDA cores, and thus 4 warps of 32 threads each can be executed simultaneously on each SM. The work size (which should be a multiple of 32 items), represents the number of threads we assign to each SM. If we set the work size larger than 128 items, more threads than can physically be executed at once are assigned to one SM. The SM can then switch between executing different warps, for instance to hide latency of memory accesses. If the work size is smaller than 128 items, a number of CUDA cores will be unutilized.

In Table 7.2, we present some experiments that show running times on several graphs, depending on whether shared memory or global memory is used, for several sizes of work group (which is the number of threads allocated to a single SM).

There is not much difference between running the program using shared or global memory. In most instances, the shared memory version is slightly faster. Surprisingly, it also appears that the work size used does not affect the running time significantly. This suggests that our program is limited by the throughput of memory, rather than being computationally bound.

Name	$\|V\|$	tw	Time (sec.)			
			$W = 32$	$W = 64$	$W = 128$	$W = 256$
1igd_graph (G)	59	25	109	107	107	107
1igd_graph (S)	59	25	94,8	95,6	98,2	103
1ku3_graph (G)	60	22	238	235	235	235
1ku3_graph (S)	60	22	214	217	222	230
DoubleStarSnark (G)	30	6	34,3	34,5	34,5	34,5
DoubleStarSnark (S)	30	6	32,8	32,8	32,8	32,9
DyckGraph (G)	32	7	278	281	280	281
DyckGraph (S)	32	7	266	266	266	267
NonisotropicUnitaryPolarGraph_3_3 (G)	63	53	1,26	1,15	1,16	1,17
NonisotropicUnitaryPolarGraph_3_3 (S)	63	53	1,05	1,02	1,03	1,07
queen7_7 (G)	49	35	1,04	0,969	0,967	0,977
queen7_7 (S)	49	35	0,860	0,861	0,876	0,890
queen8_8 (G)	64	45	29,5	26,6	26,3	26,0
queen8_8 (S)	64	45	25,1	24,1	24,5	25,0

Table 7.2. Running time (sec.) for various work group sizes (W), using shared (S) or global (G) memory. Each cell lists the average result of 4 test runs, where the complete set of runs was executed in a randomized order.

Name	$\|V\|$	tw	Time (sec.)		Exp
			GPU	CPU	
1e0b_graph	55	24	721	-	1730 $\times 10^6$
1fjl_graph*	57	26	1600	-	3660 $\times 10^6$
1igd_graph	59	25	98,2	5120	261 $\times 10^6$
1ku3_graph	60	22	222	-	542 $\times 10^6$
1ubq*	47	~~11~~	1040	-	2290 $\times 10^6$
8x6_torusGrid*	48	~~7~~	1040	-	2080 $\times 10^6$
BN_97	48	18	944	-	2310 $\times 10^6$
BN_98	47	21	643	-	1590 $\times 10^6$
contiki_dhcpc_handle_dhcp*	39	6	1350	-	2840 $\times 10^6$
DoubleStarSnark	30	6	32,8	873	87,6 $\times 10^6$
DyckGraph	32	7	266	-	639 $\times 10^6$
HarborthGraph*	40	5	647	-	1530 $\times 10^6$
KneserGraph_8_3*	56	~~24~~	1580	-	4100 $\times 10^6$
McGeeGraph	24	7	1,24	25,3	3,85 $\times 10^6$
myciel4	23	10	0,238	0,460	0,0978 $\times 10^6$
myciel5*	47	19	1850	70.600	3990 $\times 10^6$
NonisotropicUnitaryPolarGraph_3_3	63	53	1,03	60,4	1,56 $\times 10^6$
queen5_5	25	18	0,179	0,0230	0,00313 $\times 10^6$
queen6_6	36	25	0,241	0,389	0,0360 $\times 10^6$
queen7_7	49	35	0,875	43,5	1,90 $\times 10^6$
queen8_8	64	45	24,5	2040	57,9 $\times 10^6$
RandomBarabasiAlbert_100_2*	41	12	1470	-	3260 $\times 10^6$
RandomBoundedToleranceGraph_60	59	30	0,263	0,635	0,0560 $\times 10^6$
SylvesterGraph	36	15	229	-	632 $\times 10^6$
te*	62	~~7~~	1100	-	2140 $\times 10^6$
water	21	9	0,207	0,00600	0,00124 $\times 10^6$

Table 7.3. The same experiment as in Table 7.1, but using shared instead of global memory. Work size 128 items.

7.4.4. Minor-Min-Width

In Table 7.4, we list results obtained when using Minor-Min-Width to prune states.

The computational expense of using MMW is comparable to that of the initial computation (for determining the degree of vertices): the algorithm does a linear search for a minimum degree vertex (using the precomputed degree values), and then does a graph traversal (using BFS) to find a minimum degree neighbour (recall that we do not store the intermediate graph, and use only a single copy of the original graph). Once such a neighbour is found, the contraction is performed (by updating the disjoint set data structure) and another graph traversal is required (to compute the number of common neighbours, and thus update the degree of the vertex).

The lower bound given by MMW does not appear to be very strong, at least for the graphs considered in our experiment: the reduction in number of states expanded is not very large (for instance, from 1730 million states to 1660 million for 1e0b, or from 1590 million to 1480 million for BN_98). The largest reductions are visible for graphs on which we run out of memory (for instance, from 4130 million to 1330 million for

Name	$\|V\|$	tw	With MMW		Without MMW	
			Time	Exp	Time	Exp
1e0b_graph	55	24	2750	1660×10^6	779	1730×10^6
1fjl_graph*	57	26	timeout	3260×10^6	1730	3680×10^6
1igd_graph	59	25	471	235×10^6	107	261×10^6
1ku3_graph	60	22	1090	511×10^6	235	542×10^6
1ubq*	47	~~11~~	2010	1500×10^6	1130	2300×10^6
8x6_torusGrid*	48	~~7~~	1350	1300×10^6	1110	2100×10^6
BN_97	48	18	2260	2020×10^6	1020	2310×10^6
BN_98	47	21	1480	1440×10^6	689	1590×10^6
contiki_dhcpc_handle_dhcp*	39	6	2670	2900×10^6	1490	2930×10^6
DoubleStarSnark	30	6	38,3	$76{,}0 \times10^6$	34,5	$87{,}6 \times10^6$
DyckGraph	32	7	343	592×10^6	280	639×10^6
HarborthGraph*	40	5	1460	1570×10^6	1710	1540×10^6
KneserGraph_8_3*	56	~~24~~	1330	1220×10^6	1730	4130×10^6
McGeeGraph	24	7	1,88	$3{,}42 \times10^6$	1,30	$3{,}85 \times10^6$
myciel4	23	10	0,614	$0{,}0751 \times10^6$	0,234	$0{,}0978 \times10^6$
myciel5*	47	19	2550	3200×10^6	2000	4000×10^6
NonisotropicUnitaryPolarGraph_3_3	63	53	3,36	$1{,}30 \times10^6$	1,16	$1{,}56 \times10^6$
queen5_5	25	18	0,810	$0{,}00291 \times10^6$	0,212	$0{,}00313 \times10^6$
queen6_6	36	25	1,16	$0{,}0308 \times10^6$	0,254	$0{,}0360 \times10^6$
queen7_7	49	35	2,91	$1{,}75 \times10^6$	0,966	$1{,}90 \times10^6$
queen8_8	64	45	83,5	$51{,}1 \times10^6$	26,3	$57{,}9 \times10^6$
RandomBarabasiAlbert_100_2*	41	12	2390	2840×10^6	1610	3280×10^6
RandomBoundedToleranceGraph_60	59	30	0,630	$0{,}0478 \times10^6$	0,274	$0{,}0560 \times10^6$
SylvesterGraph	36	15	274	503×10^6	248	632×10^6
te*	62	~~10~~	2260	1690×10^6	1170	2160×10^6
water	21	9	0,410	$0{,}000938 \times10^6$	0,197	$0{,}00124 \times10^6$

Table 7.4. The effect of using the Minor-Min-Width Heuristic. Time is in seconds. Global memory, work size 128 items.

KneserGraph_8_3), but this is likely because the search is terminated before we reach the actual treewidth (so we avoid the part of our search where using a heuristic is least effective) and there are no graphs on which we previously ran out of memory for which MMW allows us to determine the treewidth (the biggest improvement is that we are able to determine that te has treewidth at least 10, up from treewidth at least 7).

Consistent with the relatively low reduction in the number of states expanded, we see the computation using MMW typically takes around 2–3 times longer. On the graphs considered here, the reduction in search space offered by MMW does not offset the additional cost of computing it.

Again, the GPU version is significantly faster than executing the same algorithm on the CPU: we observed a 55× speedup for queen8_8. Still, given what we observed in Section 7.4.3, it is not clear whether our approach of not storing the intermediate graphs explicitly is indeed the best approach. Our main motivation for taking this approach was to be able to store the required data structures entirely in shared memory, but our experiments indicate that for MMW, using global memory gives better performance than using shared memory. However, the relatively good performance of global memory

Name	$\lvert V\rvert$	tw	With MMW		Without MMW	
			Time	Exp	Time	Exp
1e0b_graph	55	24	3650	1660 $\times 10^6$	721	1730 $\times 10^6$
1fjl_graph*	57	26	timeout	2440 $\times 10^6$	1600	3660 $\times 10^6$
1igd_graph	59	25	645	235 $\times 10^6$	98,2	261 $\times 10^6$
1ku3_graph	60	22	1460	511 $\times 10^6$	222	542 $\times 10^6$
1ubq*	47	~~11~~	2510	1480 $\times 10^6$	1040	2290 $\times 10^6$
8x6_torusGrid*	48	~~7~~	1890	1310 $\times 10^6$	1040	2080 $\times 10^6$
BN_97	48	18	3130	2020 $\times 10^6$	944	2310 $\times 10^6$
BN_98	47	21	1970	1440 $\times 10^6$	643	1590 $\times 10^6$
contiki_dhcpc_handle_dhcp*	39	6	3270	2860 $\times 10^6$	1350	2840 $\times 10^6$
DoubleStarSnark	30	6	50,9	76,0 $\times 10^6$	32,8	87,6 $\times 10^6$
DyckGraph	32	7	440	592 $\times 10^6$	266	639 $\times 10^6$
HarborthGraph*	40	5	1900	1560 $\times 10^6$	647	1530 $\times 10^6$
KneserGraph_8_3*	56	~~24~~	1880	1210 $\times 10^6$	1580	4100 $\times 10^6$
McGeeGraph	24	7	2,02	3,42 $\times 10^6$	1,24	3,85 $\times 10^6$
myciel4	23	10	0,955	0,0751 $\times 10^6$	0,238	0,0978 $\times 10^6$
myciel5*	47	19	3370	3180 $\times 10^6$	1850	3990 $\times 10^6$
NonisotropicUnitaryPolarGraph_3_3	63	53	4,17	1,30 $\times 10^6$	1,03	1,56 $\times 10^6$
queen5_5	25	18	0,704	0,00291 $\times 10^6$	0,179	0,00313 $\times 10^6$
queen6_6	36	25	0,998	0,0308 $\times 10^6$	0,241	0,0360 $\times 10^6$
queen7_7	49	35	3,64	1,75 $\times 10^6$	0,837	1,90 $\times 10^6$
queen8_8	64	45	116	51,1 $\times 10^6$	24,5	57,9 $\times 10^6$
RandomBarabasiAlbert_100_2*	41	12	3080	2830 $\times 10^6$	1470	3260 $\times 10^6$
RandomBoundedToleranceGraph_60	59	30	0,666	0,0478 $\times 10^6$	0,263	0,0560 $\times 10^6$
SylvesterGraph	36	15	368	503 $\times 10^6$	229	632 $\times 10^6$
te*	62	~~10~~	3100	1630 $\times 10^6$	1100	2140 $\times 10^6$
water	21	9	0,543	0,000938 $\times 10^6$	0,207	0,00124 $\times 10^6$

Table 7.5. The same experiment as in Table 7.4, but using shared instead of global memory. Work size 128 items.

might be (partially) due to caching and the small amount of data transferred, so it is an interesting open question to determine whether the additional memory costs of using more involved data structures is compensated by the potential speedup.

7.4.5. Loop Unnesting

Finally, we experimented with another technique, which aims to increase parallelism (and thus speedup) by limiting branch divergence. However, as the results were discouraging, we limit ourselves to a brief discussion.

The algorithm of Algorithm 7.1 consists of a loop (lines 5–22) over the (not yet eliminated) vertices, inside of which is a depth-first search (which computes the degree of the vertex, to determine whether it can be eliminated). The depth-first search in turn consists of a loop which runs until the stack becomes empty (lines 10–19) inside of which is a final loop over the neighbours of the current vertex (lines 12–18). This leads to two sources of branch divergence:

- First, if the graph is irregular, all threads in a warp have to wait for the thread that is processing the highest degree vertex, even if they only have low-degree vertices.
- Second, all threads in a warp have to wait for the longest of the BFS searches to finish before they can start processing the next vertex.

To alleviate this, we propose a technique which we call *loop unnesting*: rather than have 3 nested loops, we have only one loop, which simulates a state machine with 3 states: (1) processing the adjacency list of a vertex, (2) having finished processing of an adjacency list and being ready to pop a new vertex off the queue, or (3) having finished a BFS, and being ready to begin computing the degree of a new vertex.

We considered a slightly more general version of this idea: in an (x, y)-unnesting of our program, after every x iterations of the inner loop (exploring neighbours of the current vertex) one iteration of the middle loop is executed (if exploring the adjacency list is finished, get a new vertex from the queue), and for every y iterations of the middle loop, one iteration of the outer loop is executed (begin processing an entirely new vertex). Thus, a $(1, 1)$-unrolling corresponds to the state machine simulation described above, and an (∞, ∞)-unrolling corresponds to the original program.

Picking the right values for x, y means finding the right trade-off between checking frequently enough whether a thread is ready to start working on another vertex, and the cost of performing those checks. What we observed was surprising: while $(1, 1)$, $(3, 2)$ and $(1, \infty)$-unrollings gave reasonable results, the best results were obtained with (∞, ∞)-unrollings (i.e. the original, unmodified algorithm) and the performance of $(\infty, 1)$-unrollings was abysmal.

Name	$\|V\|$	tw	Time (sec.)				
			$(1,1)$	(∞,∞)	$(1,\infty)$	$(\infty,1)$	$(3,2)$
1igd_graph (G)	59	25	121	107	113	313	112
1igd_graph (S)	59	25	128	98,2	105	241	114
1ku3_graph (G)	60	22	268	235	247	626	248
1ku3_graph (S)	60	22	291	222	238	490	262
DoubleStarSnark (G)	30	6	35,3	34,5	34,6	38,8	34,7
DoubleStarSnark (S)	30	6	32,9	32,8	32,9	37,7	32,8
DyckGraph (G)	32	7	289	280	281	316	283
DyckGraph (S)	32	7	266	266	267	306	266
NonisotropicUnitaryPolarGraph_3_3 (G)	63	53	2,17	1,16	1,51	7,11	1,73
NonisotropicUnitaryPolarGraph_3_3 (S)	63	53	2,17	1,03	1,40	5,16	1,68
queen7_7 (G)	49	35	1,37	0,967	1,13	4,51	1,20
queen7_7 (S)	49	35	1,31	0,876	1,02	3,30	1,12
queen8_8 (G)	64	45	52,2	26,3	34,3	148	42,7
queen8_8 (S)	64	45	55,4	24,5	33,8	111	43,2

Table 7.6. Results on loop unnesting. Work size used was 128 items. Each cell lists the average result of 4 test runs, where the complete set of runs was executed in a randomized order.

We believe that a possible explanation may be that loop unnesting does work to some extent, but not unnesting the loops has the advantage that all BFS searches running simultaneously start from the same initial vertex, and (up to differences caused by different sets S being used) will access largely the same values from the adjacency lists at the same time, which may increase the efficiency of read operations. On the other hand, $(\infty, 1)$-unnesting can not take advantage of either phenomenon: different initial vertices may be processed at any given time (so there is little consistency in memory accesses) and the inner loop is not unnested at all so there is no potential to gain speedup there either. Perhaps for larger graphs, where the difference in length of adjacency lists may be more pronounced, or the amount of time a BFS takes varies more strongly with the initial vertex and S, loop unnesting does provide speed up, but for the graphs considered here it does not appear to be a beneficial choice.

7.5. Conclusions

We have presented an algorithm that computes treewidth on the GPU, achieving a very large speedup over running the same algorithm on the CPU. Our algorithm is based on the classical $O^*(2^n)$-time dynamic programming algorithm [20] and our results represent (promising) first steps in speeding up dynamic programming for treewidth on the GPU. The current best known practical algorithm for computing treewidth is the algorithm due to Tamaki [104]. This algorithm is much more complicated, and porting it to the GPU would be a formidable challenge but could possibly offer an extremely efficient implementation for computing treewidth.

Given the large speedup achieved, we are no longer mainly limited by computation time. Instead, our ability to solve larger instances is hampered by the memory required to store the very large lists of partial solutions. Using minor-min-width did not prove effective in reducing the number of states considerably, so it would be interesting to see how other heuristics and pruning rules (such as simplicial vertex detection) could be implemented on the GPU.

GPUs are traditionally used to solve easy (e.g. linear time) problems on very large inputs (such as the millions of pixels rendered on a screen, or exploring a graph with millions of nodes), but clearly, the speedup offered by inexpensive GPUs would also be very welcome in solving hard ($\mathcal{NP}$-complete) problems on small instances. Exploring how techniques from FPT and exact algorithms can be used on the GPU raises many interesting problems - not only practical ones, but also theoretical: how should we model complex devices such as GPUs, with their many types of memory and branch divergence issues?

Source Code and Instances. We have made our source code, as well as the graphs used for the experiments, available on GitHub [112].

8 Computing the Shapley Value of Connectivity Games

8.1. Introduction

Motivated by the need to identify important vertices in networks, many measures for ranking vertices have been suggested. Among these are the classical centrality measures, such as betweenness, closeness [61] and – perhaps the simplest – degree. However, what makes a "good" centrality measure is subjective and strongly dependent on the application. In light of this, *game-theoretic centrality* measures have received considerable interest. By viewing subgraphs of the input network as coalitions in a coalitional game, we can apply game-theoretic measures (such as the Shapley value [101] or Banzhaf power index [10]) to determine the importance of vertices (and/or edges) in a network.

The game-theoretic approach is very flexible, since we can adapt the underlying game to obtain different measures. It can also take information other than the network structure into account [82]. However, as both the Shapley value and Banzhaf index are defined as an expression in terms of all possible coalitions, save for a few special cases (see e.g. [93]), they cannot be computed efficiently.

In this chapter, we consider two game-theoretic centrality measures: one based on the Shapley value associated with a $\{0, 1\}$-valued game due to Amer and Gimenez [6], the second a vertex-weighted variant of the same due to Lindelauf et al. [82]. Lindelauf et al. studied these centrality measures in the context of identifying the most important vertices in terrorist networks [82, 107]. They considered two networks: one (due to Koschade [78]) consisting of 17 terrorists involved in a 2002 bombing in Bali, the second (due to Krebs [79]) consisting of 69 terrorists involved in planning and executing the 9/11 attacks. Whereas for the first network they were able to compute the exact Shapley value, for the second network this was infeasible and they considered only the part of the network made up by the 19 hijackers that actually carried out the attack.

Michalak et al. [94] showed that computing the Shapley value for the unweighted game is unfortunately $\#\mathcal{P}$-hard. As such, it is unlikely that there exists an efficient algorithm for computing these values. On the other hand, Michalak et al. proposed an algorithm that is slightly more efficient than the brute-force approach used by Lindelauf et al., called *FasterSVCG*. Using this algorithm, the authors computed the Shapley value for a larger version of the 9/11 network, with 36 vertices (corresponding to the hijackers and some accomplices). Their approach, rather than considering all $2^{|V|}$ coalitions, considers only the *connected* coalitions, of which there may be considerably fewer than $2^{|V|}$. However, in the worst case, the number of connected coalitions may still be exponential. As such, for the full 69-vertex network, running this algorithm is still infeasible.

Michalak et al. [94] also considered an approximation method based on random sampling and studied its performance on the 36-vertex 9/11 network. Lindelauf et al. [107] proposed a different sampling method, *structured random sampling*, that aims to be more efficient than random sampling. Using this method, they computed an approximation of the Shapley value for the 69-vertex 9/11 network. Unfortunately, neither method comes with any formal guarantees on the quality of the approximation.

While, in general, one should not expect to find an efficient algorithm for computing the exact Shapley value of these games (due to the $\#\mathcal{P}$-hardness), we can attempt to exploit the structure that the networks may have in order to obtain more efficient algorithms.

We show that the Shapley value (for both the weighted and the unweighted game) can be computed efficiently on graphs of bounded treewidth. Our result is not merely theoretical: we also provide an implementation and show that it can be used to compute Shapley values for graphs of practical interest.

Using our approach, we are able to compute the exact Shapley value for the full 69-vertex network of the 9/11 attacks. Given the exact values, we evaluate the approximation computed by Lindelauf et al. [107]. We are also able to compute the exact Shapley value for some much larger networks, having up to a thousand vertices.

Of course, our method crucially depends on the network having bounded treewidth. Fortuitously, the network of the 9/11 attacks has a rather low treewidth of only 8. In general one cannot expect social networks to have small treewidth: social networks often have large cliques, and the size of the largest clique forms a lower bound on the treewidth of a graph (see e.g. [1] for a study of tree decompositions of social networks). However, terrorist and criminal organizations are often well-served by keeping their networks sparsely connected, as this helps to avoid detection and as such one would not expect large cliques. As another example of networks that may have low treewidth, the interaction networks in a hierarchical organization would naturally be tree-like.

Our goal is to develop an algorithm that, given a graph G with $n = |V|$ and tree decomposition of G of width tw, computes the Shapley value in time $f(tw)n^{O(1)}$, where f is some exponential function and $n^{O(1)}$ a (low-degree) polynomial. As such, we hope to "hide" the exponential behaviour of computing ϕ in a function that depends only on the treewidth of the graph, and obtain an algorithm whose running time is (for graphs of bounded treewidth) polynomial in n.

Specifically, we show that for a graph G of treewidth tw and a vertex $v \in V$, $\phi_v(v^{wconn2})$ can be computed in time $2^{O(tw \log tw)} n^4 \log n$. Note that our algorithm

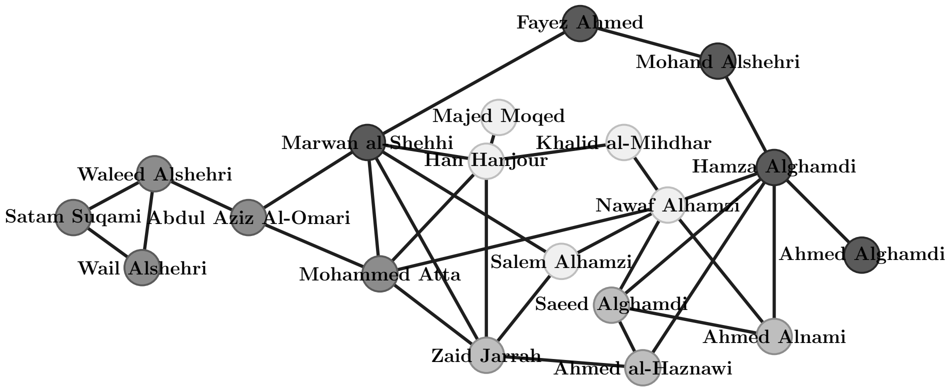


Figure 8.1. Graph showing the connections between the 19 hijackers that carried out the 9/11 attacks. Vertices are coloured according to the flights they were on. Note that the full network consists of 69 vertices.

for computing the Shapley value requires multiplying large ($O(n)$-bit) integers; this running time is achieved if using the $O(n \log n)$-time algorithm of Harvey and van der Hoeven [63]. Moreover, we usually want to know $\phi_v(v^{wconn2})$ for all $v \in V$ rather than for a specific vertex. Rather than running the previous algorithm n times, we also show that computing $\phi(v)$ for all $v \in V$ can be done in the same time, by reusing the intermediate results of previous computations.

For instance, the graph considered by Lindelauf et al. that represents the communications between the perpetrators of the 9/11 attacks, consists of 69 vertices but only has treewidth 8. While evaluating all 2^{69} subsets of vertices is clearly infeasible, our algorithm can compute $\phi(v)$ in a couple of seconds thanks to the low treewidth of the graph.

Given a graph $G = (V, E)$, the algorithm of Michalak et al. [94] runs in time $O((|V| + |E|)|C|)$[1], where $|C|$ denotes the number of connected induced subgraphs of G. This algorithm, while offering a moderate improvement over the brute-force approach still requires exponential time on almost all interesting classes of graphs. We remark that there exist graphs of low treewidth that have a very large number of connected induced subgraphs (for example, the star on n vertices has treewidth 1 and more than 2^{n-1} induced connected subgraphs), while graphs with a small number of induced connected subgraphs also have low treewidth: a graph with at most $|C|$ induced connected subgraphs has treewidth at most $2 \log |C|$ (in fact, pathwidth at most $2 \log |C|$: if we fix some arbitrary vertex v, then there are at most $\log |C|$ vertices at distance exactly r from v). While this bound is tight up to a constant factor (for instance on an n-vertex clique), in many instances the treewidth is much smaller than $\log |C|$.

Recently, Greco et al. [56] proposed using treewidth to compute Shapley values

[1]The authors of [94] ignore in the analysis of their running time the fact that the numbers involved in the computation of the Shapley value can get exponentially large, and thus we can no longer presume that arithmetic operations can be done in $O(1)$ time. Our running times do account for this, and are thus a factor $n \log n$ higher.

for matching games in graphs. In the matching game, the value of a coalition is the size of the maximum matching in the graph induced by that coalition. However, their algorithm is based on a formulation in Monadic Second Order Logic and the application of theoretical frameworks that allow counting of satisfying assignments of MSO formulas. For graphs of bounded treewidth, this yield a polynomial-time algorithm, where the degree of the polynomial may depend on the treewidth. In contrast, the degree of the polynomial in our algorithm is fixed, and only the constant factor in the running time is affected by the treewidth (i.e., we obtain a *fixed-parameter tractable* algorithm). Moreover, due to the application of these algorithmic metatheorems, their algorithm is not very efficient in practice: Greco et al. [56] report that, even for a graph of treewidth only 3 with 30 vertices, their implementation (using the MSO solver Sequoia [80]) took nearly 9 minutes to determine the Shapley value. We are able to process much more complex (i.e., higher treewidth) graphs with significantly more vertices in a much shorter time.

8.2. Preliminaries

8.2.1. Shapley Value / Power Indices

A *coalitional game* consists of a set of players N (the *grand coalition*) together with a *characteristic function* $v : 2^N \to \mathbb{R}$ such that $v(\emptyset) = 0$. In this chapter, we consider coalitional games where the players correspond to vertices in a graph. Given a characteristic function, the Shapley value $\Phi_i(v)$ of a player i is defined as [101]:

$$\Phi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(|N| - |S| - 1)!}{|N|!} (v(S \cup \{i\}) - v(S)) \tag{8.1}$$

While this chapter focuses primarily on the Shapley value, we remark that our techniques can also be used to compute the Banzhaf Value, which is defined as [10]:

$$\Phi_i^B(v) = \frac{1}{2^{|N|-1}} \sum_{S \subseteq N \setminus \{i\}} (v(S \cup \{i\}) - v(S)) \tag{8.2}$$

Essentially the same techniques may be used to compute the Banzhaf value as we use for computing the Shapley value, merely a change in weighting factors is required.

8.2.2. Game-Theoretic Centrality

The connectivity game v^{conn}, introduced by Amer and Giminez [6], is given by the weight function:

$$v^{conn}(S) = \begin{cases} 1 & \text{if } G[S] \text{ is connected and } |S| > 1, \\ 0 & \text{otherwise.} \end{cases}$$

Note that a coalition consisting of a single player, while connected, has value 0.

Lindelauf et al. [82] consider several weighted variants of this connectivity game. They assume each vertex i has a weight w_i and each edge (between vertices i, j) a weight f_{ij}. They consider three games: one that uses vertex weights, one that uses edge weights, and one that combines both. The games are given by the following weight functions:

$$v^{wconn1}(S) = \begin{cases} \max_{i,j\in S, i\neq j} f_{ij} & \text{if } G[S] \text{ is connected,} \\ 0 & \text{otherwise.} \end{cases}$$

$$v^{wconn2}(S) = \begin{cases} \sum_{i\in S} w_i & \text{if } G[S] \text{ is connected,} \\ 0 & \text{otherwise.} \end{cases}$$

$$v^{wconn3}(S) = \begin{cases} (\sum_{i\in S} w_i) \cdot \max_{i,j\in S, i\neq j} f_{ij} & \text{if } G[S] \text{ is connected,} \\ 0 & \text{otherwise.} \end{cases}$$

We give an algorithm for computing v^{wconn2}, that, as a byproduct, also computes v^{conn}. Similar techniques can be used to compute v^{wconn1} and v^{wconn3}, though this would increase the running time by a factor of n.

Henceforth, we shall use the letter v to refer to vertices of G, rather than to a game (but we still use the notation v^{conn} and v^{wconn2} to denote games). In the following, we let $n = |V(G)|$.

8.3. An Algorithm for Computing Shapley Values

In this section, we present our algorithm for computing the Shapley value of v^{conn} and v^{wconn2}.

Theorem 8.1. *Given a graph $G = (V, E)$ and a tree decomposition T of width tw, the Shapley value $\phi_v(v^{conn}), v \in V$ and $\phi_v(v^{wconn2}), v \in V$ can be computed in time $2^{O(tw \log tw)} n^4 \log n$.*

We give the algorithm for computing $\phi_v(v^{wconn2})$; the results obtained from this algorithm can also be used to compute $\phi_v(v^{conn})$. We first show that for a given $v \in V$, the (single) value $\phi_v(v^{wconn2})$ can be computed in time $2^{O(tw \log tw)} n^4 \log n$. Next, note that computing a table of the values $\phi_v(v^{wconn2})$ for all $v \in V$ would require running this algorithm n times, and thus have a time complexity of $2^{O(tw \log tw)} n^5 \log n$. To avoid this, we then show that by reusing intermediate results of the computation, we can preserve the $2^{O(tw \log tw)} n^4 \log n$ running time.

We begin by rewriting Equation 8.1 to obtain the following more suitable expression for computing $\phi_v(v^{wconn2})$, by splitting the summation into different terms, depending on the cardinality k of S:

$$\phi_v(v^{wconn2}) = \sum_{k=0}^{|V|-1} \left(\frac{|S|!(|V|-|S|-1)!}{|V|!} \sum_{S\subseteq V\setminus\{v\},|S|=k} (v^{wconn2}(S\cup\{v\}) - v^{wconn2}(S)) \right)$$
$$= \sum_{k=0}^{|V|-1} \left(\frac{|S|!(|V|-|S|-1)!}{|V|!} \Big(\sum_{S\subseteq V\setminus\{v\},|S|=k} v^{wconn2}(S\cup\{v\}) - \sum_{S\subseteq V\setminus\{v\},|S|=k} v^{wconn2}(S) \Big) \right)$$

Since $v^{wconn2}(S) = 0$ whenever S induces a subgraph with more than one connected component, the problem of computing $\phi_v(v^{wconn2})$ reduces to computing, for each k, the total value over all connected subsets $S \subseteq V(G)$ with $|S| = k$ and $v \in S$, respectively $v \notin S$.

In the following, we assume that we are given a graph $G = (V, E)$ and a nice tree decomposition of G with width tw. Furthermore, we assume that the bag X_r associated with the root node r of T contains only a single vertex v (which is the vertex of which we will compute the Shapley value).

As is standard for algorithms using dynamic programming on tree decompositions, for each node t of the tree decomposition we consider the subgraph $G[t]$. For each such subgraph, we consider the subsets (coalitions) $S \subseteq G[t]$ (i.e., all subsets of vertices appearing in the bag X_t or a bag below it in the decomposition; see Chapter 2).

Call a subset $S \subseteq G[t]$ *good* if $G[S]$ is connected or every connected component of $G[S]$ has a non-empty intersection with X_t.

Our algorithm considers, for each $t \in T$, all good subsets $S \subseteq V(G[t])$. The following lemma shows that subsets that are not good do not count towards the Shapley value of the game.

Lemma 8.2. *Let $S \subseteq V(G)$ induce a connected subgraph of G and let $t \in T$. Then either each connected component of the subgraph induced by $S \cap V(G[t])$ has a non-empty intersection with X_t or $S \cap V(G[t])$ is connected.*

Proof. By contradiction. Suppose $S \cap V(G[t])$ is not connected. Then some component of $S \cap V(G[t])$ has an empty intersection with X_t. Then S cannot be connected, since X_t separates $G[t]$ from the rest of the graph. □

Of course, there can still be exponentially many good subsets. The key to our algorithm is that for each such subset S, we do not need to know exactly how the subset is made up: if we know how the subset S behaves within X_t, we know how it interacts with the rest of the graph (outside of $G[t]$), since X_t is a separator. Subsets which behave similarly within X_t can be grouped together, thus speeding up the computation. Specifically, each subset $S \subseteq G[t]$ has a *characteristic* (w.r.t. $G[t]$) that consists of

- the intersection $R = S \cap X_t$,
- an equivalence relation $\sim$ on $S \cap X_t$ such that $a \sim b$ if and only if a and b are in the same connected component of the subgraph induced by S,
- the cardinality of S, $k = |S|$.

The number of distinct characteristics is $2^{O(tw \log tw)}n$. For every node $t \in T$ and each characteristic $(R, \sim, k)$, we will compute

- $n_t(R, \sim, k)$: the number of good subsets $S \subseteq G[t]$ with characteristic $(R, \sim, k)$,
- $w_t(R, \sim, k)$: the total weight of all good subsets $S \subseteq G[t]$ with characteristic $(R, \sim, k)$.

We define the weight of a subset $S \subseteq G[t]$ as the sum of the vertex weights, i.e. $\sum_{v \in S} w(v)$.

Suppose that r is the root of T, and $X_r = \{v\}$. Then $w_r(\{v\}, \{(v, v)\}, k)$ is exactly the total weight of connected subsets $S \subseteq V(G)$ with $|S| = k$ and $v \in S$, whereas $w_r(\emptyset, \emptyset, k)$ is the total weight of connected subsets of size k not including v. This gives us exactly the information we need to compute $\phi(v)$.

For every node $t \in T$ we compute $n_t(R, \sim, k)$ and $w_t(R, \sim, k)$ for each characteristic $(R, \sim, k)$ in a bottom-up fashion. We may start at the leaf vertices, and then work our way up to the root of the tree. We handle each of the cases as follows:

Leaf. If $t \in T$ is a leaf node, then $X_t = \{v\}$ for some $v \in V$. Therefore t has exactly two characteristics $c_1 = (\emptyset, \emptyset, 0)$ and $c_2 = (\{v\}, \{(v, v)\}, 1)$. It is easy to see that $n_t(c_1) = 1, w_t(c_1) = 0$ and $n_t(c_2) = 1, w_t(c_1) = w(v)$.

Introduce. If $t \in T$ is an introduce node, then it has a child $u \in T$ such that $X_t = X_u \dot{\cup} \{v\}$ for some $v \in V(G)$. Every characteristic $(R, \sim, k)$ (w.r.t. $G[u]$) corresponds to $n_u(R, \sim, k)$ distinct subsets of $G[u]$, and we may extend these subsets to subsets of $G[t]$ by either adding the introduced vertex v or not. Thus, the $n_u(R, \sim, k)$ subsets of $G[u]$ give rise to

- $n_u(R, \sim, k)$ good subsets of $G[t]$ with characteristic $(R, \sim \cup \{(v, v)\}, k)$ and total weight $w_t(R, \sim, k)$, and, if $k = 0$ or $R \neq \emptyset$ to
- $n_u(R, \sim, k)$ good subsets of $G[t]$ with characteristic $(R \cup \{v\}, \sim', k + 1)$ and total weight $w_u(R, \sim, k) + n_u(R, \sim, k) \cdot w(v)$,

where $\sim'$ is the relation obtained as the transitive closure of $\sim \cup \{(v, v)\} \cup \{(v, x) \mid x \in R, (v, x) \in E(G)\}$.

Note that two distinct characteristics $(R, \sim, k)$ and $(R', \sim', k')$ with $R = R'$, $k = k'$ and $\sim \neq \sim'$ may give rise (upon addition of the vertex v) to $n_u(R, \sim, k) + n_u(R', \sim', k')$ subsets with the same characteristic $(R \cup \{v\}, \sim'', k + 1)$ with total weight $w_u(R, \sim, k) + w_u(R', \sim', k') + (n_u(R, \sim, k) + n_u(R', \sim', k')) \cdot w(v)$. Indeed, subsets corresponding to many distinct characteristics may end up having the same characteristic when v is added to them, and so the subsets of $G[t]$ with characteristic $(R \cup \{v\}, \sim', k + 1)$ may correspond to subsets of $G[u]$ with characteristics $(R, \sim, k)$ for many different $\sim$ (and thus, to compute n_t and w_t, we should take the sum over all such $\sim$).

The following lemma (cf. Lemma 8.2) ensures the correctness of the introduce step:

Lemma 8.3. *Let $t \in T$ and suppose that t is an introduce node with child $u \in T$. Suppose $S \subseteq G[t]$ is a good subset. Then $S \cap G[u]$ is a good subset of $G[u]$.*

Proof. Suppose that $S \cap G[u]$ is not connected, and some connected component C of $S \cap G[u]$ has an empty intersection with X_u. Suppose the introduced vertex is v. Then v must be adjacent to some vertex of C, but this is impossible since $C \cap X_u = \emptyset$ and v is not incident to $G[u] \setminus X_u$. □

This ensures that we count each good subset $S \subseteq G[t]$ at least once. That we count each good subset $S \subseteq G[t]$ at most once follows from the fact that $S \cap G[u]$ corresponds to a unique characteristic w.r.t $G[t]$.

Forget. If $t \in T$ is a forget node, it has a child $u \in T$ such that $X_u \dot{\cup} \{v\} = X_t$ for some $v \in V(G)$. If for characteristic $(R, \sim, k)$ (w.r.t. $G[u]$), $v \notin R$, then $(R, \sim, k)$ is also a characteristic w.r.t. $G[t]$. If $v \in R$, then there are three cases:

1. $R = \{v\}$. Then we obtain $n_u(R, \sim, k)$ good subsets of $G[t]$ with characteristic $\{\emptyset, \emptyset, k\}$ and total weight $w_t(R, \sim, k)$,
2. $R \neq \{v\}$ and $\{(r, r)\} \in \sim$. Then none of the $n_u(R, \sim, k)$ good subsets of $G[u]$ are good for $G[t]$, since the connected component containing v does not intersect X_t, and there is some other connected component that does intersect X_t.
3. Otherwise, we obtain $n_u(R, \sim, k)$ good subsets of $G[t]$ with characteristic $(R \cap X_t, \sim', k)$,

where $\sim'$ is the relation obtained by projecting the relation $\sim$ on R to $R \cap X_t$ (i.e., $\sim' = \sim \cap \{(u, v) \mid u, v \in X_t\}$).

Similarly to the introduce case, multiple distinct characteristics for u may correspond to the same characteristic for t; we should again take the sum over these characteristics. The correctness follows from the following Lemma:

Lemma 8.4. *Let $t \in T$ and suppose that t is a forget vertex with child $u \in T$. Suppose $S \subseteq G[t]$ is a good subset. Then S is a good subset of $G[u]$.*

Proof. If S is not connected, then S has a non-empty intersection with X_t. Since $X_t \subset X_u$, S also has a non-empty intersection with X_u. □

Join. If $t \in T$ is a join node, then it has two children l, r such that $X_l = X_r = X_t$. Suppose that $(R_l, \sim_l, k_l)$ is a characteristic of l and $(R_r, \sim_r, k_r)$ is a characteristic of r and suppose that $R_l = R_r$. Then there are $n_l(R_l, \sim_l, k_l) \cdot n_r(R_r, \sim_r, k_r)$ subsets with characteristic $(R_l, \sim', k_l + k_r - |R_l|)$ and total weight $n_l(R_l, \sim_l, k_l) \cdot w_r(R_r, \sim_r, k_r) + n_r(R_r, \sim_r, k_r) \cdot w_l(R_l, \sim_l, k_l) - n_l(R_l, \sim_l, k_l) \cdot n_r(R_r, \sim_r, k_r) \cdot (\Sigma_{v \in R_l} w(v))$, where $\sim$ is the transitive closure of $\sim_l \cup \sim_r$.

Similarly to the introduce and forget cases, multiple distinct characteristics for l, r may, when combined, correspond to the same characteristic for t; we should again take the sum over these characteristics. The correctness follows from the following Lemma:

Lemma 8.5. *Let $t \in T$ and suppose that t is a join vertex children $l, r \in T$. Suppose $S \subseteq G[t]$ is a good subset. Then $S \cap V(G[l])$ (resp. $S \cap V(G[r])$) is a good subset of $G[l]$ (resp. $G[r]$).*

Proof. By contradiction. We show the case for the left child, the case for the right child is symmetric.

If $S \subseteq V(G[l])$ then the lemma follows automatically. Therefore, assume there exists $v \in S$ such that $v \notin V(G[l])$. In particular, this means that $v \in G[r] \setminus X_r$.

Suppose $S \cap V(G[l])$ is not connected and has a connected component C with empty intersection with X_l. Since none of the vertices of $S \cap V(G[r])$ are incident to C, C is still a maximal connected component of S, but S has at least one other connected component (since $S \cap V(G[l])$ is not connected) and so is not connected, and C has empty intersection with $X_l = X_t$. □

By processing the vertices of the tree decomposition in a bottom-up fashion, we can compute the values $n_r(R, \sim, k)$ and $w_r(R, \sim, k)$ for all characteristics $(R, \sim, k)$ of the root node r. As we have seen before, knowing these values is sufficient to compute the Shapley value of vertex v. We thus obtain the following result:

Theorem 8.6. *Given a graph G of treewidth at most tw, the Shapley value of a vertex $v \in V(G)$ can be computed in time $2^{O(tw \log tw)} n^4 \log n$ and space $2^{O(tw \log tw)} n^2 \log n$.*

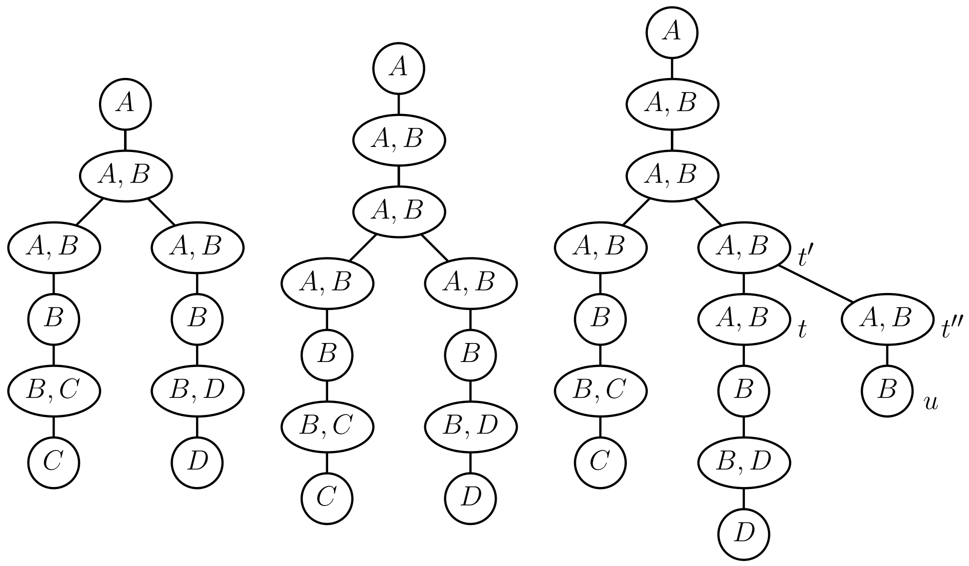
Proof. We assume that we are given a nice tree decomposition of G (which we may assume has $O(n)$ nodes). For each node, there are $2^{O(tw \log tw)} n$ characteristics. To compute the values for one characteristic requires considering (in the worst case, which is the join node) $2^{O(tw \log tw)} n^2$ pairs of characteristics for the child nodes. For each such pair, we perform a constant number of multiplications of $O(n)$-bit integers, taking $O(n \log n)$ time. The dynamic programming table for one node of the tree decomposition takes up $2^{\Theta(tw \log tw)} n^2$ space, but at any given time we only need to keep $O(\log n)$ of them in memory. □

Of course, this only allows us to evaluate the Shapley value for a *single* vertex v, under the assumption that for the root bag r, $X_r = \{v\}$ (i.e., v is the only vertex in the root bag). To compute the Shapley value for *all* vertices, we perform the following operations, starting from a nice tree decomposition:

- For every join node t, we create a new node t' with $X_{t'} = X_t$. t' is made the parent of t, and the original parent of t becomes the parent of t'. In case t was the root, t' becomes the root. Note that t' is neither a join, introduce, forget, or leaf node, however, the dynamic programming tables for t' are simply equal to those for t (we shall from now on, refer to nodes such as t' as nochange nodes).

- For every vertex $v \in V(G)$, we pick a node of the tree decomposition t such that $v \in V_t$. We create a copy t' of t, which is made the parent of t, and the original parent of t becomes the parent of t'. In case t was the root, t' becomes the root. Next, we create another copy t'' of t'. t' is made the parent of t'' (making t' into a join node). We then create a series of introduce nodes, starting from t'', such that eventually we end up with a leaf node u, whose bag contains only v. If we now reroot our tree decomposition so that the root becomes u, thanks to the previous transformation, every join node remains a join node – the roles of introduce, forget and nochange nodes can become interchanged.

Figure 8.2 shows an example of this process. Starting from a nice tree decomposition (Figure 8.2a) a nochange node is added before the join bag A, B (Figure 8.2b). To create a leaf bag for vertex B, we pick a bag t containing it (in this example the right child of the join bag), insert a node t' which becomes the parent of t, create



(a) Nice tree decomposition **(b)** With nochange node added **(c)** Additional nodes for vertex B

Figure 8.2. (a) A (nice) tree decomposition. (b) A nochange node is added before the join bag A, B. (c) Extra nodes t' and t'' are added to enable the creation of a leaf bag containing vertex B, which can be used to re-root the decomposition.

an additional child (of t') t'' (thus making t' into a join node), then add a leaf bag u (containing only B) as child of t'' (making t'' into a forget node).

This process can be repeated until for each vertex $v \in V(G)$ there exists a leaf bag containing it. Note that in the example the tree decomposition is rooted at A, but it can also be viewed as being rooted at u (or any other leaf node); this turns t'' from a forget node into a nochange node, t' remains a join node, while the nochange node A, B (currently a child of the root node A) becomes an introduce node (introducing B to the leaf node containing A).

Thus, we now have a tree decomposition that can be rerooted such that any vertex v becomes the sole vertex in the root bag. However, this only gives a $2^{\Theta(tw \log tw)} n^5 \log n$-time algorithm for computing the Shapley value for all the vertices in a given graph, since this would require running the algorithm separately for each root vertex. However, there is a lot of overlap in these computations, as the dynamic programming tables for each subtree may be computed multiple times: in the example of Figure 8.2, regardless of whether we select the original root bag (containing A) or the node u as root for the computation, the dynamic programming table computed at node t is the same. For each node of the decompositions there are at most three distinct tables that need to be computed: for a join bag there are three possibilities (depending which of its three neighbours is the parent); for an introduce, forget or nochange node there are two (again depending on the direction in which it is evaluated). By memoizing a table when it is computed (similar to belief propagation in Bayesian Networks, see e.g. [99]), we thus obtain a $2^{O(tw \log tw)} n^4 \log n$-time algorithm using $2^{O(tw \log tw)} n^3$ space:

Theorem 8.7. *Given a graph G of treewidth at most tw, the Shapley value of all vertices $v \in V(G)$ can be computed in time $2^{O(tw \log tw)} n^4 \log n$ and space $2^{O(tw \log tw)} n^3$.*

8.4. Computational Experiments

Lindelauf et al. [107] propose a randomized approximation heuristic (however, they do not derive any guarantee on the quality of the approximation) to compute Shapley values. They used this method, called Structured Random Sampling, to compute Shapley values for the 69-node network of interactions between terrorists involved in the 9/11 attacks due to Krebs [79].

These values are based on a random sampling of 10.000 permutations of the players in the network. Lindelauf et al. report that this computation of approximate Shapley values took 5 minutes. On the other hand, using our method for computing the exact Shapley values of this network takes less than 5 seconds.

Ranking	Name	Approximation due to [107]	Exact Value
1	Mohamed Atta	0,1137	0,114099473
2	Essid Sami Ben Khemais	0,1111	0,111249806
3	Hani Hanjour	0,1107	0,110702087
4 (5)	Khalid Almihdhar	0,1069	0,107273424
5 (4)	Djamal Beghal	0,1070	0,107153568
6	Mahmoun Darkazanli	0,1067	0,106635302
7	Zacarias Moussaoui	0,1009	0,101188122
8	Nawaf Alhazmi	0,0995	0,099594346
9	Ramzi Bin al-Shibh	0,0985	0,098434678
10	Raed Hijazi	0,0949	0,094777059
11 (19)	Ahmed Alghamdi	0,0016	0,008901234
12	Fayez Ahmed	0,0088	0,008790340
13	Marwan al-Shehhi	0,0046	0,004533725
14 (15)	Saeed Alghamdi	0,0037	0,003693424
15 (11)	Hamza Alghamdi	0,0090	0,003666945

Table 8.1. Results on the 9/11 networks due to Krebs. Where it differs from ours, shown in parentheses is the rank assigned by Lindelauf et al. [107].

Table 8.1 shows the results of our computation (exact values, albeit reported as decimal representation with finite precision – our method actually computes the exact fraction). Based on our results, we see that the approximation of Lindelauf et al. [107] is quite accurate, correctly identifying the 10 most important players in the network (albeit with the 4th and 5th ranked players swapped) and computing the Shapley value to within three significant digits. However, it overestimates the importance of Hamza Alghamdi (who receives Shapley value 0,009 rather than the correct value of 0,0037) and underestimates the importance of Ahmed Alghamdi: his Shapley value of 0,0089 means he is the 11th ranked player in the network, but he does not appear in the top-15 of Lindelauf et al. [107]. The 14th ranked player according to Lindelauf et al. [107] actually has rank 20, and an exact Shapley value of 0,00145 (rather than 0.0038).

This can perhaps be explained by the fact that while the top-10 vertices have Shapley values $> 0,09$, the remaining vertices all have values $< 0,01$. Thus, there is a rather large gap between the most important players and the rest. Correctly ordering the vertices with low values is relatively hard, since a small change in Shapley value can make a big difference to the rank.

Of course, the method of Lindelauf et al. [107] can be applied to any graph rather than just to graphs of small treewidth. However, it is not yet known how the performance of their approximation depends on the structure of the graph (and it might be the case that it does not work well on graphs of large treewidth). Still, when the treewidth of the graph is small, our method provides an excellent way to compute exact Shapley values.

To evaluate the performance, we also tested our algorithm using several covert networks found in the literature:

- A network of 71 Islamic State members in Europe (ise-extended), where edges represent some kind of tie (such as cooperating in an attack, being related or being present at the same location) [59].
- A network of 293 drug users (drugnet), where edges represent acquaintanceships [115].
- A network of 36 Montreal gangs (montreal), where edges represent ties between gangs [42].
- A network of 67 members of Italian gangs (italian), where an edge represents joint membership of a gang [95].

For each network, we considered the largest connected component. Each of these networks has relatively low treewidth. The Islamic State network has the highest treewidth (13), while the Italian Gang network is very sparse (treewidth 3). We also considered using the Noordins top terrorist network [48]. However, as this 79-vertex network has treewidth at least 19, applying our techniques is not feasible.

Table 8.2 reports results on several benchmark graphs with varying treewidth and number of vertices. Our implementation uses the .NET BigInteger library, which performs multiplications in $\Theta(n^2)$ time using a method similar to grid multiplication. While there are several asymptotically faster methods for multiplication, and we experimented with several such implementations, none of these resulted in a significant speed up for the graphs considered. The time reported is that for computing the Shapley values of *all* vertices in the graph, using the method that stores all intermediate tables to achieve a $2^{O(tw \log tw)} n^4 \log n$ computation time. The time reported does not include the time for computing a tree decomposition.

We performed our experiments on a platform equipped with an Intel Core i7-6700 CPU (3.40GHz) and 32GB of RAM (4x8GB DDR4). In each case, the time reported is the average of five runs.

As is clear from Table 8.2, our algorithm is feasible even for graphs of moderately high treewidth (13) and large numbers of vertices (990). Due to the use of arbitrary

Graph	Treewidth	Vertices	Edges	Time (seconds)
italian	3	65	113	0,40
montreal	6	29	75	0,23
9/11	8	69	163	4,6
drugnet	8	193	273	1.026,5
ise-extended	13	77	274	38,7
pace_005	5	201	253	29,9
pace_012	5	572	662	1.777
pace_022	6	732	1084	20.833
pace_023	6	990	1258	26.429
pace_028	7	139	202	2.003
pace_070	10	106	399	43,6

Table 8.2. Performance of the algorithm on several real-world social networks and several benchmark graphs from the PACE 2018 challenge. For disconnected graphs, we considered only the largest connected component in the graph (for which the number of vertices and edges is given).

precision integers, the polynomial factor in the running time is quite significant and often, the feasibility of our algorithm is determined not by treewidth, but by the number of vertices. Note that several of the covert networks considered in the literature, do indeed have rather small treewidth compared to their size.

8.5. Conclusions

Game-theoretic centrality measures are a powerful tool for identifying important vertices in networks. We have shown that, using treewidth, two game-theoretic centrality measures can be practically computed on graphs much larger than previously feasible, allowing us to analyze larger networks than before.

Our algorithm runs in time $2^{O(tw \log tw)}n^{O(1)}$. The log-factor in the exponent is due to the need to keep track of a connectivity partition. A very interesting open question is whether the algorithm can be improved to have running time single-exponential in n, that is, is it possible to attain a $2^{O(tw)}n^{O(1)}$-time algorithm? For several (counting) problems involving connectivity, this is indeed possible: For instance, it is possible to count Hamiltonian Cycles or Steiner Trees in single-exponential time [15] by using approaches involving matrix determinants. Either a positive answer to this question or a conditional lower bound ruling out such an algorithm would be interesting.

A well-known technique to speed up dynamic programming on tree decompositions is to use subset convolution in the join procedure [114]. In a join node, to compute the values for a specific characteristic $(R, \sim, k)$, we need to consider all $\Theta(k^2)$ ways to split up $k = a + b$. It is likely that this could be improved using convolution-based techniques.

We remark that the log-factor in the exponent represents only the worst case. However, since we are dealing with induced subgraphs, if two vertices share an edge, they can never be in two distinct connected components. Therefore, the actual number of connectivity partitions considered may be lower than suggested by the worst case bound. It would be interesting to see if it is possible to take this phenomenon into account when generating a tree decomposition: perhaps it would be possible to optimize a tree decomposition to limit the number of feasible partitions (for instance, by giving preference to bags that are cliques). Such an approach has previously been considered for Steiner Tree [109].

Another interesting question is which other connectivity measures can be computed using treewidth. For instance, v^{conn} assigns a value of 0 to any disconnected coalition, even if there exists a large connected component. It might be more reasonable to make the value of a coalition equal to the size of the largest connected component inside this coalition. It is easy to adapt our techniques to obtain an algorithm running in time $n^{O(tw)}$ for this case; it would be interesting to see if a fixed-parameter tractable algorithm exists.

Source Code and Instances. We have made our source code, as well as the graphs used for the experiments, available on GitHub [110].

9 Conclusions

Treewidth and tree decompositions provide us with very versatile techniques that can be used to solve many problems efficiently in graphs of bounded treewidth. Even in cases where we are not operating on a graph of bounded treewidth directly, treewidth is often an important technical ingredient.

In this thesis, we have studied treewidth from both a theoretical and practical perspective. On the theoretical side, we have seen two classes of problems: one being the graph embedding problems where, although treewidth can be exploited, the fact that we have to track subsets of connected components makes it (under the ETH) impossible to improve upon our running time of $2^{\Theta(n/\log n)}$. In the case of problems on geometric intersection graphs, we have given a framework that allows us to use treewidth-based techniques to tackle problems, even though geometric intersection graphs can be dense and do not have bounded treewidth. In this case, we have obtained running times of the form $2^{O(n^{1-1/d})}$, which, just like our results for graph embedding problems, can not be improved upon unless the ETH fails.

Many problems in planar graphs can be solved in time $2^{O(\sqrt{n})}$ (and this is tight under the ETH). Since this is seemingly universal behaviour, this fact has been dubbed the "square root phenomenon". Our results could be viewed as two new "phenomena": a $n/\log n$-phenomenon for solving graph embedding problems in planar or H-minor free graphs, and a $n^{1-1/d}$-phenomenon for solving problems in intersection graphs of d-dimensional objects.

While on the theoretical side we have thus obtained tight bounds and algorithms, we have also explored how treewidth can be used in practice. Our results on using the GPU to compute treewidth shows that it is promising to exploit the massive processing power of GPUs to solve $\mathcal{NP}$-hard graph problems. Our results on computing Shapley values show that treewidth can be used to solve problems on real-world graphs from other fields (in this case social network analysis and game theory) of practical importance.

We conclude by giving several open problems related to the chapters of this thesis.

FPT algorithms for Graph Embedding Problems. Whereas we have shown that Subgraph Isomorphism (and several related problems) can be solved in $2^{O(n/\log n)}$ time on planar graphs, it would be very interesting to see if it is possible to obtain an FPT algorithm with running time $2^{O(k/\log k)}n^{O(1)}$ (where k denotes the number of vertices of the pattern graph). Our results can be combined with those of Fomin et al. [50] to obtain a randomized algorithm achieving this running time for the case where the pattern is connected. Very recently [96] it has been shown that for the case of a planar host graph (and a pattern that need not be connected), this running time can be achieved without randomization. It would be interesting to see whether this can be generalized to H-minor free graphs.

3-**Partition.** What is the best possible running time (under the ETH) for solving 3-Partition, in terms of the size of the input (given in unary)? It is fairly easy to obtain a $2^{O(\sqrt{n}\log n)}$-time algorithm. Is this tight under the ETH, or is an improvement possible?

Practical uses of Canonization. We have used *canonization* extensively in our algorithms for graph embedding problems. The theoretical results we obtain have large constants hidden in the exponent, so the theoretical bounds likely do not lead to practical improvements. However, it is still possible that using isomorphism tests to identify isomorphic partial solutions could be used to speed up dynamic programming, even for problems other than graph embedding. Which problems benefit from this and how can such isomorphism tests be implemented practically?

Finding better tree decompositions. For practical use in dynamic programming, it is important to have good tree decompositions. While Tamaki's algorithm [104] has been an impressive recent advance in this area, it only computes a tree decomposition with minimum width. It would be very helpful to have good heuristics or algorithms that compute tree decompositions that are optimized for use with a particular dynamic programming algorithm, for instance by minimizing the number of join bags (while keeping the width reasonable) or trying to create bags of maximum density. Here, the question is not only to find a good measure of the "cost" of a tree decomposition, but to find such a measure for which a decomposition minimizing it is feasible to compute. This will require balancing the accuracy of the measure with the complexity of computing it.

Centrality Measures. Our results on Shapley values show that two game-theoretic centrality measures can be practically computed on graphs of bounded treewidth. For which other centrality measures can we obtain feasible treewidth-based algorithms? There are many interesting theoretical and practical problems related to centrality measures, for instance computing how the graph could be modified to affect the centrality of a given vertex (allowing a certain player to strategically break/form bonds).

Appendix

A List of Problems

This appendix provides a list of ($\mathcal{NP}$-hard) problems that were studied in this thesis.

A.1. Classical Graph Problems

Vertex Cover
Given: A graph $G = (V, E)$ and an integer s.
Question: Does there exist a vertex set $X \subseteq V$ of size at most s, such that for any edge $(u, v) \in E$, $u \in X$ or $v \in X$ (i.e., X is a vertex cover)?

Independent Set
Given: A graph $G = (V, E)$ and an integer s.
Question: Does there exist a vertex set $X \subseteq V$ of size at least s, such that for all $u \neq v \in V$, $(u, v) \notin E$ (i.e., X is an independent set)?

Dominating Set
Given: A graph $G = (V, E)$ and an integer s.
Question: Does there exist a vertex set $X \subseteq V$ of size at most s, such that for all $v \in V$, if $v \notin X$ then there exists an edge $(u, v) \in E$ such that $u \in X$ (i.e., X is a dominating set)?

r-Dominating Set
Given: A graph $G = (V, E)$ and an integer s.
Question: Does there exist a vertex set $X \subseteq V$ of size at most s, such that for all $v \in V$, if $v \notin X$ then there exists a vertex $u \in X$ such that v is within distance at most r of u?

Feedback Vertex Set
Given: A graph $G = (V, E)$, a set of terminals $K \subseteq V$ and an integer s.
Question: Does there exist a vertex set $X \subseteq V$ of size at most s, such that $V \setminus X$ induces a forest (i.e., X is a feedback vertex set)?

Hamiltonian Path
Given: A graph $G = (V, E)$.
Question: Does G contain a (simple) path that visits all vertices of G?

Steiner Tree
Given: A graph $G = (V, E)$, a set of terminals $K \subseteq V$ and an integer s.
Question: Does there exist a vertex set $X \subseteq V$ of size at most s, such that $K \subseteq X$, and X induces a connected subgraph of G?

Hamiltonian Cycle
Given: A graph $G = (V, E)$.
Question: Does G contain a (simple) cycle that visits all vertices of G?

Connected Vertex Cover
Given: A graph $G = (V, E)$ and an integer s.
Question: Does there exist a vertex set $X \subseteq V$ of size at most s, such that for any edge $(u, v) \in E$, $u \in X$ or $v \in X$ and X induces a connected subgraph of G?

Connected Dominating Set
Given: A graph $G = (V, E)$ and an integer s.
Question: Does there exist a vertex set $X \subseteq V$ of size at most s, such that for all $v \in V$, if $v \notin X$ then there exists an edge $(u, v) \in E$ such that $u \in X$ and X induces a connected subgraph of G?

Connected Feedback Vertex Set
Given: A graph $G = (V, E)$, a set of terminals $K \subseteq V$ and an integer s.
Question: Does there exist a vertex set $X \subseteq V$ of size at most s, such that $V \setminus X$ induces a forest and X induces a connected subgraph of G?

Travelling Salesman (TSP)
Given: A graph $G = (V, E)$, a weight function $w : E \to \mathbb{R}_{\geq 0}$ and a number $s \in \mathbb{R}$.
Question: Does there exist a (simple) cycle in G, the total weight of whose edges is at most s and that visits all vertices?

Graph Colouring
Given: A graph $G = (V, E)$ and an integer k.
Question: Does there exist a function $f : V \to \{1, \ldots, k\}$ (a *k-colouring*) such that for all $(u, v) \in E$, $f(u) \neq f(v)$?

A.2. Graph Embedding Problems

Subgraph Isomorphism
Given: A graph $G = (V, E)$ (host) and a graph $P = (V, E)$ (pattern).
Question: Does P occur as a subgraph of G, that is, do there exist subsets $V' \subseteq V(G)$, $E' \subseteq E(G)$, such that P is isomorphic to (V', E')?

Induced Subgraph
Given: A graph $G = (V, E)$ (host) and a graph $P = (V, E)$ (pattern).
Question: Does P occur as an induced subgraph of G, that is, does there exist a subset $V' \subseteq V(G)$, such that P is isomorphic to $G[V']$?

Graph Minor
Given: A graph $G = (V, E)$ (host) and a graph $P = (V, E)$ (pattern).
Question: Is P a minor of G?

Induced Minor
Given: A graph $G = (V, E)$ (host) and a graph $P = (V, E)$ (pattern).
Question: Is P an induced minor of G?

Topological Minor
Given: A graph $G = (V, E)$ (host) and a graph $P = (V, E)$ (pattern).
Question: Is P a topological minor of G?

Immersion
Given: A graph $G = (V, E)$ (host) and a graph $P = (V, E)$ (pattern).
Question: Is P an immersion minor of G?

A.3. Miscellaneous Problems

Satisfiability
Given: A set of m clauses C over n variables $x_1, \ldots, x_n$.
Question: Does C have a satisfying assignment?

3-SAT
Given: A set of m clauses C over n variables $x_1, \ldots, x_n$, where each clause is the disjunction of exactly three variables.
Question: Does C have a satisfying assignment?

Exact Cover by 3-Sets (X3C)
Given: A set (universe) X with $X = 3q$ and a set C of three-element subsets of X.
Question: Does C have a q-element subset whose union is X?

3-Dimensional Matching
Given: Sets X, Y, Z, a set of triples $T \subseteq X \times Y \times Z$ and an integer s.
Question: Does there exist a subset $M \subseteq T$ of size s such that for any $(x_1, y_1, z_1) \neq (x_2, y_2, z_2) \in M$, it holds that $x_1 \neq x_2$, $y_1 \neq y_2$, $z_1 \neq z_2$?

3-Partition
Given: A collection C of $3n$ integers $x_1, \ldots, x_{3n}$.
Question: Does there exist a partition of C into n triples, such that the sum of each triple is equal to $\frac{1}{n}\Sigma_{i=1}^{3n} x_i$?

String Crafting
Given: String s, and n strings $t_1, \ldots, t_n$, with $|s| = \sum_{i=1}^{n} |t_i|$.
Question: Is there a permutation $\Pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$, such that the string $t^\Pi = t_{\Pi(1)} \cdot t_{\Pi(2)} \cdots t_\Pi$ fulfils that for each i, $1 \leq i \leq |s|$, $s(i) \geq t^\Pi(i)$?

Orthogonal Vector Crafting
Given: String s, and n strings $t_1, \ldots, t_n$, with $|s| = \sum_{i=1}^{n} |t_i|$.
Question: Is there a permutation $\Pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$, such that the string $t^\Pi = t_{\Pi(1)} \cdot t_{\Pi(2)} \cdots t_\Pi$ fulfils that for each i, $1 \leq i \leq |s|$, $s(i) \cdot t^\Pi(i) = 0$, i.e., when viewed as vectors, s is orthogonal to t^Π?

Minimum Size Tree Decomposition (k-MSTD)
Given: A graph G, integers k, s.
Question: Does G admit a tree decomposition $(T, \{X_t \mid t \in T\})$ of width at most k, such that $|V(T)| \leq s$?

Minimum Size Path Decomposition (k-MSPD)
Given: A graph G, integers k, s.
Question: Does G admit a path decomposition $(P, \{X_p \mid p \in P\})$ of width at most k, such that $|V(P)| \leq s$?

Intervalizing Coloured Graphs
Given: A graph $G = (V, E)$ together with a proper colouring $c : V \to \{1, 2, \ldots, k\}$.
Question: Is there an interval graph G' on the vertex set V, for which c is a proper colouring, and which is a supergraph of G?

Polyomino Packing
Given: A collection of polyominoes C and a target shape T.
Question: Is it possible to pack the polyominoes in C into the target shape of T?

Exact Polyomino Packing
Given: A collection of polyominoes C and a target shape T, such that the total area of polyominoes in C is equal to the area of T.
Question: Is it possible to pack the polyominoes in C into the target shape of T?

Bibliography

[1] Aaron B. Adcock, Blair D. Sullivan, and Michael W. Mahoney. Tree decompositions and social graphs. *Internet Mathematics*, 12(5):315–361, 2016.

[2] Isolde Adler, Frederic Dorn, Fedor V. Fomin, Ignasi Sau, and Dimitrios M. Thilikos. Fast minor testing in planar graphs. *Algorithmica*, 64(1):69–84, 2012.

[3] Noga Alon, Paul Seymour, and Robin Thomas. A separator theorem for nonplanar graphs. *Journal of the American Mathematical Society*, 3(4):801–808, 1990.

[4] Noga Alon, Raphy Yuster, and Uri Zwick. Color-coding: A new method for finding simple paths, cycles and other small subgraphs within large graphs. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing*, STOC '94, pages 326–335, New York, NY, USA, 1994. ACM.

[5] Carme Àlvarez, Josep Diáz, and Maria Serna. The hardness of intervalizing four colored caterpillars. *Discrete Mathematics*, 235(1):19 – 27, 2001.

[6] Rafael Amer and José Miguel Giménez. A connectivity game for graphs. *Mathematical Methods of Operations Research*, 60(3):453–470, 2004.

[7] Omid Amini, Fedor V. Fomin, and Saket Saurabh. Counting subgraphs via homomorphisms. *SIAM Journal on Discrete Mathematics*, 26(2):695–717, 2012.

[8] Austin Appleby. SMHasher. Accessed 12-04-2017. URL: `https://github.com/aappleby/smhasher`.

[9] Nikhil Bansal, Tjark Vredeveld, and Ruben van der Zwaan. Approximating vector scheduling: Almost matching upper and lower bounds. In Alberto Pardo and Alfredo Viola, editors, *LATIN 2014: Theoretical Informatics: 11th Latin American Symposium, Montevideo, Uruguay, March 31–April 4, 2014. Proceedings*, pages 47–59. Springer, 2014.

[10] John F. Banzhaf III. Weighted voting doesn't work: A mathematical analysis. *Rutgers L. Rev.*, 19:317, 1964.

[11] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[12] Hans L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25:1305–1317, 1996.

[13] Hans L. Bodlaender. Treewidth: Algorithmic techniques and results. In Igor Prívara and Peter Ružička, editors, *Mathematical Foundations of Computer Science 1997*, volume 1295 of *Lecture Notes in Computer Science*, pages 19–36. Springer Berlin Heidelberg, 1997.

[14] Hans L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209(1-2):1–45, 1998.

[15] Hans L. Bodlaender, Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Deterministic single exponential time algorithms for connectivity problems parameterized by treewidth. *Information and Computation*, 243:86–111, 2015.

[16] Hans L. Bodlaender and Babette de Fluiter. Intervalizing k-colored graphs. Technical Report UU-CS-1995-15, Department of Information and Computing Sciences, Utrecht University, 1995.

[17] Hans L. Bodlaender and Babette de Fluiter. On intervalizing k-colored graphs for DNA physical mapping. *Discrete Applied Mathematics*, 71(1):55 – 77, 1996.

[18] Hans L. Bodlaender, Pål Grønås Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshtanov, and Michał Pilipczuk. A $c^k n$ 5-approximation algorithm for treewidth. *SIAM Journal on Computing*, 45(2):317–378, 2016.

[19] Hans L. Bodlaender and Fedor V. Fomin. Tree decompositions with small cost. *Discrete Applied Mathematics*, 145(2):143–154, 2005.

[20] Hans L. Bodlaender, Fedor V. Fomin, Arie M.C.A. Koster, Dieter Kratsch, and Dimitrios M. Thilikos. On exact algorithms for treewidth. *ACM Transactions on Algorithms*, 9(1):12:1–12:23, December 2012.

[21] Hans L. Bodlaender and Arie M.C.A. Koster. Safe separators for treewidth. *Discrete Mathematics*, 306(3):337–350, 2006.

[22] Hans L. Bodlaender and Arie M.C.A. Koster. Combinatorial optimization on graphs of bounded treewidth. *The Computer Journal*, 51(3):255–269, 2008.

[23] Hans L. Bodlaender and Arie M.C.A. Koster. Treewidth computations II. Lower bounds. *Information and Computation*, 209(7):1103 – 1119, 2011.

[24] Hans L. Bodlaender and Rolf H. Möhring. The pathwidth and treewidth of cographs. *SIAM Journal on Discrete Mathematics*, 6(2):181–188, 1993.

[25] Hans L. Bodlaender and Jesper Nederlof. Subexponential time algorithms for finding small tree and path decompositions. In *23rd Annual European Symposium on Algorithms (ESA 2015)*, pages 179–190. Springer, 2015.

[26] Hans L. Bodlaender, Jesper Nederlof, and Tom C. van der Zanden. Subexponential time algorithms for embedding H-minor free graphs. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, volume 55 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:14, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[27] Hans L. Bodlaender and Udi Rotics. Computing the treewidth and the minimum fill-in with the modular decomposition. *Algorithmica*, 36(4):375–408, 2003.

[28] Hans L. Bodlaender and Johan M. M. van Rooij. Exact algorithms for intervalizing coloured graphs. *Theory of Computing Systems*, 58(2):273–286, 2016.

[29] Paul Bonsma. Surface split decompositions and subgraph isomorphism in graphs on surfaces. *arXiv preprint arXiv:1109.4554*, 2011.

[30] Michael Brand. Small polyomino packing. *Information Processing Letters*, 126:30–34, 2017.

[31] Heinz Breu and David G. Kirkpatrick. Unit disk graph recognition is NP-hard. *Computational Geometry*, 9(1):3 – 24, 1998.

[32] François Clautiaux, Jacques Carlier, Aziz Moukrim, and Stéphane Nègre. New lower and upper bounds for graph treewidth. In Klaus Jansen, Marian Margraf, Monaldo Mastrolilli, and José D. P. Rolim, editors, *Experimental and Efficient Algorithms: Second International Workshop, WEA 2003, Ascona, Switzerland, May 26–28, 2003 Proceedings*, pages 70–80, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

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

[34] Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 1st edition, 2015.

[35] Marek Cygan, Jakub Pachocki, and Arkadiusz Socała. The hardness of subgraph isomorphism. *arXiv preprint arXiv:1504.02876*, 2015.

[36] Mark de Berg, Hans L. Bodlaender, Sándor Kisfaludi-Bak, Dániel Marx, and Tom C. van der Zanden. A framework for ETH-tight algorithms and lower bounds in geometric intersection graphs. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2018, pages 574–586, New York, NY, USA, 2018. ACM.

[37] Holger Dell, Thore Husfeldt, Bart M.P. Jansen, Petteri Kaski, Christian Komusiewicz, and Frances A. Rosamond. The parameterized algorithms and computational experiments challenge (PACE). Accessed 05-04-2017. URL: `https://pacechallenge.wordpress.com/pace-2016/track-a-treewidth/`.

[38] Holger Dell, Thore Husfeldt, Bart M.P. Jansen, Petteri Kaski, Christian Komusiewicz, and Frances A. Rosamond. The first parameterized algorithms and computational experiments challenge. In Jiong Guo and Danny Hermelin, editors, *11th International Symposium on Parameterized and Exact Computation (IPEC 2016)*, volume 63 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:9, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[39] Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. The pace 2017 parameterized algorithms and computational experiments challenge: The second iteration. In Daniel Lokshtanov and Naomi Nishimura, editors, *12th International Symposium on Parameterized and Exact Computation (IPEC 2017)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 30:1–30:12, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[40] Erik D. Demaine and Martin L. Demaine. Jigsaw puzzles, edge matching, and polyomino packing: Connections and complexity. *Graphs and Combinatorics*, 23(1):195–208, 2007.

[41] Erik D. Demaine and Mohammad Taghi Hajiaghayi. Bidimensionality: New connections between FPT algorithms and PTASs. In *Proceedings of the sixteenth annual ACM-SIAM Symposium on Discrete Algorithms*, pages 590–601. Society for Industrial and Applied Mathematics, 2005.

[42] Karine Descormiers and Carlo Morselli. Alliances, conflicts, and contradictions in Montreal's street gang landscape. *International Criminal Justice Review*, 21(3):297–314, 2011.

[43] Frederic Dorn. Planar subgraph isomorphism revisited. *arXiv preprint arXiv:0909.4692*, 2009.

[44] P. Alex Dow. *Search Algorithms for Exact Treewidth.* PhD thesis, University of California, Los Angeles, CA, USA, 2010. AAI3405666.

[45] P. Alex Dow and Richard E. Korf. Best-first search for treewidth. In *Proceedings of the 22nd National Conference on Artificial Intelligence - Volume 2*, AAAI'07, pages 1146–1151. AAAI Press, 2007.

[46] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity.* Monographs in Computer Science. Springer-Verlag New York, 1999.

[47] David Eppstein. Subgraph isomorphism in planar graphs and related problems. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '95, pages 632–640, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[48] Sean F. Everton. *Disrupting Dark Networks*, volume 34. Cambridge University Press, 2012.

[49] Fedor V. Fomin, Alexander Golovnev, Alexander S. Kulikov, and Ivan Mihajlin. Tight bounds for subgraph isomorphism and graph homomorphism. *arXiv preprint arXiv:1507.03738*, 2015.

[50] Fedor V. Fomin, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. Subexponential parameterized algorithms for planar and apex-minor-free graphs via low treewidth pattern covering. In *Foundations of Computer Science (FOCS), 2016 IEEE 57th Annual Symposium on*, pages 515–524. IEEE, 2016.

[51] Fedor V. Fomin, Daniel Lokshtanov, Fahad Panolan, Saket Saurabh, and Meirav Zehavi. Finding, hitting and packing cycles in subexponential time on unit disk graphs. In *Proceedings of the 44th International Colloquium on Automata, Languages, and Programming, ICALP 2017*, volume 80 of *LIPICS*, pages 65:1–65:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017.

[52] Fedor V. Fomin, Sang-il Oum, and Dimitrios M. Thilikos. Rank-width and tree-width of H-minor-free graphs. *European Journal of Combinatorics*, 31(7):1617 – 1628, 2010.

[53] Bin Fu. Theory and application of width bounded geometric separators. *Journal of Computer and System Sciences*, 77(2):379 – 392, 2011.

[54] Jakub Gajarský, Petr Hliněný, Jan Obdržálek, Sebastian Ordyniak, Felix Reidl, Peter Rossmanith, Fernando Sanchez Villaamil, and Somnath Sikdar. Kernelization using structural parameters on sparse graph classes. In *21st Annual European Symposium on Algorithms (ESA 2013)*, pages 529–540. Springer, 2013.

[55] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* W. H. Freeman & Co., New York, NY, USA, 1979.

[56] Francesco Scarcello Gianluigi Greco, Francesco Lupia. The tractability of the Shapley value over bounded treewidth matching games. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 1046–1052, 2017.

[57] Vibhav Gogate and Rina Dechter. A complete anytime algorithm for treewidth. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, UAI '04, pages 201–208, Arlington, Virginia, United States, 2004. AUAI Press.

[58] Arvind Gupta and Naomi Nishimura. The complexity of subgraph isomorphism for classes of partial k-trees. *Theoretical Computer Science*, 164(1):287–298, 1996.

[59] Alexander Gutfraind and Michael Genkin. A graph database framework for covert network analysis: An application to the Islamic State network in Europe. *Social Networks*, 51:178–188, 2017.

[60] MohammadTaghi Hajiaghayi and Naomi Nishimura. Subgraph isomorphism, log-bounded fragmentation, and graphs of (locally) bounded treewidth. *Journal of Computer and System Sciences*, 73(5):755–768, 2007.

[61] Robert A. Hanneman and Mark Riddle. *Introduction to Social Network Methods.* 2005. Accessed 19-12-2018. URL: `http://faculty.ucr.edu/~hanneman/nettext/`.

[62] Sariel Har-Peled and Kent Quanrud. Approximation algorithms for polynomial-expansion and low-density graphs. In *23rd Annual European Symposium on Algorithms (ESA 2015)*, volume 9294 of *Lecture Notes in Computer Science*, pages 717–728. Springer, 2015.

[63] David Harvey and Joris van der Hoeven. Integer multiplication in time O(n log n). hal-02070778, 2019.

[64] Robert A. Hearn and Erik D. Demaine. *Games, Puzzles, and Computation.* CRC Press, 2009.

[65] Alexander Hein and Arie M.C.A. Koster. An experimental evaluation of treewidth at most four reductions. In Panos M. Pardalos and Steffen Rebennack, editors, *Proceedings of the 10th International Symposium on Experimental and Efficient Algorithms, SEA 2011*, volume 6630 of *Lecture Notes in Computer Science*, pages 218–229. Springer Verlag, 2011.

[66] Michael Held and Richard M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.

[67] Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367–375, 2001.

[68] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512 – 530, 2001.

[69] Hiro Ito and Masakazu Kadoshita. Tractability and intractability of problems on unit disk graphs parameterized by domain area. In *Proceedings of the 9th International Symposium on Operations Research and Its Applications, ISORA 2010*, pages 120–127, 2010.

[70] Klaus Jansen, Felix Land, and Kati Land. Bounding the running time of algorithms for scheduling and packing problems. In Frank Dehne, Roberto Solis-Oba, and Jörg-Rüdiger Sack, editors, *Algorithms and Data Structures: 13th International Symposium, WADS 2013, London, ON, Canada, August 12-14, 2013. Proceedings*, pages 439–450. Springer, 2013.

[71] Ross J. Kang and Tobias Müller. Sphere and dot product representations of graphs. *Discrete & Computational Geometry*, 47(3):548–568, 2012.

[72] Haim Kaplan and Ron Shamir. Bounded degree interval sandwich problems. *Algorithmica*, 24(2):96–104, 1999.

[73] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger, editors, *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations*, pages 85–103. Springer, 1972.

[74] Ken-ichi Kawarabayashi, Yusuke Kobayashi, and Bruce Reed. The disjoint paths problem in quadratic time. *Journal of Combinatorial Theory, Series B*, 102(2):424 – 435, 2012.

[75] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better Bloom filter. In Yossi Azar and Thomas Erlebach, editors, *14th*

Annual European Symposium on Algorithms (ESA 2006), pages 456–467, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[76] Sándor Kisfaludi-Bak and Tom C. van der Zanden. On the exact complexity of Hamiltonian cycle and q-colouring in disk graphs. In Dimitris Fotakis, Aris Pagourtzis, and Vangelis Th. Paschos, editors, *Algorithms and Complexity*, pages 369–380, Cham, 2017. Springer International Publishing.

[77] David A. Klarner and Ronald L. Rivest. A procedure for improving the upper bound for the number of n-ominoes. *Canadian Journal of Mathematics*, 25(3):585–602, 1973.

[78] Stuart Koschade. A social network analysis of Jemaah Islamiyah: The applications to counterterrorism and intelligence. *Studies in Conflict & Terrorism*, 29(6):559–575, 2006.

[79] Valdis E. Krebs. Mapping networks of terrorist cells. *Connections*, 24(3):43–52, 2002.

[80] Alexander Langer. *Fast Algorithms for Decomposable Graphs.* PhD thesis, RWTH Aachen, 2013.

[81] Leonid A. Levin. Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973.

[82] Roy H. A. Lindelauf, Herbert J. M. Hamers, and Bart G. M. Husslage. Cooperative game theoretic centrality analysis of terrorist networks: The cases of Jemaah Islamiyah and Al Qaeda. *European Journal of Operational Research*, 229(1):230–238, 2013.

[83] Andrzej Lingas. Subgraph isomorphism for biconnected outerplanar graphs in cubic time. *Theoretical Computer Science*, 63(3):295 – 302, 1989.

[84] Richard J. Lipton and Robert E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[85] Richard J. Lipton and Robert Endre Tarjan. Applications of a planar separator theorem. *SIAM Journal on Computing*, 9(3):615–627, 1980.

[86] Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. Lower bounds based on the exponential time hypothesis. *Bulletin of the EATCS*, (105):41–72, 2011.

[87] Dániel Marx. What's next? Future directions in parameterized complexity. In Hans L. Bodlaender, Rod Downey, Fedor V. Fomin, and Dániel Marx, editors, *The Multivariate Algorithmic Revolution and Beyond: Essays Dedicated to Michael R. Fellows on the Occasion of His 60th Birthday*, pages 469–496. Springer, 2012.

[88] Dániel Marx. The square root phenomenon in planar graphs. In Michael Fellows, Xuehou Tan, and Binhai Zhu, editors, *Frontiers in Algorithmics and Algorithmic Aspects in Information and Management: Third Joint International Conference, FAW-AAIM 2013, Dalian, China, June 26-28, 2013. Proceedings.*, pages 1–1. Springer, 2013.

[89] Dániel Marx and Michał Pilipczuk. Everything you always wanted to know about the parameterized complexity of subgraph isomorphism (but were afraid to ask). *arXiv preprint arXiv:1307.2187*, 2013.

[90] Dániel Marx and Michal Pilipczuk. Optimal parameterized algorithms for planar facility location problems using Voronoi diagrams. In *23rd Annual European Symposium on Algorithms (ESA 2015)*, volume 9294 of *Lecture Notes in Computer Science*, pages 865–877. Springer, 2015.

[91] Dániel Marx and Anastasios Sidiropoulos. The limited blessing of low dimensionality: When $1 - 1/d$ is the best possible exponent for d-dimensional geometric problems. In *Proceedings of the 30th Annual Symposium on Computational Geometry, SOCG 2014*, pages 67–76. ACM, 2014.

[92] Jiří Matoušek and Robin Thomas. On the complexity of finding iso-and other morphisms for partial k-trees. *Discrete Mathematics*, 108(1):343–364, 1992.

[93] Tomasz P. Michalak, Karthik V. Aadithya, Piotr L. Szczepanski, Balaraman Ravindran, and Nicholas R. Jennings. Efficient computation of the Shapley value for game-theoretic network centrality. *Journal of Artificial Intelligence Research*, 46:607–650, 2013.

[94] Tomasz P. Michalak, Talal Rahwan, Nicholas R. Jennings, Piotr L. Szczepański, Oskar Skibski, Ramasuri Narayanam, and Michael J. Wooldridge. Computational analysis of connectivity games with applications to the investigation of terrorist networks. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 293–301. AAAI Press, 2013.

[95] Mitchell Centre for Social Network Analysis. Italian gangs network. UCINET website. Accessed 26-07-2018. URL: `https://sites.google.com/site/ucinetsoftware/datasets/covert-networks/italiangangs`.

[96] Jesper Nederlof. Detecting and counting small patterns in planar graphs in subexponential parameterized time. *arXiv preprint arXiv:1904.11285*, 2019.

[97] NVIDIA. NVIDIA GeForce GTX 1080 whitepaper. Accessed 10-04-2017. URL: `http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf`.

[98] NVIDIA. NVIDIA's next generation CUDA compute architecture: FERMI. Accessed 12-04-2017. URL: `http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf`.

[99] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[100] Neil Robertson and Paul D. Seymour. Graph minors. XIII. The disjoint paths problem. *Journal of Combinatorial Theory, Series B*, 63(1):65 – 110, 1995.

[101] Lloyd S. Shapley. A value for n-person games. *Contributions to the Theory of Games*, 2(28):307–317, 1953.

[102] Warren D. Smith and Nicholas C. Wormald. Geometric separator theorems & applications. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science, FOCS 1998*, pages 232–243. IEEE Computer Society, 1998.

[103] Maciej M. Sysło. The subgraph isomorphism problem for outerplanar graphs. *Theoretical Computer Science*, 17(1):91 – 97, 1982.

[104] Hisao Tamaki. Positive-instance driven dynamic programming for treewidth. In Kirk Pruhs and Christian Sohler, editors, *25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 68:1–68:13, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[105] Dimitrios M. Thilikos. Graph minors and parameterized algorithm design. In Hans L. Bodlaender, Rod Downey, Fedor V. Fomin, and Dániel Marx, editors, *The Multivariate Algorithmic Revolution and Beyond*, pages 228–256. Springer-Verlag, Berlin, Heidelberg, 2012.

[106] Julian R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

[107] Tjeerd van Campen, Herbert J. M. Hamers, Bart G. M. Husslage, and Roy H. A. Lindelauf. A new approximation method for the Shapley value applied to the WTC 9/11 terrorist attack. *Social Network Analysis and Mining*, 8(1):3, Dec 2017.

[108] Frank van den Eijkhof, Hans L. Bodlaender, and Arie M.C.A. Koster. Safe reduction rules for weighted treewidth. *Algorithmica*, 47(2):139–158, 2007.

[109] Luuk W. van der Graaff. Dynamic programming on nice tree decompositions. Master's thesis, Utrecht University, 2015.

[110] Tom C. van der Zanden. ShapleyTreewidth. Accessed 14-01-2019. URL: `https://github.com/TomvdZanden/ShapleyTreewidth`.

[111] Tom C. van der Zanden and Hans L. Bodlaender. BZTreewidth. Accessed 11-04-2017. URL: `https://github.com/TomvdZanden/BZTreewidth`.

[112] Tom C. van der Zanden and Hans L. Bodlaender. GPGPU treewidth. Accessed 21-04-2017. URL: `https://github.com/TomvdZanden/GPGPU-Treewidth`.

[113] Thomas C. van Dijk, Jan-Pieter van den Heuvel, and Wouter Slob. Computing treewidth with LibTW, 2006. Accessed 16-06-2017. URL: `http://www.treewidth.com/treewidth`.

[114] Johan M.M. van Rooij, Hans L. Bodlaender, and Peter Rossmanith. Dynamic programming on tree decompositions using generalised fast subset convolution. In *17th Annual European Symposium on Algorithms (ESA 2009)*, pages 566–577. Springer, 2009.

[115] Margaret R. Weeks, Scott Clair, Stephen P. Borgatti, Kim Radda, and Jean J. Schensul. Social networks of drug users in high-risk sites: Finding the connections. *AIDS and Behavior*, 6(2):193–206, 2002.

[116] Mingyu Xiao and Hiroshi Nagamochi. Exact algorithms for maximum independent set. *Information and Computation*, 255:126 – 146, 2017.

[117] Yang Yuan. A fast parallel branch and bound algorithm for treewidth. In *2011 IEEE 23rd International Conference on Tools with Artificial Intelligence*, pages 472–479, Nov 2011.

[118] Rong Zhou and Eric A. Hansen. Combining breadth-first and depth-first strategies in searching for treewidth. In *Proceedings of the 21st International Joint Conference on Artifical Intelligence*, IJCAI'09, pages 640–645, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

Nederlandse Samenvatting

Voor het oplossen van alledaagse – en minder alledaagse – problemen zijn *algoritmen* van groot belang. Een algoritme is een precies omschreven, stapsgewijze manier om een bepaalde taak uit te voeren. Zo kan men denken aan een algoritme dat gebruikt wordt door een routeplanner om de beste route uit te rekenen, of aan het sorteeralgoritme dat gegevens in een spreadsheet ordent. Een algoritme is vaak een computerprogramma, maar hoeft dat niet per se te zijn: een algoritme kan ook met de hand worden uitgevoerd – zoals een vermenigvuldigingsalgoritme dat kinderen op de basisschool leren.

Het aantal stappen dat nodig is om een algoritme uit te voeren (c.q. de tijd die het kost) is de *complexiteit*. De complexiteit geeft aan hoe de looptijd van een algoritme afhangt van de grootte van de invoer: zo zal het uitrekenen van een (kortste) route van Utrecht naar Madrid meer tijd kosten dan het uitrekenen van een route van Utrecht naar Eindhoven aangezien er meer informatie moet worden verwerkt (de kaart van heel Europa in plaats van de kaart van alleen Nederland).

De complexiteit van een algoritme bepaalt hoe goed het schaalt naar grotere invoeren: het uitrekenen van een product door herhaald optellen is redelijk te doen voor kleine getallen zoals $5 \times 6 = 6 + 6 + 6 + 6 + 6 = 30$, maar voor het uitrekenen van een groter product zoals 153×12 is het beter om een andere methode (= algoritme) te gebruiken, zoals rekenen met tientallen: $153 \times 12 = 153 \times 10 + 153 \times 2 = 1530 + 306 = 1836$.

In dit proefschrift kijken we naar problemen op grafen: een *graaf* is een verzameling punten (de *knopen*) die verbonden zijn middels lijnstukken (de *kanten*). Een graaf kan bijvoorbeeld een wegennetwerk modelleren (waarbij knopen steden zijn en kanten wegen) of een sociaal netwerk (waarin knopen mensen zijn en kanten vriendschappen).

Veel ogenschijnlijk simpele problemen op grafen zijn moeilijk op te lossen. Een voorbeeld daarvan is Independent Set: de vraag is om in een graaf k knopen vinden, waarvan geen enkel paar door een kant verbonden is. Dit probleem is $\mathcal{NP}$-volledig, en het best bekende algoritme heeft een looptijd die exponentieel is in het aantal knopen n van de graaf.

Als een graaf een bepaalde onderliggende structuur heeft, is het soms mogelijk om die te gebruiken om een probleem sneller op te lossen. Zo zijn veel problemen (waaronder Independent Set) in lineaire tijd op te lossen als de graaf de structuur van een *boom* heeft (d.w.z.: een graaf waar iedere knoop te bereiken is vanuit iedere andere knoop door een uniek (kortste) pad). Vaak zijn zulke problemen ook op te lossen als een graaf op een boom *lijkt*, iets wat te formaliseren is in het begrip *boombreedte* (treewidth). Boombreedte hangt nauw samen met het begrip *boomdecompositie*. Een boomdecompositie is een recursieve opdeling van de graaf voor middel van separatoren: kleine verzamelingen knopen die de graaf opdelen in qua grootte gebalanceerde verbonden componenten.

Een *planaire graaf* is een graaf die in het platte vlak getekend kan worden zonder dat kanten elkaar kruisen. Het is welbekend dat planaire grafen gebalanceerde separatoren van grootte $O(\sqrt{n})$ hebben (waarbij n het aantal knopen van de graaf is). Hieruit volgt dat de boombreedte van een planaire graaf, ook $O(\sqrt{n})$ is. Door gebruik te maken van standaardtechnieken voor dynamisch programmeren op boomdecomposities, is het mogelijk om veel problemen in $2^{O(\sqrt{n})}$ tijd op te lossen in planaire grafen — hieronder ook Independent Set. Als men aanneemt dat Independent Set in algemene grafen niet op te lossen is in $2^{o(n)}$ tijd (iets wat volgt uit aanname van de *Exponential Time Hypothesis* (ETH)), dan volgt daaruit verrassenderwijs dat Independent Set in planaire grafen ook niet is op te lossen in tijd $2^{o(\sqrt{n})}$.

Dit fenomeen blijkt bij veel (moeilijke) problemen voor te komen: als het best bekende algoritme in algemene grafen looptijd $2^{O(n)}$ heeft, is het probleem in planaire grafen vaak op te lossen in tijd $2^{O(\sqrt{n})}$ en is het (tenzij de ETH niet waar blijkt te zijn) niet mogelijk dat er een sneller algoritme bestaat. Dit staat bekend als het *square root phenomenon*.

In het eerste deel van dit proefschrift kijken we naar *graafinbeddingsproblemen*. Het meest bekende voorbeeld daarvan is Subgraph Isomorphism: gegeven een graaf G (de *host*) moeten we bepalen of een (kleinere) graaf P (het *pattern*) ergens in G voorkomt als subgraaf (dus, is het mogelijk om door kanten en knopen van G weg te gooien, de graaf P te krijgen?). We geven antwoord op de vraag of dit probleem in planaire grafen (en meer algemeen: in H-minor-vrije grafen) sneller op te lossen is dan in het algemeen.

Enerzijds laten we zien dat het probleem voor een planaire host G (met n knopen) op te lossen is in tijd $2^{O(n/\log n)}$. Dit is sneller dan het beste (onder de ETH) algoritme voor algemene grafen (met looptijd $2^{O(n\log n)}$), maar langzamer dan de $2^{O(\sqrt{n})}$ die men misschien zou verwachten op basis van het square root phenomenon. Anderzijds laten we zien dat dit optimaal is: als de ETH geldt, dan is er geen algoritme voor Subgraph Isomorphism in planaire grafen met looptijd $2^{o(n/\log n)}$ — het square root phenomenon gaat dus niét op voor Subgraph Isomorphism.

De techniek voor het algoritme is gebaseerd op traditioneel dynamisch programmeren op boomdecomposities, gecombineerd met *canonizatie*. Door te herkennen dat bepaalde partiële oplossingen isomorf (d.w.z.: dat twee (deel-)grafen met een mogelijk een verschillende set knopen, toch qua structuur dezelfde graaf voorstellen) aan elkaar zijn, is het mogelijk om, ook indien de graaf uiteen valt in een zeer groot aantal kleine componenten, het aantal opties dat bekeken moet worden terug te brengen van $2^{\Theta(n)}$ naar $2^{\Theta(n/\log n)}$.

Voor het bewijzen van de ondergrens maken we gebruik van binaire encodering, om kleine bouwsteentjes te maken die corresponderen met een specifieke clause of variabele uit een satisfiability-formule. We introduceren twee hulp-problemen, String Crafting en Orthogonal Vector Crafting, die als handig startpunt voor reducties om ondergrenzen van de vorm $2^{\Omega(n/\log n)}$ te laten zien kunnen dienen.

Gebruikmakend van deze technieken, laten we zien dat de looptijd en ondergrens $2^{\Theta(n/\log n)}$ geldt voor een groot aantal graafinbeddingsproblemen in planaire grafen, zoals Subgraph Isomorphism, Graph Minor en Topological Minor. Ter afsluiting van het eerste deel geven we een klein intermezzo, waarin we onze technieken voor graafinbedding toepassen op het oplossen van polyomino-puzzels.

In het tweede deel kijken we naar problemen in geometrische doorsnijdingsgrafen. Een geometrische doorsnijdingsgraaf ontstaat door te kijken naar een verzameling objecten in de ruimte, deze objecten te identificeren met knopen van een graaf, en een kant tussen twee knopen toe te voegen als de corresponderende objecten elkaar in de ruimte doorsnijden.

In tegenstelling tot een planaire graaf, kan een geometrische doorsnijdingsgraaf dicht zijn (d.w.z.: veel kanten hebben) en hoge treewidth hebben. Daarom is het verrassend dat er zich in dit soort grafen toch een soort square root phenomenon voordoet. Het is bijvoorbeeld bekend dat Independent Set in $2^{O(\sqrt{n})}$ tijd kan worden opgelost in doorsnijdingsgrafen van cirkels met straal 1 in het platte vlak, en er zijn generalisaties van dit soort resultaten bekend naar hogere dimensies, waarbij problemen in tijd $n^{O(n^{1-1/d})}$ of $2^{O(n^{1-1/d})}$ kunnen worden opgelost voor doorsnijdingsgrafen van bepaalde klassen d-dimensionale objecten.

Wij geven een framework, waarmee het mogelijk is bestaande algoritmen voor dynamisch programmeren op boomdecomposities aan te passen voor gebruik in dit soort grafen. Hiermee kunnen veel problemen, waaronder Independent Set, Dominating Set, Steiner Tree en Hamiltonian Cycle, worden opgelost in $2^{O(n^{1-1/d})}$ tijd in doorsnijdingsgrafen van een zeer algemene klasse van d-dimensionale objecten. Het framework is gebaseerd op een boomdecompositie van een samengetrokken graaf, waarbij elke knoop van de boomdecompositie correspondeert met een separator bestaande uit een aantal cliques, die onder een bepaalde wegingsfunctie een laag gewicht hebben.

Het blijkt dat deze looptijd van $2^{O(n^{1-1/d})}$ optimaal is: het is mogelijk (alhoewel dit geen onderdeel is van dit proefschrift) om te laten zien dat, uitgaande van de ETH, er geen algoritme voor deze problemen bestaat met looptijd $2^{o(n^{1-1/d})}$, zelfs in een zeer beperkte klasse van doorsnijdingsgrafen.

Er is een interessant contrast tussen de resultaten over graafimbeddingsproblemen en de resultaten over doorsnijdingsgrafen. Men zou namelijk misschien verwachten dat graafimbeddingsproblemen in planaire grafen in $2^{O(\sqrt{n})}$ tijd kunnen worden opgelost, maar dit blijkt niet zo te zijn. Daarentegen zou men voor problemen in doorsnijdingsgrafen juist géén snellere algoritmen verwachten (aangezien de boombreedte van deze grafen onbegrensd is), en blijken deze juist wel te bestaan.

In het derde en laatste deel van dit proefschrift kijken we naar praktische toepassingen van boombreedte en algoritmen op boomdecomposities. Eerst kijken we naar parallelle algoritmen voor het berekenen van een boomdecompositie, en hoe zo'n algoritme op de GPU geïmplementeerd kan worden. Het berekenen van een (optimale) boomdecompositie kost in het algemeen exponentiële tijd, dus daarom is het aantrekkelijk om parallellisme te gebruiken om grotere instanties toch binnen redelijke tijd op te kunnen lossen. Het gebruik van een *Graphical Processing Unit* (GPU) is zeer interessant, omdat GPU's een hele grote hoeveelheid rekenkracht bieden tegen een relatief lage prijs. GPU's zijn eigenlijk bedoeld voor het uitvoeren van videotaken op een computer, maar kunnen middels *General Purpose computing on Graphical Processing Units* (GPGPU) toch worden ingezet voor andere taken. Dit brengt wel bijzondere uitdagingen met zich mee.

We laten zien hoe een klassiek DP-algoritme voor het berekenen van boombreedte kan worden aangepast voor gebruik op de GPU, en dat het mogelijk is om hiermee een grote versnelling in de rekentijd te halen. Ook experimenteren we met enkele optimalisaties en de implementatie van de heuristiek "minor-min width" op de GPU.

De tweede praktische toepassing die we bekijken is het uitrekenen van *Shapleywaar-*

den van connectiviteitsspellen. De motivatie hiervoor is dat, als we een sociaal netwerk bekijken (een graaf waarin de knopen personen zijn en de kanten relaties tussen personen), we dan – op basis van de structuur van het netwerk – een rangschikking kunnen maken van hoe belangrijk iedere persoon is. Een dergelijke rangschikking kan gemaakt worden met behulp van een *centraliteitsmaat*. Er bestaan veel verschillende centraliteitsmaten, en een interessante groep daarvan zijn de speltheoretische centraliteitsmaten. We bekijken enkele voorbeelden van speltheoretische centraliteitsmaten, gebaseerd op de Shapleywaarden van enkele connectiviteitsspellen. Door gebruik te maken van een boomdecompositie van de graaf, kunnen we deze maten efficiënt uitrekenen. We passen dit onder andere toe op een "sociaal" netwerk van Al-Qaedaterroristen, die betrokken waren bij de aanslagen van 11 september. Voor dit netwerk was het eerder nog niet mogelijk om de Shapleywaarde (exact) uit te rekenen, maar met onze methode is dit goed mogelijk.

Curriculum Vitae

Tom C. van der Zanden was born on December 29, 1992 in Nieuwegein, The Netherlands. In 2010, he obtained his preparatory scientific education (VWO) degree from the Utrechts Stedelijk Gymnasium. In 2013, he obtained bachelor degrees (Cum Laude) in both Mathematics and Computer Science from Utrecht University. In 2015, he obtained his master's degree (in Computing Science, also Cum Laude) from the same university, with his master's thesis entitled "Parameterized Complexity of Graph Constraint Logic". After finishing his master's, Tom started his PhD with Hans L. Bodlaender, in the Department of Information and Computing Sciences at Utrecht University.